Testing

****

When you try to define what testing is, you will come up with a multitude of answers, and many of us will not understand the full benefits of testing until we've been burnt by buggy software or we have tried to change a complex code base which has no tests.

When I tried to define testing, I came up with the following:

*"The art of a good night's sleep is knowing you will not get woken by a support call and the piece of mind from being able to confidently change your software in an always moving market."*

OK, so I am trying to be funny, but the concept is correct. Nobody enjoys debugging poorly written code, and indeed, nobody enjoys the stress caused when a system fails. Starting out with a mantra of quality first can alleviate many of these problems.

Over the last 20 years, techniques like TDD have become commonplace. In some instances, it is not as common as I would like, but at least people are talking about testing now. In some ways, we have the Agile Alliance to thank for this:

*the principle of releasing little and often provides significant business benefits; the downside (or the benefit, depending on your viewpoint) to releasing little and often is that you can no longer spend three months running through a regression test suite before you release to market.*

In my office, context switching is one of the biggest complaints. Nobody enjoys having to drop what they are doing to investigate a problem on work that they or even a colleague may have carried out months or years ago. We want to be moving forward; and to ensure we can do that, we have to make sure that what we have previously delivered meets the specification and is of high enough quality to meet the client's requirement.

I also mentioned a change in my definition, and one of the biggest problems with change is the concern that the change you are making may have an undesirable effect on another part of the system. This effect applies to microservices as well as large monolithic systems.

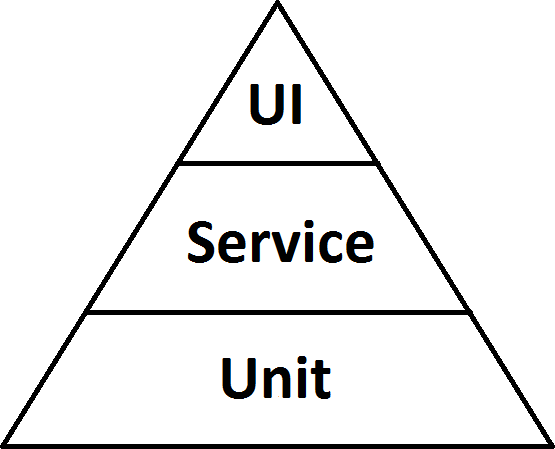
What if I also told you that the side effect of code that is easy to test is probably well-written code that is loosely coupled and has the right abstractions?

Testing, however, is not just about the developer: there is a definite need for manual testing by people detached from the code base. This exploratory testing can bring out missing requirements or incorrect assumptions. In itself, this is a specialized field and way beyond the scope of this book, so we are going to concentrate on the kind of testing that you should be doing.

# The testing pyramid

Mike Cohn is credited with having created the concept of a testing pyramid in his book *Succeeding with Agile*. The concept is that your cheapest (fastest) tests to run, which will be your unit tests, go at the bottom of the pyramid; service level integration tests are on top of this, and at the very top, you place full end-to-end tests, which are the costliest element.

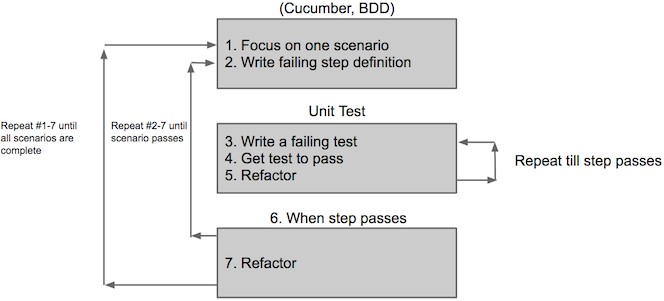
Because this is a pyramid, the number of tests gets smaller as you move up the pyramid.



In the early days of automated testing, all the testing was completed at the top of the pyramid. While this did work from a quality perspective, it meant the process of debugging the area at fault would be incredibly complicated and time-consuming. If you were lucky, there might be a complete failure which could be tracked down to a stack trace. If you were unlucky, then the problem would be behavioral; and even if you knew the system inside out, it would involve plowing through thousands of lines of code and manually repeating the action to reproduce the failure.

# Outside-in development

When writing tests, I like to follow a process called outside-in development. With outside-in development, you start by writing your tests almost at the top of the pyramid, determine what the functionality is going to be for the story you are working on, and then write some failing test for this story. Then you work on implementing the unit tests and code which starts to get the various steps in the behavioral tests to pass.

This initial specification also becomes the living documentation for your system. We will go into more detail as to how you can create this later in this chapter, but more often than not it is written in a language like **Gherkin** and is defined by working in a group with a domain specialist like a product owner, a developer, and a testing expert. The intention behind Gherkin is to create a universal language that everyone understands. This ubiquitous language uses verbs and nouns that have special meaning to the team, and which is almost always domain-specific, but should also be understandable to outsiders.

**Feature**: As a user when I call the search endpoint, I would like to receive a list of kittens

The feature is the story which, in an agile environment is owned by the product owner. The feature is then broken down into scenarios which explain in greater detail the qualities that the code must have to be acceptable.

**Scenario**: Invalid query

Given I have no search criteria When I call the search endpoint

Then I should receive a bad request message

When we get to the section on BDD in a little while we will examine this in greater depth; we will also look at a framework for Go for writing and executing Cucumber specifications. Now, however, I am going to break the rules of outside in development by showing you how to write great unit tests in Go. The concepts we are about to learn will be greatly beneficial when we do start to look at BDD, so I think it is best we cover them first. Like the previous chapters, it will be useful to you when reading through this chapter to have the source code handy; you can clone the code from the following location: [https://github.co](https://github.com/building-microservices-with-go/chapter4.git)

[m/building-microservices-with-go/chapter4.git](https://github.com/building-microservices-with-go/chapter4.git)

## Unit tests

Our unit tests go right down to the bottom of the pyramid. This book was never intended to be a lesson in TDD, there are plenty of better places to learn that. We will, however, take a look at the testing framework which is built into Go. Before we do that, let's just remind ourselves of the three laws of testing as defined by the awesome Uncle Bob Martin in his book *Clean Code*:

