Testing

Testing

Testing in Go is easy, and simple to use. There is strong emphasis on testing in Go. The compiler will catch a lot of bugs for you, but it can not ensure your business logic is sound or bug-free.

Naming

Naming of files and functions play an incredibly important part of how tests work in Go.

Test Files

Test files in Go live next to the files that they are going to be testing.

```
foo.go
foo_test.go
```

They are identified by having the suffix `_test.go`. This is a **required** naming pattern by Go.

Simple Test

Tests **must** be in a `*_test.go` file and **must** be named `Test<name>(*testing.T)`.

```
package main

import "testing"

func TestSimple(t *testing.T) {
    if true {
        t.Error("expected false, got true")
    }
}
```

Ø

```
--- FAIL: TestSimple (0.00s)
 simple_test.go:7: expected false, got true
FAIL
```

Ø

*testing.T

The *testing.T type has the following methods available for us to use to control the flow of a teset.

```
Error(args ...interface{})
Errorf(format string, args ...interface{})
Fail()
FailNow()
Failed() bool
Fatal(args ...interface{})
Fatalf(format string, args ...interface{})
Log(args ...interface{})
Logf(format string, args ...interface{})
Name() string
Skip(args ...interface{})
SkipNow()
Skipf(format string, args ...interface{})
Skipped() bool
```

Table Driven Tests

The testing system in Go does not allow for writing custom `setup` and `teardown` functions that can be run before/after each test.

To work around this, several testing styles have appeared in the Go community, the first of which is known as "table driven tests".

Table Driven Tests

```go
func TestTableDriven(t *testing.T) {
    tt := []struct {
        A        int
        B        int
        Expected int
    }{
        {A: 1, B: 1, Expected: 2},
        {A: 2, B: 2, Expected: 4},
        {A: 3, B: 3, Expected: 5},
        {A: 4, B: 4, Expected: 6},
    }

    for _, x := range tt {
        got := x.A + x.B
        if got != x.Expected {
            t.Errorf("expected %d, got %d", x.Expected, got)
        }
    }
}
```

Ø

```
--- FAIL: TestTableDriven (0.00s)
 table_driven_test.go:20: expected 5, got 6
 table_driven_test.go:20: expected 6, got 8
```

Ø

Sub Tests

One of the short comings with table driven tests is it can often be confusing as to which test was the failing test. It is also difficult to run just a single test in the loop.

In Go 1.7 the ability to run "sub tests" were added to address these issues.

Sub Tests

```
func TestSub(t *testing.T) {
    tt := []struct {
        A        int
        B        int
        Expected int
    }{
        {A: 1, B: 1, Expected: 2},
        {A: 2, B: 2, Expected: 4},
        {A: 3, B: 3, Expected: 5},
        {A: 4, B: 4, Expected: 6},
    }

    for i, x := range tt {
        t.Run(fmt.Sprintf("sub test (%d)", i), func(st *testing.T) {
            got := x.A + x.B
            if got != x.Expected {
                st.Errorf("expected %d, got %d", x.Expected, got)
            }
        })
    }
}
```

Ø

Sub Tests

Because we can give each of our sub tests names, the output of the failed tests makes it more clear which iterations of the test loop failed.

```
--- FAIL: TestSub (0.00s)
    --- FAIL: TestSub/sub_test_(2) (0.00s)
     sub_test.go:24: expected 5, got 6
    --- FAIL: TestSub/sub_test_(3) (0.00s)
     sub_test.go:24: expected 6, got 8
```

Ø

Later we will look at how using sub tests we can run just the tests we are interested, and not the entire loop of tests.

Magic Testing Package

Sometimes when testing a package we might hit on a circular dependency, an issue caused by two packages trying to import each other.

To solve this problem, Go let's us create a "magic" testing package.

```
// foo.go
package foo
```

```
// foo_test.go
package foo_test
```

Magic Testing Package

The `_test` package is the exception to the "one package per folder" requirement of Go.

