RUNNING TESTS IN PARALLEL IN THE SAME FOLDER

One other type of parallelism to take into account is what happens if you try to run multiple automated tests in parallel against the same Terraform folder. For example, perhaps you'd want to run several different tests against *examples/hello-world-app*, where each test sets different values for the input variables before running `terraform apply`. If you try this, you'll hit a problem: the tests will end up clashing because they all try to run `terraform init` and end up overwriting one another's *.terraform* folder and Terraform state files.

If you want to run multiple tests against the same folder in parallel, the easiest solution is to have each test copy that folder to a unique temporary folder, and run Terraform in the temporary folder to avoid conflicts. Terratest, of course, has a built-in helper to do this for you, and it even does it in a way that ensures that relative file paths within those Terraform modules work correctly: check out the `test_structure.CopyTerraformFolderToTemp` method and its documentation for details.

Integration Tests

Now that you've got some unit tests in place, let's move on to integration tests. Again, it's helpful to start with the Ruby web server example to build up some intuition that you can later apply to the Terraform code. To do an integration test of the Ruby web server code, you need to do the following:

1. Run the web server on localhost so that it listens on a port.
2. Send HTTP requests to the web server.
3. Validate you get back the responses you expect.

Let's create a helper method in *web-server-test.rb* that implements these steps:

```
def do_integration_test(path, check_response)
  port = 8000
  server = WEBrick::HTTPServer.new :Port => port
  server.mount '/', WebServer

  begin
    # Start the web server in a separate thread so it
```

```
      # doesn't block the test
      thread = Thread.new do
        server.start
      end

      # Make an HTTP request to the web server at the
      # specified path
      uri = URI("http://localhost:#{port}#{path}")
      response = Net::HTTP.get_response(uri)

      # Use the specified check_response lambda to validate
      # the response
      check_response.call(response)
    ensure
      # Shut the server and thread down at the end of the
      # test
      server.shutdown
      thread.join
    end
  end
```

The `do_integration_test` method configures the web server on port 8000, starts it in a background thread (so the web server doesn't block the test from running), sends an HTTP GET to the `path` specified, passes the HTTP response to the specified `check_response` function for validation, and at the end of the test, shuts down the web server. Here's how you can use this method to write an integration test for the / endpoint of the web server:

```
  def test_integration_hello
    do_integration_test('/', lambda { |response|
      assert_equal(200, response.code.to_i)
      assert_equal('text/plain', response['Content-Type'])
      assert_equal('Hello, World', response.body)
    })
  end
```

This method calls the `do_integration_test` method with the / path and passes it a lambda (essentially, an inline function) that checks the response was a 200 OK with the body "Hello, World." The integration tests for the other endpoints are analogous. Let's run all of the tests:

```
$ ruby web-server-test.rb

(...)

Finished in 0.221561 seconds.
--------------------------------------------
8 tests, 24 assertions, 0 failures, 0 errors
100% passed
--------------------------------------------
```

Note that before, with solely unit tests, the test suite took 0.000572 seconds to run, but now, with integration tests, it takes 0.221561 seconds, a slowdown of roughly 387 times. Of course, 0.221561 seconds is still blazing fast, but that's only because the Ruby web server code is intentionally a minimal example that doesn't do much. The important thing here is not the absolute numbers but the relative trend: integration tests are typically slower than unit tests. I'll come back to this point later.

Let's now turn our attention to integration tests for Terraform code. If a "unit" in Terraform is a single module, an integration test that validates how several units work together would need to deploy several modules and see that they work correctly. In the previous section, you deployed the "Hello, World" app example with mock data instead of a real MySQL DB. For an integration test, let's deploy the MySQL module for real and make sure the "Hello, World" app integrates with it correctly. You should already have just such code under *live/stage/data-stores/mysql* and *live/stage/services/hello-world-app*. That is, you can create an integration test for (parts of) your staging environment.

