# Go Testing Explained In Color

REVISION 1

Hawthorne-Press.com

Go Testing Explained in Color

Published by

Hawthorne-Press.com

10310 Moorberry Lane
Houston, Texas 77043, USA

# TABLE OF CONTENTS

HAWTHORNE-PRESS.COM ..... 1

INTRODUCTION ..... 1

SOURCE CODE COVERAGE ..... 1

STRUCTURE OF CODE TO BE TESTED ..... 2

TEST HANDLER STRUCTURE ..... 3

TEST SEQUENCE OPERATIONS ..... 4

A FINAL WORD ..... 6

# GO TESTING EXPLAINED IN COLOR

## INTRODUCTION

The code that we will be testing is the from an employment code test. The criteria were stated as follows:

> *"**Design & code a very simple in-memory file system and expose the file system operations over HTTP"***

I was given an afternoon to code against this requirement. As can be seen, this is a wide-open description. I, perhaps foolishly, took it a little farther than would be expected. I declined to use Go Collections and other packages to ease the project. I only used two major packages, HTTP and JSON. The other packages were standard, such as FMT, IOUTIL, OS, TIME and STRINGS.

I made some minor corrections to the code before attempting to test it. This example of testing is NOT the standard TDD methodology. That is, you write a test, then write the code to pass the test, and refactor as necessary.

The testing we describe here is contemplated when a working project, without original tests, is going to be rewritten or undergo extensive modifications. By creating a test suite for a working project, the new project code can be tested for compliance as modifications are made. It is also possible for these tests to uncover deficiencies in the original project code.

The code examples, this and other White Papers are available at: www.hawthorne-press.com

The source code to be tested is named: ***goEmploy.go.*** This is an updated version of the code created in the White Paper "*Go Employment Test Explained in Color* and available under the filename ***goTest.go***.

## SOURCE CODE COVERAGE

The first thing a tester must to determine is *what to test*. For efficiency, we should test the smallest number of functions that will give you the greatest *coverage*. While the goal is to get to 100% code coverage, the larger the program, the less likely this can be achieved. For example, the ***goEmploy_test.go*** test program, only achieves 92.3% coverage. However, as we will discover, this is actually about as close as we can get without modifying the program under test. Only two simple error paths and the main function were not tested. The two error paths were tested manually by temporally modifying the original code outside the error path to test that this code worked.

Included in the code examples is the output of the Go coverage tool. It displays tested source code in green and the code that was not tested in red. This type of browser display is created as follows:

```
go test -coverprofile=coverage.out

go tool cover -html=coverage.out
```

Examining the coverage display provides a guide to what code still needs to be tested. The following abbreviated example of the ***goEmploy.go*** coverage shows all the untested code in red. Notice that only code that specifically generates code is in either green or red. Everything else is colored grey.

```
Package main

                •   •   •
                data, err := json.Marshal(xMem) // Marshal xMem ==> Data set
                check("Marshalling Failed", err)
                writeData(data)                          // Write Data Set to disk
                p, _, ok = findName(xMem, string(name)) // Find newly created name!
                if !ok {
                        fmt.Println("Update Failure") // Notify of failure
                        return
                }
                 •   •   •
//
// Error Check
//
func check(where string, e error) {
        if e != nil { // If error -
                fmt.Println("Where: ", where) // Where error occured string with return error value!
                panic(e)                      // Program "Panic"
        }
}
//
//  Main Function
//
func main() {
        loadDatabase() // Load Database

        http.HandleFunc("/view/", viewHandler) // Web Handler Functions
        http.HandleFunc("/edit/", editHandler)
        http.HandleFunc("/save/", saveHandler)
        http.HandleFunc("/delete/", deleteHandler)
        http.ListenAndServe(":8080", nil) // Start HTTP Server
}
```

## Structure of Code to be Tested

The first thing to consider in a test plan is what functions are present and which are subordinate to other functions. Subordinate functions should be tested last and only if they are not completely tested in the course of other tests. This consideration may also come up in a standard TDD setting. When the main functions in a program are tested, it may become apparent that some subordinate function tests are redundant.

The ***goEmploy.go*** program consists of a main routine and ten functions:

1. viewHandler – HTTP function handler for "/view/".
2. editHandler – HTTP function handler for "/edit/".
3. deleteHandler – HTTP function handler for "/delete/".
4. saveHandler – HTTP function handler for "/save/".
5. save – Updates in-memory database and writes result to the external database file.
6. writeData – Writes "marshalled" data to external database file.
7. LoadDatabase – Loads external database file into the in-memory database.
8. findName – Find entries in the in-memory database "containing" key text.
9. findExactName – Find entries in the in-memory database "exactly" matching key text.
10. check – If error code not NIL, display error message and calls the panic function.

The main function calls the loadDatabase function, setups the four HTTP handler functions and calls the HTTP ListenAndServe on port 8081, which starts the HTTP Server.