**First law**: You may not write production code until you have written a failing unit test

**Second law**: You may not write more of a unit test than is sufficient to fail, and not compiling is failing

**Third law**: You may not write more production code than is sufficient to pass the currently failing test

One of the most effective ways to test a microservice in Go is not to fall into the trap of trying to execute all the tests through the HTTP interface. We need to develop a pattern that avoids creating a physical web server for testing our handlers, the code to create this kind of test is slow to run and incredibly tedious to write. What need to be doing is to test our handlers and the code within them as unit tests. These tests will run far quicker than testing through the web server, and if we think about the coverage, we will be able to test the wiring of the handlers in the Cucumber tests that execute a request to the running server which overall gives us 100% coverage of our code.

main.go

1. func main() {
2. err := http.ListenAndServe(":2323", &handlers.SearchHandler{})
3. if err != nil {
4. log.Fatal(err) 14 }

15 }

You will see in the main function that we have split the handlers out into a separate package. Breaking up the code in this way allows us to test these in isolation, so let's go ahead and write some unit tests for our SearchHandler.

The convention is that we define our test files in the same folder as the package to which they belong, and we name them the same as the file they are testing followed by \_test. For our example, we are going to write some tests for our SearchHandler which lives in the handlers/search.go file; therefore our test file will be named handlers/search\_test.go.

The signature for a test method looks like this:

func TestXxx(\*testing.T)

The name of the test must have a particular name beginning with Test and then immediately following this an uppercase character or number. So we could not call our test TestmyHandler, but we could name it Test1Handler or TestMyHandler.

Again, drawing on Uncle Bob's wisdom, we need to think about the names of our tests as carefully as we would the names for our methods in our production code.

The first test we are going to write is the one who will validate that search criteria have been sent with the request and the implementation is going to look like this:

9 func TestSearchHandlerReturnsBadRequestWhenNoSearchCriteriaIsSent(t

\*testing.T) {

1. handler := SearchHandler{}
2. request := httptest.NewRequest("GET", "/search", nil)
3. response := httptest.NewRecorder() 13

14 handler.ServeHTTP(response, request) 15

1. if response.Code != http.StatusBadRequest {
2. t.Errorf("Expected BadRequest got %v", response.Code) 18 }

19 }

The net/http/httptest package has two fantastic convenience methods for us NewRequest and NewResponse, if you are familiar with unit testing concepts, then one of the fundamentals isolate dependency. Often we replace the dependencies with Mocks or Spies which allow us to test the behavior of our code without having to execute code in the dependencies. These two functions enable us to do exactly this; they generate Mock versions of the dependent objects http.Request and http.ResponseWriter.

### httptest.NewRequest

The first line we need to pay attention to is line **11**: the net/http/httptest package has some nice convenience methods for us. The NewRequest method returns an incoming server request which we can then pass to our http.Handler:

func NewRequest(method, target string, body io.Reader) \*http.Request

We can pass parameters to the method and the target, which is either the path or an absolute URL. If we only pass a path, then example.com will be used as our host setting. Finally, we can give it an io.Reader file which will correspond to the body of the request; if we do not pass a nil value then Request.ContentLength is set.

### httptest.NewRecorder

In line **12** we are creating a ResponseRecorder type: this is going to be our instance of ResponseWriter that we pass to the handler. Because a handler has no return function to validate correct operation, we need to check what has been written to the output. The ResponseRecorder type is an implementation of http.ResponseWriter which does just that: it records all the mutations we make so that it is later possible to make our assertions against it.

type ResponseRecorder struct {

Code int // the HTTP response code from WriteHeader HeaderMap http.Header // the HTTP response headers

Body \*bytes.Buffer // if non-nil, the bytes.Buffer to append

written data to

Flushed bool

// contains filtered or unexported fields

}

All we then need to do is to call the ServeHTTP method with our dummy request and response and then assert that we have the correct outcome.

Go does not have an assertion library as you would find with RSpec or JUnit. We will look at a third-party framework later in this chapter, but for now, let's concentrate on the standard packages.

In line **16**, we are checking to see if the response code returned from the handler is equal to the expected code http.BadRequest. If it is not, then we call the Errorf method on the testing framework.

#### ErrorF

func (c \*T) Errorf(format string, args ...interface{})

The Errorf function takes the parameters of a format string and a variadic list of parameters; internally this calls the Logf method before calling Fail.

If we run our tests by running the command go test -v -race ./... , we should see the following output:

**=== RUN TestSearchHandlerReturnsBadRequestWhenNoSearchCriteriaIsSent**

**--- FAIL: TestSearchHandlerReturnsBadRequestWhenNoSearchCriteriaIsSent (0.00s)**

**search\_test.go:17: Expected BadRequest got 200**

**FAIL**

**exit status 1**

**FAIL github.com/nicholasjackson/building-microservices-in- go/chapter5/handlers 0.016s**

The -v flag will print the output in a verbose style, and it will also print all the text written to the output by the application, even if the test succeeds.

The -race flag enables Go's race detector which holds discover bugs with concurrency problems. A data race occurs when two Go routines access the same variable concurrently, and at least one of the accesses is a write. The race flag adds a small overhead to your test run, so I recommend you add it to all executions.

Using -./... as our final parameter allows us to run all our tests in the current folder as well as the child folders, it saves us from manually having to construct a list of packages or files to test.

Now we have a failing test we can go ahead and write the implementation to make the test pass:

1. decoder := json.NewDecoder(r.Body)
2. defer r.Body.Close() 20
3. request := new(searchRequest)
4. err := decoder.Decode(request)
5. if err != nil {
6. http.Error(rw, "Bad Request", http.StatusBadRequest)
7. return 26 }

When we rerun the tests, we can see that they have succeeded:

**=== RUN TestSearchHandlerReturnsBadRequestWhenNoSearchCriteriaIsSent**

**--- PASS: TestSearchHandlerReturnsBadRequestWhenNoSearchCriteriaIsSent (0.00s)**

**PASS**

**ok github.com/nicholasjackson/building-microservices-in- go/chapter5/handlers 1.022s**

This output is awesome; but what if passing a query to the request with a blank string constitutes a failure? Time to write another test:

