RX-M Cloud Native Consulting

Go

Testing

Go provides support for automated testing via the `testing` package. In an effort to simplify testing, Go takes a very streamlined approach when it comes to testing functionality it provides. In a nutshell, they authors of Go hope you choose to build testing abstractions as you need them, versus hiding potentially useful details. Over time, patterns of usage, contributed libraries, and other familiar test constructs have become available if you are familiar with testing in another language. In this lab, we will focus on the built in functionality provided by Go.

1. Sample Application

Before diving into details, we will create our application we will test against. This application will allow us to exercise the basic test functionality such as unit tests, code coverage, and benchmarking. Our program will execute two different functions that calculate the same result. The first function, called `loop()` will use iteration to calculate a total; the second function, called `recursive()` will use recursion to calculate a total. We will compare the two methods and confirm they equate the same result value.

```
user@ubuntu:~$ mkdir -p ~user/go/src/lab-tests
user@ubuntu:~$ cd ~user/go/src/lab-tests/
user@ubuntu:~/go/src/lab-tests$
```

Create the `loop()` function:

```
user@ubuntu:~/go/src/lab-tests$ vim loop.go
user@ubuntu:~/go/src/lab-tests$ cat loop.go
```

```
package main

func loop(iter int) int {
        i := 0
        total := 0

        for i <= iter {
                total += i
                i++
        }

        return total
}
```

```
user@ubuntu:~/go/src/lab-tests$
```

Create the `recursive()` function:

```
user@ubuntu:~/go/src/lab-tests$ vim recursive.go
user@ubuntu:~/go/src/lab-tests$ cat recursive.go
```

```
package main

func recursive(iter int) int {
        if iter <= 1 {
                return iter
        }

        return iter + recursive(iter-1)
}
```

```
user@ubuntu:~/go/src/lab-tests$
```

Create the `main()` function:

```
user@ubuntu:~/go/src/gonuts$ vim main.go
user@ubuntu:~/go/src/gonuts$ cat main.go
```

```
package main

import "fmt"

func main() {
  iter := 3
  fmt.Println("loop:",loop(iter))
  fmt.Println("recursive:",recursive(iter))
}
```

```
user@ubuntu:~/go/src/gonuts$
```

Lets make sure everything works.

Run program:

```
user@ubuntu:~/go/src/lab-tests$ go run main.go loop.go recursive.go
loop: 6
recursive: 6
user@ubuntu:~/go/src/lab-tests$
```

Success, each solution takes the same input (3), specified in the `main.go`, and calculate the correct (and matching) output of 6.

Looping versus recursion is a great way to understand how you code affects your performance. We won't go into details here, but if your interested, just search "looping vs recursion" in your favorite search engine.

2. Test

Now that we have our sample application lets see what Go has to offer regarding testing. In the same directory, lets create a test for our `loop()` function.

```
user@ubuntu:~/go/src/lab-tests$ vim loop_test.go
user@ubuntu:~/go/src/lab-tests$ cat loop_test.go
```

```
package main

import "testing"

func TestLoop(t *testing.T) {
        t.Skip("todo")
}
```

```
user@ubuntu:~/go/src/lab-tests$
```

Lets run the test:

```
user@ubuntu:~/go/src/lab-tests$ go test
PASS
ok      lab-tests       0.002s
user@ubuntu:~/go/src/lab-tests$
```

Take a moment and review the code. We have several items to discuss, here is a list:

- Test file naming (ex. loop_test.go)
- Importing package `testing`
- Function signature (ex. func TestLoop(t *testing.T))

- Test function argument types (ex. testing.T)
- Test functions (ex. Skip())
- `go test` usage

When creating Go tests, we create a same name file with a `\_test.go` extension. This code will be in the same package as the code you are testing. In our example, the package is `main`. These test files `\*\_test.go` are ignored when doing normal builds.

To gain access to the testing automation our tests import the `testing` package. This package provides a variety of functions and types that we use to specify tests and results. The `go test` command is scanning for these files and related `testing` package code.

There are general categories of tests functionality provided by Go. In our previous example we see a function called `TestLoop(..)`. The available categories are:

- tests
- benchmarks
- examples

When writing tests, we prefix our function with "Test". When writing benchmark code (more on this later) we prefix it with "Benchmark". Example is similar to the others "Example" (more on this later).

We see our function is prefixed with Test, so it is a test type function. This type of function requires the argument `*testing.T`. The struct testing.T provides a way to manage test state and log format. A Test function is finished when you call any of the following methods against `t *testing.T`.