Caveat - Since the `_test` package is technically a new package, all of the rules around the exporting of names (public/private) are in effect.

```
// foo.go
package foo

// not-exported
var a string
// exported
var B string
```

Running Package Tests

You can run all of the tests in a package using the `go test .` command, where `.` represents the current folder.

```
$ go test .

ok      github.com/gobuffalo/plush    0.016s
```

Running Tests With Sub-Packages

Often, Go projects will consist of multiple packages. To run all of these packages we can use the `./...` identifier to tell Go to recurse through all sub-packages as well as the current one.

```
$ go test ./...

ok      github.com/gobuffalo/plush    0.016s
ok      github.com/gobuffalo/plush/ast    0.011s
ok      github.com/gobuffalo/plush/lexer    0.013s
ok      github.com/gobuffalo/plush/parser    0.012s
?       github.com/gobuffalo/plush/token    [no test files]
```

Verbose Test Output

In can be useful, for example in CI environments, to output "verbose" information when running tests. For example, seeing which tests are running as well as debugging information. The `-v` flag turns on this verbose output.

```
$ go test -v .

  === RUN   Test_ContentForOf
  --- PASS: Test_ContentForOf (0.00s)
  === RUN   Test_Context_Set
  --- PASS: Test_Context_Set (0.00s)
  === RUN   Test_Context_Set_Concurrency
  --- PASS: Test_Context_Set_Concurrency (0.00s)
  === RUN   Test_Context_Get
  --- PASS: Test_Context_Get (0.00s)
  === RUN   ExampleRender_nilValue
  --- PASS: ExampleRender_nilValue (0.00s)
  === RUN   ExampleRender_forIterator
  --- PASS: ExampleRender_forIterator (0.00s)
PASS
ok      github.com/gobuffalo/plush    0.017s
```

Run Specific Tests

The `-run` flag allows for the passing of a regular expression to match the names of specific tests.

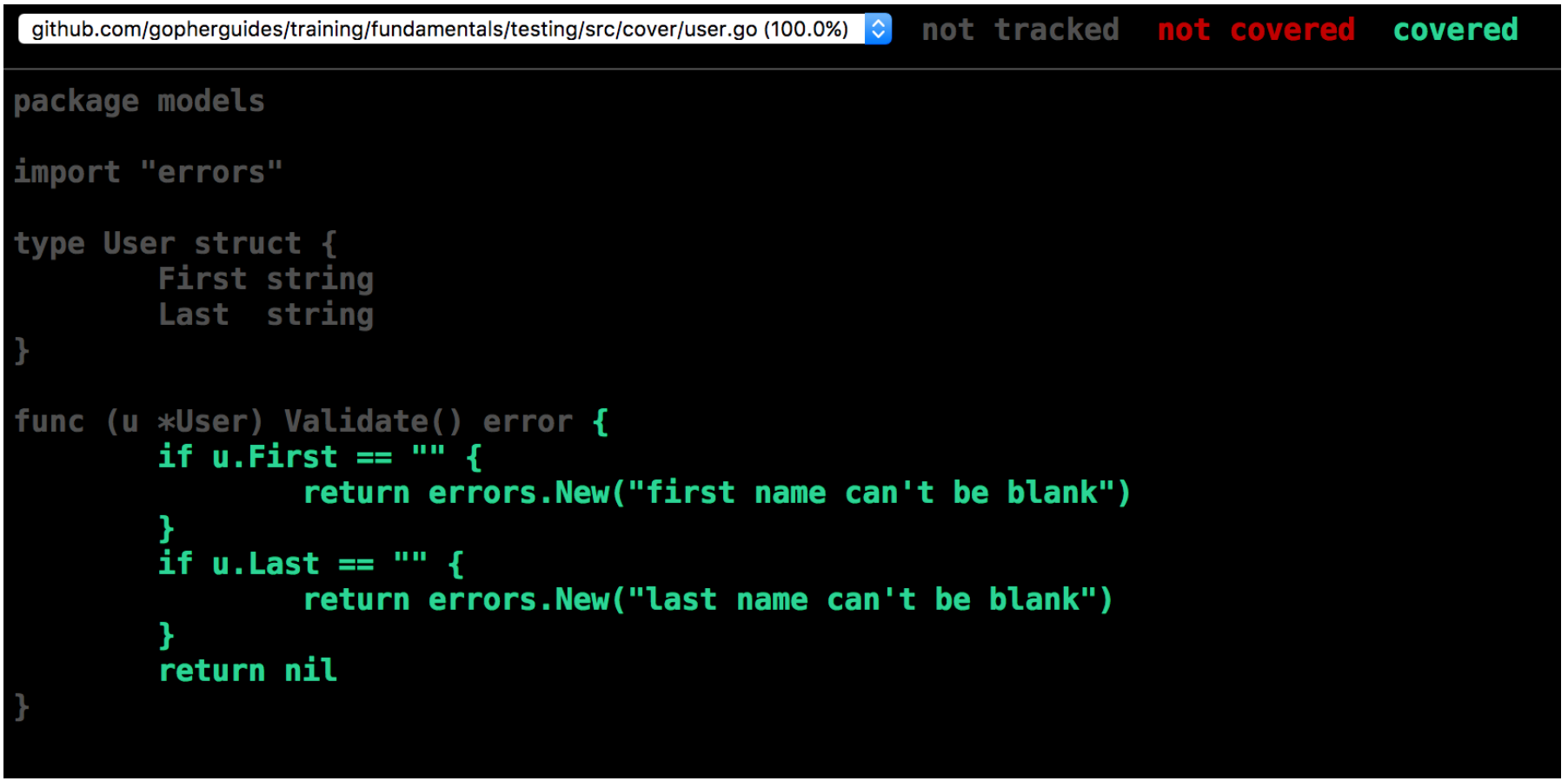
```
$ go test -run "Call" -v ./...

  === RUN   Test_Render_Function_Call
  --- PASS: Test_Render_Function_Call (0.00s)
  === RUN   Test_Render_UnknownAttribute_on_Callee
  --- PASS: Test_Render_UnknownAttribute_on_Callee (0.00s)
  PASS
  ok      github.com/gobuffalo/plush    0.012s
  testing: warning: no tests to run
  PASS
  ok      github.com/gobuffalo/plush/ast    0.011s [no tests to run]
  testing: warning: no tests to run
  PASS
  ok      github.com/gobuffalo/plush/lexer    0.013s [no tests to run]
  === RUN   Test_CallExpression
  --- PASS: Test_CallExpression (0.00s)
  === RUN   Test_CallExpressionParsing_WithCallee
  --- PASS: Test_CallExpressionParsing_WithCallee (0.00s)
  PASS
  ok      github.com/gobuffalo/plush/parser    0.010s
  ?       github.com/gobuffalo/plush/token    [no test files]
```

