Systems Programming

Unit testing in Go

Daniel Lundberg

Today

- Unit testing in go
  - What is unit testing
  - Testing example
  - Parametrized testing
  - Testify
  - Whitebox testing

What is Unit Testing?

- **Unit testing** = testing small, isolated parts of your code (e.g., a function or method)
- Goal: verify that each unit works as intended
- Helps you to:
  - catch bugs early
  - make changes with confidence
  - improve overall code quality

💡 *Good practice:* Write tests along with your code!

Why We Use It

- We will use **unit testing** in our projects
- We will also measure **code coverage** – how much of the code is tested
- Higher coverage → more reliable code (but not everything!)

📊 Code coverage helps us see: - which lines and cases are tested
- where tests are missing

🧩 Goal: understand *how* and *why* we test, not just to reach 100%

Todays project structure

We will build a small **Go calculator project** to learn about unit testing and code coverage.
This structure helps keep code and tests clean and organized.

```
test_project/
│
├── main.go # Entry point of the program
└── calc/ # Package for calculator logic
  └── calc.go # Calculator functions (add, subtract, etc.)
```

calc/calc.go

```go
package calc
func Add(a, b float64) float64 { //Capital A = Public
    return a + b
}
func Substact(a, b float64) float64 {
    return a - b
}
func Multiply(a, b float64) float64 {
    return a * b
}
func Divide(a, b float64) float64 {
    return a / b
}
```

main.go

```
package main
import (
    "fmt"
    "myproject/calc"
)
func main() {
  a, b := 1.0, 2.0
  fmt.Printf("%.2f+%.2f=%.2f\n", a, b, calc.Add(a, b))
  fmt.Printf("%.2f-%.2f=%.2f\n", a, b, calc.Substact(a, b))
  fmt.Printf("%.2f*%.2f=%.2f\n", a, b, calc.Multiply(a, b))
  fmt.Printf("%.2f/%.2f=%.2f\n", a, b, calc.Divide(a, b))
}
```

output main.go:5:2: package test_project/calc is not in std

We need to init the project

- **Module declaration:** Creates a Go module so the project is recognized as a single unit.
- **Dependency management:** Allows proper handling of external packages and versions.
- **Multi-package support:** Lets us import our own packages from different folders (e.g., `calc/`) correctly.

```
>> Go mod init test_project
```

- Create go.mod that defines the module for your Go project, managing its path, dependencies, and versions.

Real test

Testing by running your code is not the same as testing for real, we will now create **unit tests**.

test file should be placed in same folder as the file they could test (/calc in our case), and be named with suffix _test, so in our case **calc_test.go**

```
test_project/
│
├── go.mod # Go module definition
├── main.go # Entry point of the program
└── calc/ # Package for calculator logic
  └── calc.go # Calculator functions (add, subtract, etc.)
  └── calc_test.go # Unit tests for calc.go
```

Set up calc_test.go

- Whitebox testing – Tester knows the internal code and writes tests based on the logic, branches, and paths inside the program. Use same package name
- Blackbox testing – Tester only sees inputs and outputs, ignoring how the code works internally. Use package name _test

```
package calc_test
import (
    "test_project/calc" // not needed if package calc
    "testing"
)
```

Set up calc_test.go

```
..
// Function Name Test+Name of function to test
func TestAdd(t *testing.T) {
    // Arrange
    a, b := 5.0, 6.0
    expected := 11.0
    // Act
    got := calc.Add(a, b)
    //Assert
    if got != expected {
        t.Errorf("Expected %.2f got %.2f", expected, got)
    }
}
```

Run test

» To run the test write **go test**

```
>>go test ./calc
>>go test -v ./calc //more details
```

» Hoppfully we get something like this in response

```
>>ok      test_project/calc  0.221s
```

Add more tests

```go
func TestMultiply(t *testing.T) {
    a, b := -1.0, 0.5 //Arrange
    expected := -0.5
    got := calc.Multiply(a, b) //Act
    if got != expected {       //Assert
        t.Errorf("Expected %.2f got %.2f", expected, got)
    }
}
```

Add more tests

```go
func TestDivide(t *testing.T) {
    a, b := 1.0, 0.5 //Arrange
    expected := 2.0
    got := calc.Divide(a, b) //Act
    if got != expected {     //Assert
        t.Errorf("Expected %.2f got %.2f", expected, got)
    }
}
```

Add more tests

```go
func TestSubstact(t *testing.T) {
    a, b := 1.0, 0.5 //Arrange
    expected := 0.5
    got := calc.Substact(a, b) //Act
    if got != expected {       //Assert
        t.Errorf("Expected %.2f got %.2f", expected, got)
    }
}
```

```
>> go test -v  .\calc\
```

The Results

```
=== RUN   TestAdd
--- PASS: TestAdd (0.00s)
=== RUN   TestMultiply
--- PASS: TestMultiply (0.00s)
=== RUN   TestDivide
--- PASS: TestDivide (0.00s)
=== RUN   TestSubstact
--- PASS: TestSubstact (0.00s)
PASS
ok      test_project/calc       0.179s
```

Parametrized testing

- A technique where one test function runs multiple test cases with different inputs and expected outputs.
- Reduces code **duplication** — no need to write a new test for each scenario.
- Makes tests more organized, concise, and maintainable.
- Example: test many input/output pairs for the same function (e.g., Add()).