- FailNow - FailNow marks the function as having failed and stops its execution. Execution will continue at the next test or benchmark
- Fatal - Fatal is equivalent to Log followed by FailNow
- FatalF - FatalF is equivalent to Logf followed by FailNow
- SkipNow - SkipNow marks the test as having been skipped and stops its execution
- Skip - Skip is equivalent to Log followed by SkipNow
- Skipf - Skipf is equivalent to Logf followed by SkipNow

Related to the previous methods:

- Log - Record text to log file, only if test fails or `-test.v` flag is set
- Logf - Similar to Log, adds a newline

There are many more functions available, including `parallel` to run tests in parallel and `run` which runs a subtest to the current test.

- Review https://golang.org/pkg/testing

In our example we called `t.Skip("")` to indicate we haven't written our test. Lets write our first test to check for a correct value.

Remove the Skip and replace with the following:

```
user@ubuntu:~/go/src/lab-tests$ vim loop_test.go
user@ubuntu:~/go/src/lab-tests$ cat loop_test.go
```

```
package main

import "testing"

func TestLoop(t *testing.T) {
        result := loop(3)

        if result != 6 {
                t.Errorf("loop(%q) == %q, want %q", 3, result, 3)
        }
}
```

```
user@ubuntu:~/go/src/lab-tests$
```

Run the test:

```
user@ubuntu:~/go/src/lab-tests$ go test
PASS
ok      lab-tests       0.002s
user@ubuntu:~/go/src/lab-tests$
```

While its nice our single happy path test work, in Go we use 'table driven design' to set up many tests to really catch those errors. Lets add some more scenarios and loop over them.

```
user@ubuntu:~/go/src/lab-tests$ vi loop_test.go
user@ubuntu:~/go/src/lab-tests$ cat loop_test.go
```

```
package main

import "testing"

func TestLoop(t *testing.T) {
        cases := []struct {
                in   int
                want int
        }{
                {-1, 1},
                {3, 6},
                {0, 0},
        }

        for _, c := range cases {
                got := loop(c.in)

                if got != c.want {
                        t.Errorf("loop(%d) == %d, want %d", c.in, got, c.want)
                }
        }
}
```

```
user@ubuntu:~/go/src/lab-tests$
```

Run the test again:

```
user@ubuntu:~/go/src/lab-tests$ go test
--- FAIL: TestLoop (0.00s)
        loop_test.go:19: loop(-1) == 0, want 1
FAIL
exit status 1
FAIL    lab-tests       0.003s
user@ubuntu:~/go/src/lab-tests$
```

- Correct the test by fixing the incorrect input value.
- Create `recursive_test.go` and run similar tests as `loop_test.go`

The result of a test is PASS or Fail.

3. Tool Help

Now that we have a basic idea on how to create a test, take a few minutes and read the help menus for the tool itself and related concepts.

- `go help test`
- `go help testflag`
- `go help testfunc`
- `go test -h`

4. Benchmark

Lets create a benchmark test for our loop logic.

Append a benchmark test using the *Benchmark* prefix.

```
user@ubuntu:~/go/src/lab-tests$ vi loop_test.go
user@ubuntu:~/go/src/lab-tests$ cat loop_test.go
```

```
package main

import (
        "testing"
)

func TestLoop(t *testing.T) {
```

```
        cases := []struct {
                in   int
                want int
        }{
                {1, 1},
                {3, 6},
                {0, 0},
        }
        for _, c := range cases {
                got := loop(c.in)
                if got != c.want {
                        t.Errorf("loop(%d) == %d, want %d", c.in, got, c.want)
                }
        }
}

func BenchmarkLoop(b *testing.B) {
        for i := 0; i < b.N; i++ {
                loop(1000)
        }
}
```

```
user@ubuntu:~/go/src/lab-tests$
```

To run a benchmark, you must tell it which benchmark to run. This prevents us from the default behavior of just running all them all the time. You inform the test tool with a regex to match function signatures. For now, we match all functions labeled *Bench**.

```
user@ubuntu:~/go/src/lab-tests$ go test -bench=.
BenchmarkLoop-2          5000000               351 ns/op
PASS
ok      lab-tests       2.126s
user@ubuntu:~/go/src/lab-tests$
```

Our code uses `b.N` to set the number of times to loop over our code. The test driver (`go test`) sets this value as best it can to assist creation of useful statistic. Our results show us that the loop was executed 5,000,000 times at an average of 305 nanoseconds per execution.

The *PASS* is the results of your tests (ex. BenchmarkX).

- Create a benchmark for the `recursion()` method
- Run the benchmark again for both `loop()` and `recursion()`

```
user@ubuntu:~/go/src/lab-tests$ go test -bench=.
BenchmarkLoop-2          5000000               313 ns/op
BenchmarkRecursive-2      500000              2930 ns/op
PASS
ok      gonuts  3.404s
user@ubuntu:~/go/src/lab-tests$
```

As you can see, the `recursive()` call is much slower than then the `loop()` call.

- Why is the recursive call slower?