Code Coverage

Go has built in tooling to generate code coverage

```
go test -coverprofile cover.out
go tool cover -html=cover.out
```



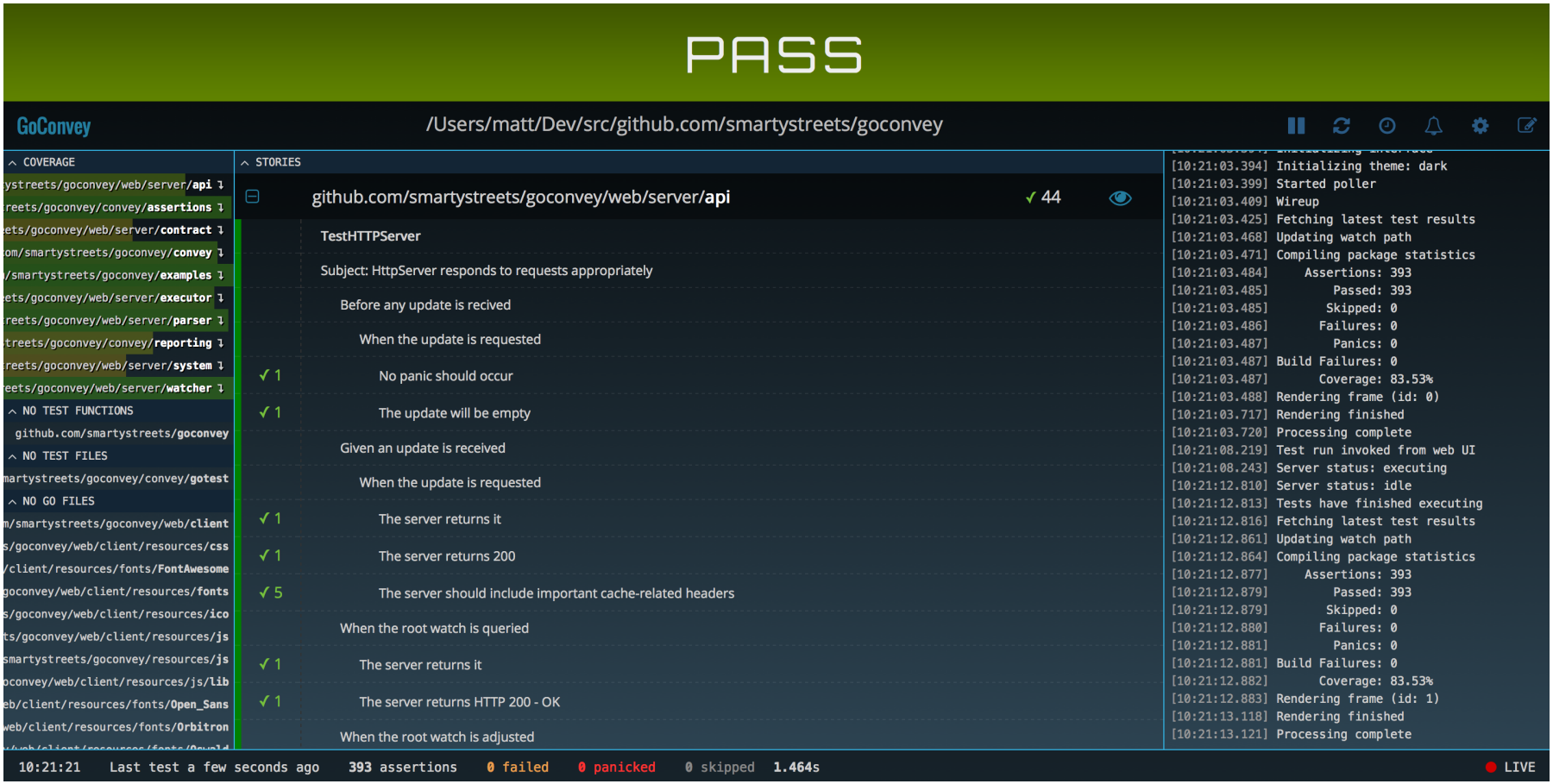
Code Coverage For A Specific Test

You can run code coverage just for a specific test:

```
go test -coverprofile cover.out -run TestUser_Validate
go tool cover -html=cover.out
```

GoConvey

For smaller projects, GoConvey is a great tool.



Race Conditions

Go is a concurrent language, and despite Go trying to make writing concurrent applications as simple as possible, it is still possible to run into a "race" condition.

A "race" condition is when a single piece of memory is trying to be accessed by multiple processes at the same time.

Race Conditions

Let's look at this simple, yet contrived, example.

```
var m = 0

func inc() {
    m++
}
```
Ø

Race Conditions

This test uses Goroutines to read and write from the `m` variable at the same time.

```
func TestRace(t *testing.T) {
    wg := &sync.WaitGroup{}
    for i := 0; i < 5; i++ {
        wg.Add(2)
        go func() {
            defer wg.Done()
            inc()
        }()
        go func() {
            defer wg.Done()
            fmt.Println(m)
        }()
    }
    wg.Wait()
    if m == 0 {
        t.Fatalf("expect m (%d) to not be zero", m)
    }
}
```

Ø

```
$ go test race_test.go
ok      command-line-arguments    0.006s
```

Race Conditions

If we were to run this code, which passes the aforementioned test, we would get a `panic` that looks something like the following:

```
Previous write at 0x0000012bb618 by goroutine 7:
  command-line-arguments.TestRace.func1()