* 1. func TestSearchHandlerReturnsBadRequestWhenBlankSearchCriteriaIsSent(t

\*testing.T) {

* 1. handler := SearchHandler{}
  2. data, \_ := json.Marshal(searchRequest{})
  3. request := httptest.NewRequest("POST", "/search", bytes.NewReader(data))
  4. response := httptest.NewRecorder() 28

29 handler.ServeHTTP(response, request) 30

1. if response.Code != http.StatusBadRequest {
2. t.Errorf("Expected BadRequest got %v", response.Code) 33 }

34 }

This test is very similar to the last one; the only difference is that we are passing some JSON in the request body. While this test will fail correctly, we should take the lead from Uncle Bob and refactor this to make it more readable:

1. func TestSearchHandlerReturnsBadRequestWhenBlankSearchCriteriaIsSent(t

\*testing.T) {

1. r, rw, handler := setupTest(&searchRequest{}) 23

|  |  |  |
| --- | --- | --- |
| 24 |  | handler.ServeHTTP(rw, r) |
| 25 |  |  |
| 26 |  | if rw.Code != http.StatusBadRequest { |
| 27 |  | t.Errorf("Expected BadRequest got %v", rw.Code) |
| 28 |  | } |
| 29 | } |  |

We have refactored our test to add a setup method which is shared across the two tests, the intention behind this is to keep our tests focused on three core areas:

Setup Execute Assert

Bad tests with duplicated code can be worse than bad code: your tests should be clear, easy to understand and contain the same care that you would add to your production code.

Now, if the test fails, we can go ahead and update our code to implement the feature:

1. if err != nil || len(request.Query) < 1 {
2. http.Error(rw, "Bad Request", http.StatusBadRequest)
3. return 26 }

All we needed to do was make a simple modification to our if statement. As our system grows in complexity and we find more cases for what constitutes an invalid search query, we will refactor this into a separate method; but, for now, this is the minimum we need to do to make the test pass.

## Dependency injection and mocking

To get the tests that return items from the Search handler to pass, we are going to need a data store. Whether we implement our data store in a database or a simple in-memory store we do not want to run our tests against the actual data store as we will be checking both data store and our handler. For this reason, we are going to need to manage the dependencies on our handler so that we can replace them in our tests. To do this, we are going to use a technique called dependency injection where we will pass our dependencies into our handler rather than creating them internally.

This method allows us to replace these dependencies with stubs or mocks when we are testing the handler, making it possible to control the behavior of the dependency and check how the calling code responds to this.

Before we do anything, we need to create our dependency. In our simple example, we are going to create an in-memory data store which has a single method:

Search(string) []Kitten

To replace the type with a mock, we need to change our handler to depend on an interface which represents our data store. We can then interchange this with either an actual data store or a mock instance of the store without needing to change the underlying code:

type Store interface { Search(name string) []Kitten

}

We can now go ahead and create the implementation for this. Since this is a simple example, we are going to hardcode our list of kittens as a slice and the search method will just select from this slice when the criteria given as a parameter matches the name of the kitten.

OK, great; we now have our data store created, so let's see how we are going to modify our handler to accept this dependency. It is quite simple: because we created a struct which implements the ServeHTTP method, we can just add our dependencies onto this struct:

Search {

Store data.Store

}

Note how we are using a reference to the interface rather than the concrete type, which allows us to interchange this object with anything that implements the store interface.

Now, back to our unit tests: we would like to ensure that, when we call the ServeHTTP method with a search string, we are querying the data store and returning the kittens from it.

To do this, we are going to create a mock instance of our data store. We could create the mock ourselves; however, there is an excellent package by Matt Ryer who incidentally is also a Packt author. Testify (<https://github.com/stretchr/testify.git>) has a fully featured mocking framework with assertions. It also has an excellent package for testing the equality of objects in our tests and removes quite a lot of the boilerplate code we have to write.

In the data package, we are going to create a new file called mockstore.go. This structure will be our mock implementation of the data store:

1. // MockStore is a mock implementation of a datastore for testing purposes
2. type MockStore struct {
3. mock.Mock 8 }

9

1. //Search returns the object which was passed to the mock on setup
2. func (m \*MockStore) Search(name string) []Kitten {
3. args := m.Mock.Called(name) 13

14 return args.Get(0).([]Kitten) 15 }

In line **6**, we are defining our MockStore object. There is nothing unusual about this, except you will note that it is embedding the mock.Mock type. Embedding Mock will give us all of the methods on the mock.Mock struct.

When we write the implementation for our search method, we are first calling the Called method and passing it the arguments that are sent to Search. Internally, the mock package is recording that this method was called and with what parameters so that we can later write an assertion against it:

args := m.Mock.Called(name)

Finally, we are returning args.Get(0).(Kitten). When we call the Called method, the mock returns us a list of arguments that we provided in the setup. We are casting this to our output type and returning to the caller. Let's take a quick look at the test method and see how this works.

Line **57** is the start of our test setup. The first thing we are going to do is to create an instance of our MockStore. We then set this as a dependency for our Search handler. If we skip back up the file to line **38**, you will see that we are calling the On method on our mockStore. The On method is a setup method for the mock and has the signature:

func (c \*Call) On(methodName string, arguments ...interface{}) \*Call

If we do not call the On method with the parameter Search then when we call the Search method in our code we will get an exception from the test saying that Search has been called yet has not been setup. One of the reasons why I like to use mocking rather than a simple Stub is this ability to assert that a method has been called and we can explicitly dictate the behavior that the code under test is allowed to exhibit. This way we can ensure that we are not doing work the output of which has not been tested.

In our instance, we are setting up the condition that, when the Search method is called with the parameter Fat Freddy's Cat, we would like to return an array of kittens.

The assertion is that we are calling the Search method on the data store and passing it the query that was sent in the HTTP response. Using assertions in this way is a handy technique as it allows us to test the unhappy path such as when a data store may not be able to return data due to an internal error or another reason. If we were trying to test this with an integration test, it could be tough to persuade the database to fail on demand.

Why don't you spend five minutes as a little exercise and go ahead and write this code to finish off?

Did it all work? Don't worry if not, you can just check out the example code to see where you have gone wrong, but I hope that the process is useful. You can see how you can take a measured approach through two layers of testing to produce a working application. These tests are now your safety net: whenever you change the code to add a new feature, you can be sure that you are not breaking something unintentionally.