5. Example

We have now seen Test and Benchmark, the final functional area is called Example. When you add examples to to your `*_test.go` files, it will be used in the generated documentation.

```
user@ubuntu:~/go/src/gonuts$ vim loop_test.go
user@ubuntu:~/go/src/lab-tests$ cat loop_test.go
```

```
package main

import "testing"
import "fmt"

func TestLoop(t *testing.T) {
        cases := []struct {
                in   int
                want int
        }{
```

```
                {-1, 0},
                {3, 6},
                {0, 0},
        }

        for _, c := range cases {
                got := loop(c.in)

                if got != c.want {
                        t.Errorf("loop(%d) == %d, want %d", c.in, got, c.want)
                }
        }
}

func BenchmarkLoop(b *testing.B) {
        for i := 0; i < b.N; i++ {
                loop(1000)
        }
}

func ExampleFLoop() {
        fmt.Println(loop(3))
        // Output: 6
}
```

```
user@ubuntu:~/go/src/lab-tests$
```

In our case however, we are using the "main" package, so we can't simply generate non-package level documentation there. For now though, we can see our tests run along with view examples.

```
user@ubuntu:~/go/src/lab-tests$ go test -v
=== RUN   TestLoop
--- PASS: TestLoop (0.00s)
=== RUN   ExampleFLoop
--- PASS: ExampleFLoop (0.00s)
PASS
ok      lab-tests       0.002s
user@ubuntu:~/go/src/lab-tests$
```

6. Coverage

Another area of concern for testing, aside from correctness and speed is how much of our code is actually being used.
In general, 100% code coverage is hard to achieve. Even if we can't hit 100%, we can get a feel for the most used and least used (hot/cold) areas of our code, allowing us to focus on correctness, speed, and other features.

Lets generate the coverage by counting the times lines are executed.

```
user@ubuntu:~/go/src/lab-tests$ go test -coverprofile=cover.out
PASS
coverage: 50.0% of statements
ok      lab-tests       0.002s
user@ubuntu:~/go/src/lab-tests$
```

The flag *-coverprofile* write our coverage data to `cover.out`. In the previous case, we simply checked if statements were used at least once. We can also count by using the `-covermode=count` flag and value.

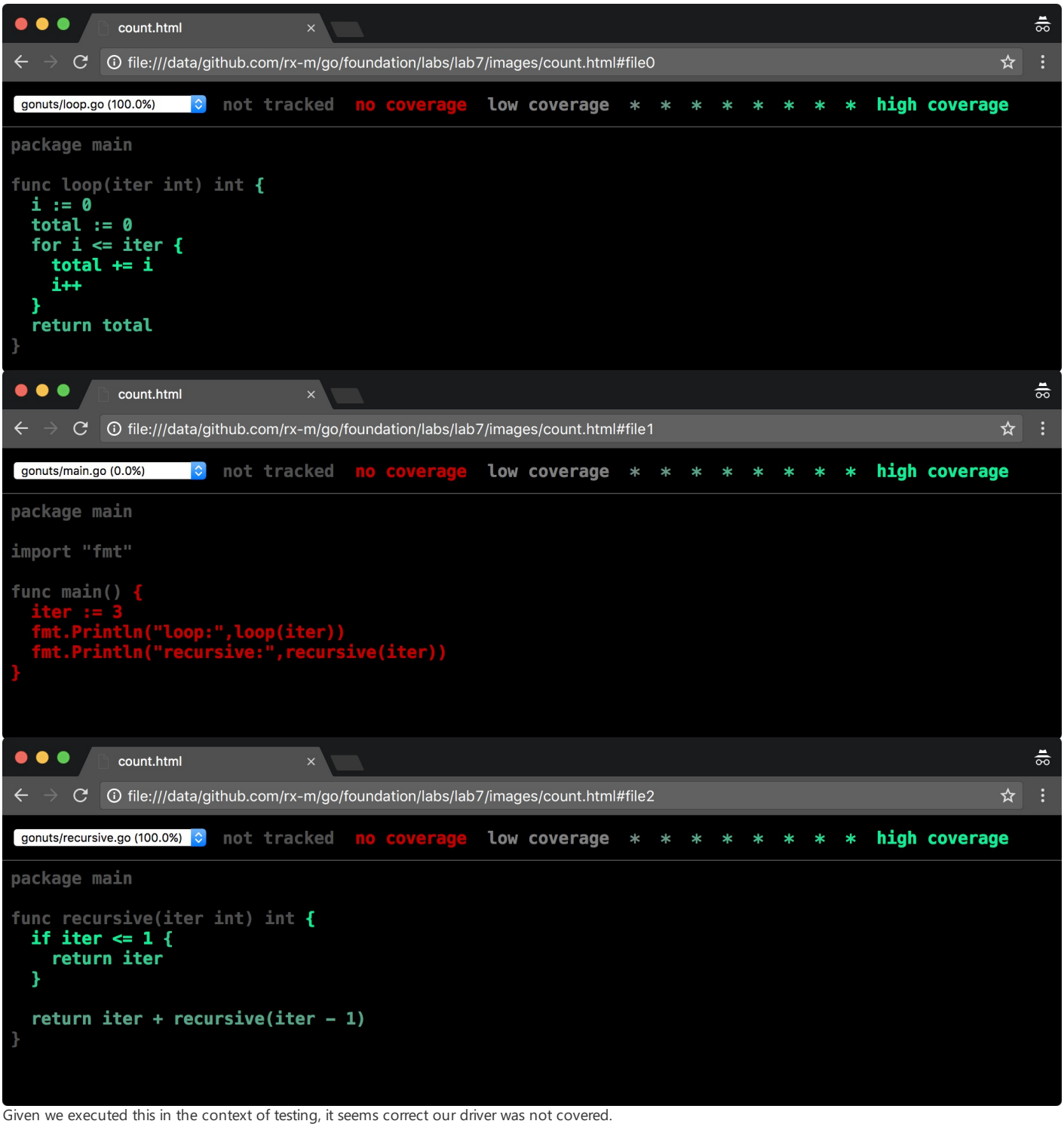
```
user@ubuntu:~/go/src/lab-tests$ go test -coverprofile=count.out -covermode=count
PASS
coverage: 50.0% of statements
ok      lab-tests       0.003s
user@ubuntu:~/go/src/lab-tests$
```

```
user@ubuntu:~/go/src/lab-tests$ diff -y count.out cover.out
mode: count                                             | mode: set
lab-tests/loop.go:3.25,7.16 3 4                         | lab-tests/loop.go:3.25,7.16 3 1
lab-tests/loop.go:12.2,12.14 1 4                                > lab-tests/loop.go:12.2,12.14 1 1
lab-tests/loop.go:7.16,10.3 2 11                                > lab-tests/loop.go:7.16,10.3 2 1
lab-tests/main.go:7.13,11.2 3 0                           lab-tests/main.go:7.13,11.2 3 0
lab-tests/recursive.go:3.30,4.15 1 0                      lab-tests/recursive.go:3.30,4.15 1 0
lab-tests/recursive.go:8.2,8.33 1 0                              lab-tests/recursive.go:8.2,8.33 1 0
```

```
lab-tests/recursive.go:4.15,6.3 1 0                    lab-tests/recursive.go:4.15,6.3 1 0
user@ubuntu:~/go/src/lab-tests$
```