Of course, as mentioned earlier in the chapter, all automated tests should run in an isolated AWS account. So while you're testing the code that is meant for staging, you should authenticate to an isolated testing account and run the tests there. If your modules have anything in them hardcoded for the staging environment, this is the time to make those values configurable so you can inject test-friendly values. In particular, in *live/stage/data-stores/mysql/variables.tf*, expose the database name via a new `db_name` input variable:

```
variable "db_name" {
  description = "The name to use for the database"
  type        = string
  default     = "example_database_stage"
}
```

Pass that value through to the `mysql` module in *live/stage/data-stores/mysql/main.tf*:

```
module "mysql" {
  source = "../../../../modules/data-stores/mysql"

  db_name     = var.db_name
  db_username = var.db_username
  db_password = var.db_password
}
```

Let’s now create the skeleton of the integration test in *test/hello_world_integration_test.go* and fill in the implementation details later:

```
// Replace these with the proper paths to your modules
const dbDirStage = "../live/stage/data-stores/mysql"
const appDirStage = "../live/stage/services/hello-world-app"

func TestHelloWorldAppStage(t *testing.T) {
	t.Parallel()

	// Deploy the MySQL DB
	dbOpts := createDbOpts(t, dbDirStage)
	defer terraform.Destroy(t, dbOpts)
	terraform.InitAndApply(t, dbOpts)

	// Deploy the hello-world-app
	helloOpts := createHelloOpts(dbOpts, appDirStage)
	defer terraform.Destroy(t, helloOpts)
	terraform.InitAndApply(t, helloOpts)

	// Validate the hello-world-app works
	validateHelloApp(t, helloOpts)
}
```

The test is structured as follows: deploy mysql, deploy the hello-world-app, validate the app, undeploy the hello-world-app (runs at the end due to defer), and, finally, undeploy mysql (runs at the end due to defer). The createDbOpts, createHelloOpts, and validateHelloApp methods don't exist yet, so let's implement them one at a time, starting with the createDbOpts method:

```
func createDbOpts(t *testing.T, terraformDir string)
*terraform.Options {
        uniqueId := random.UniqueId()

        return &terraform.Options{
                TerraformDir: terraformDir,

                Vars: map[string]interface{}{
                        "db_name":     fmt.Sprintf("test%s",
uniqueId),
                        "db_username": "admin",
                        "db_password": "password",
                },
        }
}
```

Not much new so far: the code points terraform.Options at the passed-in directory and sets the db_name, db_username, and db_password variables.

The next step is to deal with where this mysql module will store its state. Up to now, the backend configuration has been set to hardcoded values:

```
  backend "s3" {
    # Replace this with your bucket name!
    bucket         = "terraform-up-and-running-state"
    key            = "stage/data-stores/mysql/terraform.tfstate"
    region         = "us-east-2"

    # Replace this with your DynamoDB table name!
    dynamodb_table = "terraform-up-and-running-locks"
    encrypt        = true
  }
```

These hardcoded values are a big problem for testing, because if you don't change them, you'll end up overwriting the real state file for staging! One option is to use Terraform workspaces (as discussed in "Isolation via Workspaces"), but that would still require access to the S3 bucket in the staging account, whereas you should be running tests in a totally separate AWS account. The better option is to use partial configuration, as introduced in "Limitations with Terraform's Backends". Move the entire `backend` configuration into an external file, such as *backend.hcl*:

```
bucket         = "terraform-up-and-running-state"
key            = "stage/data-stores/mysql/terraform.tfstate"
region         = "us-east-2"
dynamodb_table = "terraform-up-and-running-locks"
encrypt        = true
```

leaving the `backend` configuration in *live/stage/data-stores/mysql/main.tf* empty:

```
    backend "s3" {
    }
```

When you're deploying the `mysql` module to the real staging environment, you tell Terraform to use the `backend` configuration in *backend.hcl* via the `-backend-config` argument:

```
$ terraform init -backend-config=backend.hcl
```