## Code coverage

Code coverage is an excellent metric to ensure that the code you are writing has adequate coverage.

The most simplistic way of getting a readout of test coverage is to execute our tests with the

-cover option:

**go test -cover ./...**

If we run this against our example code in the root folder of our example code we will see the following output:

**$go test -cover ./...**

**? github.com/building-microservices-with-go/chapter4 [no test files]**

**ok github.com/building-microservices-with-go/chapter4/data 0.017s coverage: 20.0% of statements**

**ok github.com/building-microservices-with-go/chapter4/features 0.018s coverage: 0.0% of statements [no tests to run]**

**ok github.com/building-microservices-with-go/chapter4/handlers 0.018s coverage: 100.0% of statements**

Now our handlers look beautiful: we have 100% coverage of this package. However, our data package is only reporting 20% coverage. Before we get too alarmed at this, let's take a look at what we are trying to test.

If we first examine the datastore.go file, this is only an interface and therefore does not have any test files; however, the memorystore.go does. This file is well covered with 100% test coverage for this file. The files that are letting us down are our mock class and our MongoDB implementation.

Now the mock type I am not too bothered about, but the Mongo store is an interesting problem.

This type would be incredibly difficult to test due to the dependency on the MongoDB. We could create a mock implementation of the Mongo package to test our code, but this could be more complicated than the implementation. There are however some critical areas where we could make mistakes in this class. Consider line **26**:

c := s.DB("kittenserver").C("kittens")

This line retrieves the collection kittens from the database kittenserver. If we make a simple spelling mistake here, then our application will not work. We do not want to wait until this code gets out into production to see that this is happening. We also do not want to have to manually test this as, in a larger application, this could be considerably time- consuming. Integration tests are really where our Cucumber tests shine. If you remember, we are writing some very high-level end-to-end tests to make sure that the input into our API results in the correct output. Because this is running against an actual database, if we had made such an error, then it would be picked up. So, while the Go coverage report states that we are not covered, it's because we have higher level tests that the Go test is not looking at, so we are covered. The central area where we could run into problems would be by omitting line **23**.

If we do not close the connection to the database after we have opened it, we are going to be leaking connections; after a while, we may find that we can no longer open another as the pool is exhausted. There is no simple way to test this, but there is, however, a way to catch the problem post-deploy. When we look at logging and monitoring in Chapter 7, *Logging and Monitoring*, we will see how we can expose such information to help us ensure our production system is functioning correctly.

## Behavioral Driven Development

**Behavioral Driven Development** (**BDD**) and is a technique often executed by an application framework called **Cucumber**. It was developed by Dan North and was designed to create a common ground between developers and product owners. In theory, it should be possible to test complete coverage of the system with BDD; however, since this would create a significant number of slow running tests it is not the best approach. What we should be doing is defining the boundaries of our system, and we can save the granularity for our unit tests.

In our Three Amigos group, we discuss the facets of the feature and what the essential qualities of it are and start to write scenarios.

#### Sad path

**Scenario**: User passes no search criteria Given I have no search criteria

When I call the search endpoint

Then I should receive a bad request message

#### Happy path

**Scenario**: User passes valid search criteria Given I have valid search criteria

When I call the search endpoint

Then I should receive a list of kittens

These scenarios are quite a simple example, but I think you can understand how, when using this language with non-technical people, it would be quite straightforward to come up with these descriptions. From an automation perspective what we then do is to write the steps which correspond to each of these Given, When, and Then statements.

For this book, we are going to take a look at the GoDog framework which allows us to implement the step definitions in Go. We will first need to install the application you can do this by running the command: fgo get github.com/DATA-DOG/godog/cmd/godog

If we look at features/search.feature we can see that we have implemented this feature and the scenarios.

If we run the godog ./ command to run these tests without first creating the features, we should see the following error message:

**Feature: As a user when I call the search endpoint, I would like to receive a list of kittens**

**Scenario: Invalid query**

**# features/search.feature:4**

**Given I have no search criteria When I call the search endpoint**

**Then I should receive a bad request message**

**Scenario: Valid query**

**# features/search.feature:9**

**Given I have valid search criteria When I call the search endpoint**

**Then I should receive a list of kittens**

**2 scenarios (2 undefined)**

**6 steps (6 undefined) 321.121µs**

You can implement step definitions for undefined steps with these snippets:

func iHaveNoSearchCriteria() error { return godog.ErrPending

}

func iCallTheSearchEndpoint() error { return godog.ErrPending

}

func iShouldReceiveABadRequestMessage() error { return godog.ErrPending

}

func iHaveAValidSearchCriteria() error { return godog.ErrPending

}

func iShouldReceiveAListOfKittens() error { return godog.ErrPending

}

func FeatureContext(s \*godog.Suite) {

s.Step(`^I have no search criteria$`, iHaveNoSearchCriteria) s.Step(`^I call the search endpoint$`, iCallTheSearchEndpoint) s.Step(`^I should receive a bad request message$`,

iShouldReceiveABadRequestMessage)

s.Step(`^I have a valid search criteria$`, iHaveAValidSearchCriteria) s.Step(`^I should receive a list of kittens$`,

iShouldReceiveAListOfKittens)

}

Usefully, this gives us the boilerplate to perform our steps; once we implement this and rerun the command we get a different message:

**Feature: As a user when I call the search endpoint, I would like to receive a list of kittens**

**Scenario: Invalid query # search.feature:4 Given I have no search criteria # search\_test.go:6 ->**

**github.com/nicholasjackson/building-microservices-in- go/chapter5/features.iHaveNoSearchCriteria**

**TODO: write pending definition When I call the search endpoint**

**Then I should receive a bad request message**

**Scenario: Valid query # search.feature:9 Given I have a valid search criteria # search\_test.go:18 ->**

**github.com/nicholasjackson/building-microservices-in- go/chapter5/features.iHaveAValidSearchCriteria**

**TODO: write pending definition When I call the search endpoint**

**Then I should receive a list of kittens**

**2 scenarios (2 pending)**

**6 steps (2 pending, 4 skipped) 548.978µs**

We can now start filling in the details for the steps which should fail as we have not yet written our code.