We can then view the results as HTML via:

```
user@ubuntu:~/go/src/lab-tests$ go tool cover -html=count.out -o count.html
user@ubuntu:~/go/src/lab-tests$
```



Given we executed this in the context of testing, it seems correct our driver was not covered.

You can install FireFox to view the page in a browser:

```
user@ubuntu:~/go/src/lab-tests$ sudo apt-get install firefox
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  libdbusmenu-gtk4 xul-ext-ubufox
```

```
...

user@ubuntu:~/go/src/lab-tests$ go tool cover -html=count.out -o count.html
user@ubuntu:~/go/src/lab-tests$ ls -l count.html

-rw-rw-r-- 1 user user 3376 Aug  8 10:24 count.html

user@ubuntu:~/go/src/lab-tests$ firefox count.html
```

If you happen to be using your laptop browser versus a browser in the VM. Try to serve up the html via Go itself!

```
user@ubuntu:~/go/src/lab-tests$ vim web.go
user@ubuntu:~/go/src/lab-tests$ cat web.go
```

```
package main

import (
        "net/http"
)

func main() {
        http.Handle("/", http.FileServer(http.Dir("./")))
        http.ListenAndServe(":8080", nil)
}
```

```
user@ubuntu:~/go/src/lab-tests$
```

```
user@ubuntu:~/go/src/gonuts$ go run web.go
...
```

Open :8080 in your laptop browser!

7. TDD and BDD via Go

As mentioned earlier, Go tries to keep things simple. This simplification comes at a price. TDD usually falls in the realm of developers while BDD wants to move the testing away from the developer level and more towards business rules via a DSL. Since Go does not provide a BDD DSL, others have stepped in. One popular library is http://goconvey.co/.

If you are unfamiliar with BDD DSL, here is an example from `goconvey`.

```
...
func TestMyCode(t *testing.T) {
  Convey("For a scenario", t) {
    Convey("It should do this", func() {
      So(loop(3), ShouldEqual, "6")
      })
  }
}
...
```

The critical thing to notice is the more English like DSL versus what the TDD code looks like. If time permits, try to install in your VM and run.

8. Challenge

Pick any program that you wrote for a lab and write a short test for it. Verify that the test runs.

Congratulations you have completed the lab!