      fundamentals/testing/src/race_test.go:21 +0x83

Goroutine 8 (running) created at:
  command-line-arguments.TestRace()
      fundamentals/testing/src/race_test.go:26 +0xd4
  testing.tRunner()
      /usr/local/go/src/testing/testing.go:657 +0x107
```

So how do we test for race conditions to make sure our code will work in a concurrent environment?

Testing Race Conditions

Using the `-race` flag when running tests will check for race conditions in our code.

```
$ go test -race race_test.go

  ==================
  WARNING: DATA RACE
  Read at 0x0000012bb618 by goroutine 8:
    runtime.convT2E()
        /usr/local/go/src/runtime/iface.go:191 +0x0
    command-line-arguments.TestRace.func2()
        fundamentals/testing/src/race_test.go:25 +0x86

  Previous write at 0x0000012bb618 by goroutine 7:
    command-line-arguments.TestRace.func1()
        fundamentals/testing/src/race_test.go:21 +0x83
  --- FAIL: TestRace (0.00s)
    testing.go:610: race detected during execution of test
  FAIL
  FAIL    command-line-arguments    0.013s
```

Testing Race Conditions

Using the `-race` flag will slow down your tests, so it is recommended to **always** set it in your CI environment, and to run your tests locally with it *before* committing code.

Exercises

Write a simple calculate package and then write some tests for it.

```
func Add(a, b float64) float64 {}
func Subtract(a, b float64) float64 {}
func Multiply(a, b float64) float64 {}
func Divide(a, b float64) float64 {}
```

```
// rounds floats to 0.xx precision
func round(f float64) (float64, error) {
  return strconv.ParseFloat(fmt.Sprintf("%.2f",f), 64)
}
```