We can implement our code using plain Go which gives us the capability to use any of the interfaces and packages. Take a look at the example which corresponds to the method iCallTheSearchEndpoint:

1. func iCallTheSearchEndpoint() error {
2. var request []byte 25
3. response, err = http.Post("http://localhost:2323", "application/json", bytes.NewReader(request))
4. return err 28 }

29

1. func iShouldReceiveABadRequestMessage() error {
2. if response.StatusCode != http.StatusBadRequest {
3. return fmt.Errorf("Should have recieved a bad response") 33 }

34

35 return nil 36 }

Now we have some tests implemented we should run Cucumber tests as some of the steps should be passing. To test the system, we need to start our main application; we could split our main function out into a StartServer function, which could be called directly from Cucumber. However, that is omitting the fact that we forgot to call StartServer in the main function. For this reason, the best approach is to test the complete application in our Cucumber test from the outside in.

To do this, we are going to add a couple of new functions to the

features/search\_test.go file:

1. func FeatureContext(s \*godog.Suite) {
2. s.Step(`^I have no search criteria$`, iHaveNoSearchCriteria)
3. s.Step(`^I call the search endpoint$`, iCallTheSearchEndpoint)
4. s.Step(`^I should receive a bad request message$`, iShouldReceiveABadRequestMessage)
5. s.Step(`^I have a valid search criteria$`, iHaveAValidSearchCriteria)
6. s.Step(`^I should receive a list of kittens$`, iShouldReceiveAListOfKittens)

65

1. s.BeforeScenario(func(interface{}) {
2. startServer()
3. fmt.Printf("Server running with pid: %v", server.Process.Pid) 69 })

70

1. s.AfterScenario(func(interface{}, error) {
2. server.Process.Kill() 73 })

74 }

75

76 var server \*exec.Cmd 77

1. func startServer() {
2. server = exec.Command("go", "run", "../main.go")
3. go server.Run()
4. time.Sleep(3 \* time.Second) 82 }

In line **66**, we are using the BeforeScenario method on godog: this allows us to run a function before our scenario starts. We would use this for clearing up any data in the data store, but in our simple example, we are just going to start our application server. Later on in this chapter, we will look at a more complex example, which uses Docker Compose to start a stack of containers containing our server and a database.

The startServer function spawns a new process to run go run ../main.go. We have to run this in gofunc as we do not want the test to block. Line **81** contains a small pause to see if our server has started. In reality, we should be checking the health endpoint of the API, but for now, this will suffice.

Line **71** will execute after the scenario has finished and tears down our server. If we don't do this then the next time we try to start our server it will fail as the process will already be running and bound to the port.

Let's go ahead and run our Cucumber tests, and the output should look something like this:

**Feature: As a user when I call the search endpoint, I would like to receive a list of kittens**

**Server running with pid: 91535**

**Scenario: Invalid query # search.feature:4 Given I have no search criteria # search\_test.go:17 ->**

**github.com/building-microservices-with- go/chapter4/features.iHaveNoSearchCriteria**

**When I call the search endpoint # search\_test.go:25 -> github.com/building-microservices-with- go/chapter4/features.iCallTheSearchEndpoint**

**Then I should receive a bad request message # search\_test.go:32 -> github.com/building-microservices-with- go/chapter4/features.iShouldReceiveABadRequestMessage**

**Server running with pid: 91615**

**Scenario: Valid query # search.feature:9 Given I have a valid search criteria # search\_test.go:40 ->**

**github.com/building-microservices-with- go/chapter4/features.iHaveAValidSearchCriteria**

**Do not have a valid criteria When I call the search endpoint**

**Then I should receive a list of kittens**

**--- Failed scenarios: search.feature:10**

**2 scenarios (1 passed, 1 failed)**

**6 steps (3 passed, 1 failed, 2 skipped) 6.010954682s**

**make: \*\*\* [cucumber] Error 1**

Perfect! We are getting there, some of the steps are passing now, and one of the features is passing. We can now go ahead finish these tests off, but first, we need to look at how we can use Docker Compose to test against a real database.

## Testing with Docker Compose

So far this has been relatively simple implementation, but it is not particularly useful as a real-world example. It is going to be pretty rare that you find yourself implementing an in- memory data store with only three items in it. More often than not you are going to be using a functioning database. Of course, the integration between a real database and our code needs testing; we need to ensure that the connection to the data store is correct and that the query we are sending to it is valid.

To do this, we need to spin up a real database and to do that we can use Docker-Compose as it is a fantastic way of starting our dependencies.

In our sample file docker-compose.yml, we have the following:

version: '2' services:

mongodb: image: mongo ports:

- 27017:27017

When we run docker-compose up command, we will download the image of MongoDB and start an instance exposing these ports on our local host.

We now need to create a new struct in our project which is going to implement the store interface. We can then execute commands against the real database as opposed to using a mock or a simple in-memory store.

The implementation for MongoStore is quite straight forward. Looking at the file in data/monogstore.go, you will see that we have two additional methods not defined in our interface, namely:

DelleteAllKittens InsertKittens

These are here because we need them for the setup of our functional tests.

If we look at our file features/search\_test.go you will see that we have added a couple of extra calls to the FeatureContext method in our setup.

The first thing we are doing is to call the waitForDB method: because we cannot control when our Mongo instance is going to be ready to accept connections, we need to wait for it before kicking off the tests. The process is that we will try to create an instance to our MongoStore using the convenience method NewMongoStore, internally this is doing the following work:

1. // NewMongoStore creates an instance of MongoStore with the given connection string
2. func NewMongoStore(connection string) (\*MongoStore, error) {

|  |  |  |
| --- | --- | --- |
| 12 |  | session, err := mgo.Dial(connection) |
| 13 |  | if err != nil { |
| 14 |  | return nil, err |
| 15 |  | } |
| 16 |  |  |
| 17 |  | return &MongoStore{session: session}, nil |
| 18 | } |  |

The Dial method attempts to connect to the instance of MongoDB specified in the connection string. If this connection fails, then an error is returned. In our code, if we receive an error, we return this to the caller of NewMongoStore with a nil instance of our struct. The waitForDB method works by repeatedly attempting to create this connection until it no longer receives an error. To avoid spamming the database while it is trying to start, we sleep for one second after every failed attempt, to a maximum time of 10 seconds. This method will block the main Go routine, but this is by design as we do not want the tests to execute until we are sure we have this connection:

* 1. func waitForDB() {
  2. var err error 100

101 for i := 0; i < 10; i++ {

1. store, err = data.NewMongoStore("localhost")
2. if err == nil {
3. break

105 }

106 time.Sleep(1 \* time.Second)

107 }

108 }

We have also added some code to the BeforeScenario setup: the first thing we are going to do is wipe our database clearing down any previous test data. Clearing data is an incredibly important step as, should we have had any methods which mutate the data; we would not get predictable test results after each run.