Given this information, the proper course is to write tests for the handler functions and the LoadDatabase function first. As we will see, after creating these tests, all the code is tested except for the main function and two minor error paths. (See above),

## TEST HANDLER STRUCTURE

Three main features of this program are critical to how it is tested.

1. The code maintains an in-memory database.
2. The external file is a JSON based file.
3. The two databases are synchronized after each HTTP Operation unless there is a file handling error.

The general plan for each handler is to use an anonymous structure slice to contain all the test parameters for each test sequence for a particular handler. The structure definition is as follows:

```
    cases := []struct {
        w                    *httptest.ResponseRecorder
        r                    *http.Request
        expectedResponseCode int
        expectedResponseBody []byte
        initial_DB           []byte
        returnedDB           []byte
    }{
      ….
     },
    }
```

Each test sequence has the following elements.

The **httptest.NewRecorder**() utility, provided by the *httptest Package,* returns a pointer to the following structure that records the results of a HTTP Requests.

```
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
}
```

A **http.Request** is created by calling **http.NewRequest** function. This function returns a pointer to a request structure given a *method, URL*, and *optional body*.

The **expectedResponseCode** and **expectedResponseBody** are the expected results returned in ResponseRecorder structure in the elements *Code* and *Body.*

The **initial_DB** is the initial in-memory database image.

The **returnedDB** is the expected in-memory database image after the request is processed.

The following is a typical test sequence:

```
    viewRequest, err := http.NewRequest("GET", "/view/", nil)
    if err != nil {
        t.Fatal("View NewRequest error: ", err)
    }

. . .
        {
            w:                   httptest.NewRecorder(),
            r:                   viewRequest,
            expectedResponseCode: http.StatusOK,
            expectedResponseBody: []byte("<h1>View: Empty Database</h1>"),
            initial_DB:          []byte(null_db),
            returnedDB:          []byte(null_db),  // View tests do use this element
        },
```

The initial and returned database values are constants containing "marshalled" images of the external file database. These images are "unmarshalled" into the main in-memory database or into an auxiliary in-memory database for comparison to the expected result. These constants are called a "Data Set". See Below:

```
const c_db = "[{\"Name\":\"Charles\",\"Body\":\"Q2hhcmxlcyBEYXRh\"}]"
```

The in-memory database is a slice of type **Page**. This slice and structure are shown below:

```
var xMem []Page // In Memory Database

type Page struct { // Database Pages
    Name string
    Body []byte
}
```

The external database file is JSON structured file. All database operations are handled by the JSON methods **Marshall** and **Unmarshall** and standard file I/O operations.

## Test Sequence Operations

Each test sequence for http handler functions does the following,

For the range of cases "c":

- Unmarsall the **c.initial_DB** into the in-memory database.
- Call the appropriate handler: xxxxHandler(**c.w, c.r**)
- For non-view handlers:
  - Unmarshall **c.returnedDB** into an auxiliary Page slice
  - Compare to current in-memory database.
- Compare **c.w.Code** to expected Response Code
- Compare **c.w.Body** to expected Response Body

Any detected errors are reported using the **t.Errorf** function calls. For Example:

```
if c.expectedResponseCode != c.w.Code {
          t.Errorf("Status Code didn't match:\nExpected Value:\t%d\nReturned Value:\t%d", c.expectedResponseCode, c.w.Code)
}
```

The only other test function is **TestLoadDatabase.** The complete test is shown below:

```
//
// Test Database Loader
//
func TestLoadDatabase(t *testing.T) {
    fmt.Println("Starting Database Test")
    err := os.Remove("Data.db")       // Remove Current Database(if any)
    if err != nil {
        fmt.Println("Database Not present - Creating")
    }

    loadDatabase()                   // Load Database Function

    data, err := json.Marshal(xMem)  // Marshall Database
    testCheck(err)

    if !reflect.DeepEqual(data, []byte(cajmj_db)) {
        t.Errorf("\nExpected = ", cajmj_db, "\nReturned = ", string(data))
    }
```

The complete source code and other files are available at [www.hawthorne-press.com](http://www.hawthorne-press.com).

Load the file ***coverage.html*** into your browser. This file shows all source line tested in green and any untested lines in red.

## A Final Word

As a developer, never get "married" to a design decision. If it becomes obvious that a piece of code is awkward or problematic, do not let pride keep you from changing it. *The only people who do not make mistakes are the ones who are not working.*

Strive for simplicity in your code. The *Occam's Razor principle* applies as strongly for software as it does for many other walks of life. In most situations, giving up a little in efficiency to simplify your code is often the right decision. Code Simplicity reduces the chances of subtle errors.

As always, comments are welcome at hawthornepresscom@gmail.com.

As to why I have an email address on **gmail** instead of on my website, has to do with AT&T and their inability to handle the concept that not everyone is running a Microsoft OS and my stubborn insistence on running Linux.

Revision History

| Date | By | Section | Changes |
|---|---|---|---|
| **04/15/2016** | C.E. Thornton | All | Initial Document |
| | | | |