When you're running tests on the `mysql` module, you can tell Terratest to pass in test-time-friendly values using the `BackendConfig` parameter of `terraform.Options`:

```
func createDbOpts(t *testing.T, terraformDir string)
*terraform.Options {
        uniqueId := random.UniqueId()

        bucketForTesting := "YOUR_S3_BUCKET_FOR_TESTING"
        bucketRegionForTesting :=
"YOUR_S3_BUCKET_REGION_FOR_TESTING"
```

```
	dbStateKey := fmt.Sprintf("%s/%s/terraform.tfstate",
t.Name(), uniqueId)

	return &terraform.Options{
		TerraformDir: terraformDir,

		Vars: map[string]interface{}{
			"db_name":     fmt.Sprintf("test%s",
uniqueId),
			"db_username": "admin",
			"db_password": "password",
		},

		BackendConfig: map[string]interface{}{
			"bucket":  bucketForTesting,
			"region":  bucketRegionForTesting,
			"key":     dbStateKey,
			"encrypt": true,
		},
	}
}
```

You'll need to update the `bucketForTesting` and `bucketRegionForTesting` variables with your own values. You can create a single S3 bucket in your test AWS account to use as a `backend`, as the `key` configuration (the path within the bucket) includes the `uniqueId`, which should be unique enough to have a different value for each test.

The next step is to make some updates to the `hello-world-app` module in the staging environment. Open *live/stage/services/hello-world-app/variables.tf*, and expose variables for `db_remote_state_bucket`, `db_remote_state_key`, and `environment`:

```
variable "db_remote_state_bucket" {
  description = "The name of the S3 bucket for the database's
remote state"
  type        = string
}

variable "db_remote_state_key" {
  description = "The path for the database's remote state in S3"
  type        = string
}

variable "environment" {
  description = "The name of the environment we're deploying to"
  type        = string
  default     = "stage"
}
```

Pass those values through to the `hello-world-app` module in *live/stage/services/hello-world-app/main.tf*:

```
module "hello_world_app" {
  source = "../../../../modules/services/hello-world-app"

  server_text            = "Hello, World"

  environment            = var.environment
  db_remote_state_bucket = var.db_remote_state_bucket
  db_remote_state_key    = var.db_remote_state_key

  instance_type      = "t2.micro"
  min_size           = 2
  max_size           = 2
  enable_autoscaling = false
  ami                = data.aws_ami.ubuntu.id
}
```

Now you can implement the `createHelloOpts` method:

```
func createHelloOpts(
        dbOpts *terraform.Options,
        terraformDir string) *terraform.Options {

        return &terraform.Options{
                TerraformDir: terraformDir,

                Vars: map[string]interface{}{
                        "db_remote_state_bucket":
dbOpts.BackendConfig["bucket"],
                        "db_remote_state_key":
dbOpts.BackendConfig["key"],
                        "environment":
dbOpts.Vars["db_name"],
                },
        }
}
```

Note that `db_remote_state_bucket` and `db_remote_state_key` are set to the values used in the `BackendConfig` for the `mysql` module to ensure that the `hello-world-app` module is reading from the exact same state to which the `mysql` module just wrote. The `environment` variable is set to the `db_name` just so all the resources are namespaced the same way.

Finally, you can implement the `validateHelloApp` method:

```
func validateHelloApp(t *testing.T, helloOpts *terraform.Options)
{
        albDnsName := terraform.OutputRequired(t, helloOpts,
"alb_dns_name")
        url := fmt.Sprintf("http://%s", albDnsName)

        maxRetries := 10
        timeBetweenRetries := 10 * time.Second

        http_helper.HttpGetWithRetryWithCustomValidation(
                t,
                url,
                nil,
                maxRetries,
                timeBetweenRetries,
                func(status int, body string) bool {
                        return status == 200 &&
                                strings.Contains(body, "Hello,
World")
                },
        )
}
```

This method uses the `http_helper` package, just as with the unit tests, except this time, it's with the `http_helper.HttpGetWithRetryWithCustomValidation` method that allows you to specify custom validation rules for the HTTP response status code and body. This is necessary to check that the HTTP response *contains* the string "Hello, World," rather than equals that string