Finally, we insert the setupData method into our test data, and then we will proceed to execute the tests.

We now have quite a few things to do before we can test our code, we need to run docker- compose, perform our tests and then stop docker-compose. An efficient way of scripting this process is to use Makefiles, Makefiles have been around forever and are still the primary software build mechanisms for many applications. They allow us to define commands in a simple text file which we can then run by executing **make [command]**.

If we look at the Cucumber command in the Makefile in the example repository, we can see how we script the steps that need to be carried out to run our tests. We are starting our Mongo instance with docker-compose, running our Cucumber tests and then tearing the database down again:

cucumber:

docker-compose up -d

cd features && godog ./ docker-compose stop

If you are wondering why we still need the waitForDB method when we are starting our database before running the tests, remember that Docker only knows when the primary process is executed. There can be a considerable lag between the commencement of the process and it being ready to accept connections. To run this we run make cucumber from the command line the result should be passing cucumber tests:

$make cucumber docker-compose up -d

chapter4\_mongodb\_1 is up-to-date cd features && godog ./

**Feature**: As a user when I call the search endpoint, I would like to receive a list of kittens

Server running with pid: 88200

**Scenario**: Invalid query # search.feature:4

Given I have no search criteria # search\_test.go:21 -> github.com/building-microservices-with- go/chapter4/features.iHaveNoSearchCriteria

When I call the search endpoint # search\_test.go:29 -> github.com/building-microservices-with- go/chapter4/features.iCallTheSearchEndpoint

Then I should receive a bad request message # search\_test.go:40 -> github.com/building-microservices-with- go/chapter4/features.iShouldReceiveABadRequestMessage

Server running with pid: 88468

**Scenario**: Valid query # search.feature:9

Given I have a valid search criteria # search\_test.go:48 -> github.com/building-microservices-with-

go/chapter4/features.iHaveAValidSearchCriteria

When I call the search endpoint # search\_test.go:29 -> github.com/building-microservices-with- go/chapter4/features.iCallTheSearchEndpoint

Then I should receive a list of kittens # search\_test.go:54 -> github.com/building-microservices-with- go/chapter4/features.iShouldReceiveAListOfKittens

2 scenarios (2 passed)

6 steps (6 passed) 7.028664s

docker-compose stop

Stopping chapter4\_mongodb\_1 ... done

That is all for this section, we have learned that, with a few well-placed patterns, it is easy to write a robust test suite that will keep us safe and sound asleep instead of getting up in the middle of the night to diagnose a broken system. In the next section, we are going to look at some of the fantastic features of Go to ensure that our code is fast and optimized.

# Benchmarking and profiling

Go has two excellent ways to analyze the performance of your code. We have benchmark tests and the fantastic pprof.

## Benchmarks

Benchmarking is a way of measuring the performance of your code by executing it multiple times with a fixed workload. We took a look at this briefly in Chapter 1, *Introduction to Microservices*, where we ascertained that the json.Marshal method was slower than the json.Encode method. While this is a useful feature, I find it tough to work out what I should benchmark. If I am writing an algorithm, then this is relatively straightforward.

However, when writing a microservice that is predominately interacting with a database, it is far more challenging.

To demonstrate how easy it is to execute benchmarks in Go, take a look at

chandlers/search\_bench\_test.go:

1. func BenchmarkSearchHandler(b \*testing.B) {
2. mockStore = &data.MockStore{}
3. mockStore.On("Search", "Fat Freddy's Cat").Return([]data.Kitten{
4. data.Kitten{
5. Name: "Fat Freddy's Cat",

|  |  |  |
| --- | --- | --- |
| 16 |  | }, |
| 17 |  | }) |
| 18 |  |  |
| 19 |  | search := Search{DataStore: mockStore} |
| 20 |  |  |
| 21 |  | for i := 0; i < b.N; i++ { |
| 22 |  | r := httptest.NewRequest("POST", "/search", |
|  |  | bytes.NewReader([]byte(`{"query":"Fat Freddy's Cat"}`))) |
| 23 |  | rr := httptest.NewRecorder() |
| 24 |  | search.ServeHTTP(rr, r) |
| 25 |  | } |
| 26 | } |  |

The most important part of this code is hidden away at line **21**:

for n := 0; n < b.N; n++

When running a benchmark, Go needs to run it multiple times to get an accurate reason. The number of times that the benchmark will run is the field N on the benchmark's struct. Before setting this number, Go will execute a few iterations of your code to get an approximate measurement of the execution time.

We would execute our benchmark using the go test-bench -benchmem command:

**go test -bench=. -benchmem**

**BenchmarkSearchHandler-8 50000 43183 ns/op 49142**

**B/op 68 allocs/op PASS**

**ok github.com/building-microservices-with-go/chapter4/handlers 2.495s**

Here we are passing an additional flag to see the memory allocations for each execution. We know that our handler when running with the mock takes 43,183 nanoseconds or 0.043183 milliseconds to execute and performs 68 memory allocations. It'd be good if the code would run this fast when running in real life, but we might have to wait a few years before we see this level of speed from an API connected to a database.

One of the other nice features of benchmark tests is that we can run them and it outputs profiles which can be used with pprof:

**go test -bench=. -cpuprofile=cpu.prof -blockprofile=block.prof - memprofile=mem.prof**

The output of this command will give us more information about where this time and memory is being consumed and can help us to optimize our code correctly.

## Profiling

When we wish to take a look at the speed of our program, the best technique we can employ is profiling. Profiling automatically samples your running application while it is executing; and then we can compute that data, such as the running time of a particular function, into a statistical summary called a profile.

Go supports three different types of profiling:

**CPU**: Identifies the tasks which require the most CPU time

**Heap**: Identifies the statements responsible for allocating the most memory **Blocking**: Identifies the operations responsible for blocking Go routines for the longest time

If we would like to enable profiling on our application, we can do one of two things: Add import "net/http/pprof" to your startup file

Manually start profiling

The first option is the most straightforward. You only add it to the beginning of your main Go file and, if you are not already running an HTTP web server, start one:

import \_ "net/http/pprof"

go func() {

log.Println(http.ListenAndServe("localhost:6060", nil))

}()

This method then exposes various paths on your HTTP server at /debug/pprof/ which can then be accessed via a URL. The side effect of this, however, is that when this import statement is in your go file, then you will be profiling, which not only could slow down your application, but you also don't want to expose this information for public consumption.

Another method of profiling is to start the profiler when you start your application by passing some additional command line flags:

1. var cpuprofile = flag.String("cpuprofile", "", "write cpu profile to file")
2. var memprofile = flag.String("memprofile", "", "write memory profile to file")
3. var store \*data.MongoStore 22
4. func main() {
5. flag.Parse()

25

1. if \*cpuprofile != "" {
2. fmt.Println("Running with CPU profile")
3. f, err := os.Create(\*cpuprofile)
4. if err != nil {
5. log.Fatal(err) 31 }

32 pprof.StartCPUProfile(f) 33 }

34

1. sigs := make(chan os.Signal, 1)
2. signal.Notify(sigs, syscall.SIGINT, syscall.SIGTERM) 37
3. go func() {
4. <-sigs
5. fmt.Println("Finished")
6. if \*memprofile != "" {
7. f, err := os.Create(\*memprofile)
8. if err != nil {
9. log.Fatal(err) 45 }
10. runtime.GC()
11. pprof.Lookup("heap").WriteTo(f, 0)
12. defer f.Close() 49 }
13. if \*cpuprofile != "" {
14. pprof.StopCPUProfile() 52 }

53

54 os.Exit(0) 55 }()

On line **26**, we are checking whether we have specified an output file for CPU profiling, and if so, we are creating the file and then starting the profiler with pprof.StartCPUProfile(f), and passing it a reference to the file:

func StartCPUProfile(w io.Writer) error

The StartCPUProfile function enables the CPU profiling for the current process and buffers the output to w. While running, the CPU profiler will stop your application roughly 100 times per second and record the data.

To profile heap allocations, we use a slightly different command:

pprof.Lookup("heap").WriteTo(f, 0) func Lookup(name string) \*Profile

The Lookup() function returns the profile with the given name, or if no such profile exists, the predefined profiles available are:

goroutine - stack traces of all current goroutines heap - a sampling of all heap allocations

threadcreate - stack traces that led to the creation of new OS threads block - stack traces that led to blocking on synchronization primitives

fuc (p \*Profile) WriteTo(w io.Writer, debug int) error

WriteTo outputs the profile to the given write in the pprof format. If we set the debug flag to 1, then WriteTo will add comments to function names and line numbers instead of just hexadecimal addresses that pprof uses. These comments are so you can read the file without needing any special tooling.

If you look at the example code in the folder benchmark, you will find an example profile and the binary from which it was generated.

We can now run the pprof tool to examine what is going on. To do this, we need to run the tool on the command line and provide a reference to the binary that the profile was executed against, and also the profile:

**go tool pprof ./kittenserver ./cpu.prof**

The simplest command we can run is top. Top will show us the functions which consumed the most CPU during the execution of our application:

Entering interactive mode (type "help" for commands) (pprof) top

24460ms of 42630ms total (57.38%) Dropped 456 nodes (cum <= 213.15ms)

Showing top 10 nodes out of 163 (cum >= 790ms)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| flat | flat% | sum% | cum | cum% |  |
| 16110ms | 37.79% | 37.79% | 16790ms | 39.39% | syscall.Syscall |
| 2670ms | 6.26% | 44.05% | 2670ms | 6.26% | runtime.\_ExternalCode |
| 1440ms | 3.38% | 47.43% | 1560ms | 3.66% | syscall.Syscall6 |
| 900ms | 2.11% | 49.54% | 900ms | 2.11% | runtime.epollctl |
| 830ms | 1.95% | 51.49% | 2370ms | 5.56% | runtime.mallocgc |
| 610ms | 1.43% | 52.92% | 1470ms | 3.45% | runtime.pcvalue |
| 510ms | 1.20% | 54.12% | 510ms | 1.20% | runtime.heapBitsSetType |
| 470ms | 1.10% | 55.22% | 2810ms | 6.59% | runtime.gentraceback |
| 470ms | 1.10% | 56.32% | 470ms | 1.10% | runtime.memmove |
| 450ms | 1.06% | 57.38% | 790ms | 1.85% | runtime.deferreturn |

(pprof)

The main offender in this instance is syscall.Syscall. If we look this up in the documentation, we find that package syscall contains an interface to the low-level operating system primitives.

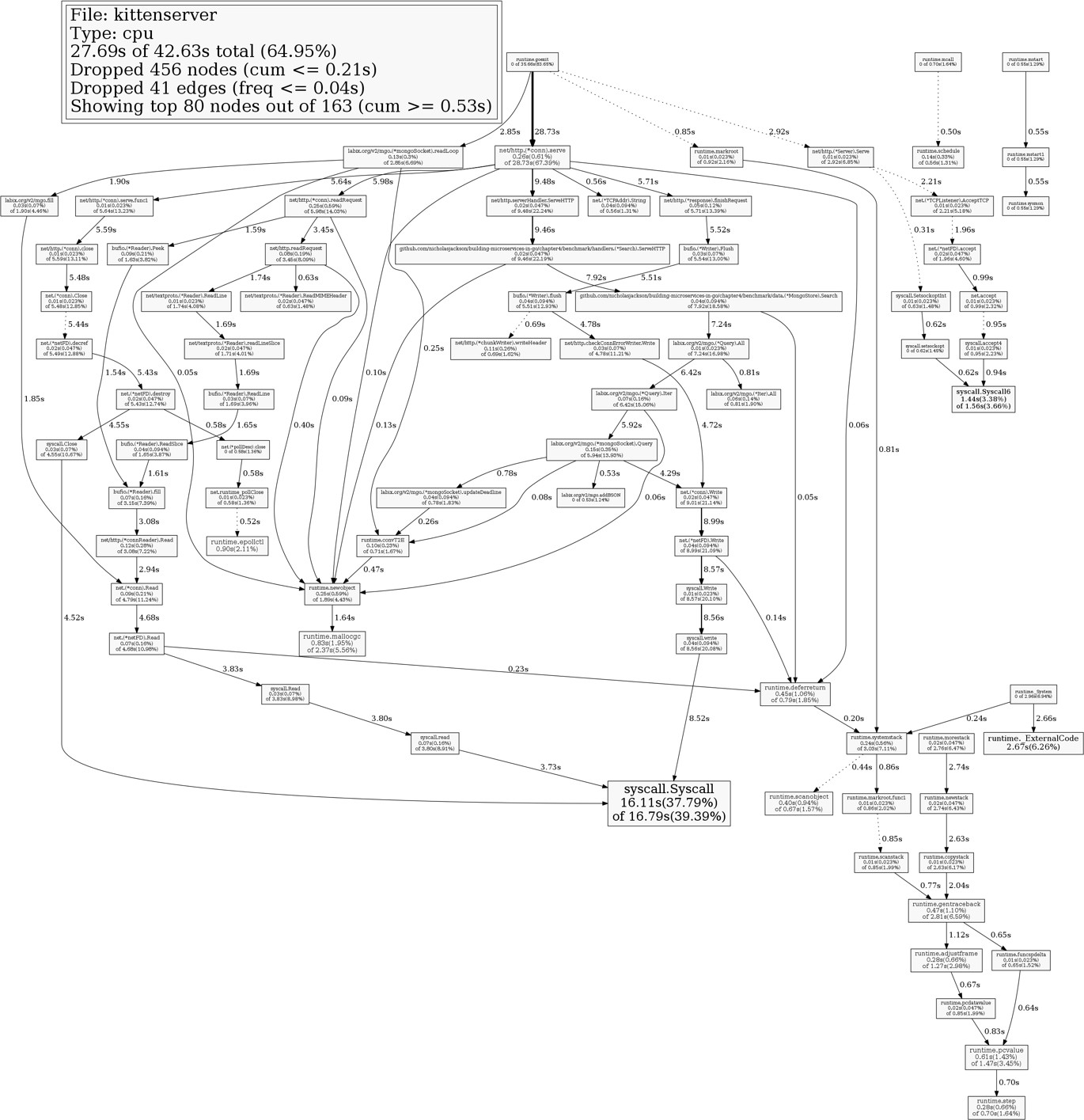
On its own this output is not particularly useful, so let's generate a call graph which will show us more detail. We can do this again from the pprof tool. However, we do need to install Graphviz first. If you are using macOS then you can install this with brew:

**brew install graphviz**

If you are using a Linux based system and have the apt package manager, you can use that:

**apt-get install graphviz**

The output for this looks like this benchmark/cpu.png:



That is quite something! However, we can see that syscall.Syscall is in the largest font as it is responsible for consuming the most CPU. If we start here at the bottom and start tracing backward, we can see that the root of this seems to be our data store's search function.

To get a closer look at exactly where this is happening, we can use the list command and then pass the name of an object or method we would like to investigate further:

**Entering interactive mode (type "help" for commands) (pprof) list Search**

**Total: 42.63s**

**ROUTINE ======================== github.com//building-microservices-with- go/chapter4/data.(\*MongoStore).Search in github.com/building-microservices- with-go/chapter4/data/mongostore.go**

**40ms 7.92s (flat, cum) 18.58% of Total**

**. . 16:**

**. . 17: return &MongoStore{session: session}, nil**

**. . 18:}**

**. . 19:**

**. . 20:// Search returns Kittens from the MongoDB instance which have the name name**

**40ms 40ms 21:func (m \*MongoStore) Search(name string)**

**[]Kitten {**

**. 270ms 22: s := m.session.Clone()**

**. 10ms 23: defer s.Close()**

**. . 24:**

**. 20ms 25: var results []Kitten**

**. 70ms 26: c := s.DB("kittenserver").C("kittens")**

**. 7.30s 27: err := c.Find(Kitten{Name:**

**name}).All(&results)**

|  |  |  |
| --- | --- | --- |
| **. .**  **. .** | **28:**  **29:** | **if err != nil {**  **return nil** |
| **. .** | **30:** | **}** |
| **. .** | **31:** |  |
| **. 210ms** | **32:** | **return results** |
| **. .** | **33:}** |  |
| **. .** | **34:** |  |

**. . 35:// DeleteAllKittens deletes all the kittens from the datastore**

**. . 36:func (m \*MongoStore) DeleteAllKittens() {**

**. . 37: s := m.session.Clone()**

When we do this with the search method, we can see that a huge percentage of our CPU cycles were spent executing queries to MongoDB. If we look at the other route from syscall.Syscall, it shows that another large consumer is the http.ResponseWriter interface.

This output all makes sense as we are not doing anything too clever in our API; it just retrieves some data from a database. The nice thing about pprof is that we can use the same commands to query the heap usage.

# Summary

In this chapter, you have learned some best practice approaches to testing microservices in Go. We have looked at the testing package, including some special features for dealing with requests and responses. We have also looked at writing integration tests with Cucumber.

Ensuring that your code works without fault, however, is only part of the job, we also need to make sure that our code is performant, and Go has some excellent tools for managing this too.

I would always recommend that you test your code and that you do this religiously. As for performance optimization, this is open for debate, no doubt you have heard comments that premature optimization is the root of all evil. However, this quote from Donald Knuth is much-misunderstood: he did not mean that you should never optimize until you have a problem; he said that you should only optimize what matters. With pprof, we have an easy way to figure out what, if anything, actually matters. Include the practice of profiling into your development routine, and you will have faster and more efficient applications, profiling is also an excellent technique to understand your application better when you are trying to track down that tricky bug.

*"Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%."*

*- Donald Knuth*