Benefits in Go

- Encourages consistent test structure (standard Go testing pattern).
- Easy to add new cases without changing logic — just extend the table.
- Fits naturally with Go's testing package (no external libraries needed).

Create our testcases

```
var testcases_divide = []struct {
    name        string
    expected    float64
    numerator   float64
    denominator float64
}{
    {"division", 5.0, 10.0, 2.0},
    {"division with neg num", -5.0, -10.0, 2.0},
    {"division with neg denum", -5.0, 10.0, -2.0},
}
```

We’re defining a slice of test cases where each element holds the data needed to test the divide function. Each struct stores the test name, input values (numerator, denominator), and the expected result.

Implement Parametrized testing

```
func TestDivide(t *testing.T) {
    for _, tc := range testcases_divide {
        t.Run(testCase.name, func(t *testing.T) {
            expected := tc.expected
            got := calc.Divide(tc.numerator, tc.denominator)

            if got != expected { //Assert
                t.Errorf("Expected %.2f got %.2f", expected, got)
            }
        }) // End run
    }
}
```

```
>> go test -v .\calc\
```

Results

```
..
=== RUN   TestDivide
=== RUN   TestDivide/division
=== RUN   TestDivide/division_with_neg_num
=== RUN   TestDivide/division_with_neg_denum
--- PASS: TestDivide (0.00s)
    --- PASS: TestDivide/division (0.00s)
    --- PASS: TestDivide/division_with_neg_num (0.00s)
    --- PASS: TestDivide/division_with_neg_denum (0.00s)
..
```

Divide by zero?

Update our testcase

```
var testcases_divide = []struct {
    name        string
    expected    float64
    numerator   float64
    denominator float64
}{
    {"division", 5.0, 10.0, 2.0},
    {"division with neg num", -5.0, -10.0, 2.0},
    {"division with neg denum", -5.0, 10.0, -2.0},
    {"division by zero", 0.0, 5.0, 0.0},
}
```

```
>> go test -v .\calc\
```

```
--- FAIL: TestDivide/division_by_zero (0.00s)
        calc_test.go:50: Expected 0.00 got +Inf
```

Update our Divide-function

We want our divide function to return an error if you try to divide by zero.

```
func Divide(a, b float64) (float64, error) {
    if b == 0.0 {
        return 0.0, errors.New("division by zero")
    }
    return a / b, nil
}
```

Update our TestDivide

```
func TestDivide(t *testing.T) {
    for _, tc := range testcases_divide {
        t.Run(tc.name, func(t *testing.T) {
            expected := tc.expected
            got, err := calc.Divide(tc.numerator, tc.denominator)
            if err != nil{}
                //  error.New('text') != error.New('text')
            if got != expected { //Assert
                t.Errorf("Expected %.2f got %.2f", expected, got)
            }
        }) // End run
    }
}
```

Compare two different Errors even with same message will get false

One solution Testify

```
>> go get github.com/stretchr/testify
go: added github.com/stretchr/testify v1.11.1
```

sometimes needed

```
>>go mod tidy
```

Add supports for asserts

Update TestCases

```
var testcases_divide = []struct {
    name        string
    expected    float64
    numerator   float64
    denominator float64
    wantError   bool
}{
    {"division", 5.0, 10.0, 2.0, false},
    {"division with neg num", -5.0, -10.0, 2.0, false},
    {"division with neg denum", -5.0, 10.0, -2.0, false},
    {"division by zero", 0.0, 5.0, 0.0, true},
}
```

Update TestDivide

```
func TestDivide(t *testing.T) {
    for _, tc := range testcases_divide {
        t.Run(tc.name, func(t *testing.T) {
            assert := assert.New(t)
            got, error := calc.Divide(tc.numerator, tc.denominator)

            if tc.wantError {
                assert.Error(error)
            }
            assert.Equal(tc.expected, got)
        }) // End run
    }
}
```

Whitebox Testing

Whitebox

» To be able to test internal functions e.g. starts with lowercase

```
package mathematics
import "math"
func Sin(angle float64) float64 {
    angle = wrapAngle(angle)
    return math.Sin(angle)
}
func wrapAngle(angel float64) float64 {
    wrapped_angle := math.Mod(angel, 360)
    if wrapped_angle < 0 {
        wrapped_angle += 360
    }
    return wrapped_angle
}
```

Whitebox

» You must have same package on your test-class mathematics_test.go

```
package mathematics

import (
    "testing"

    "github.com/stretchr/testify/assert"
)
```

Whitebox - Testcases

We just test the internal function here

```
var testcases_wrapAngle = []struct {
    name     string
    angle    float64
    expected float64
}{
    {"Less than 360", 200.0, 200.0},
    {"more than 360", 400.0, 40.0},
}
```

Whitebox - Test

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

    for _, tc := range testcases_wrapAngle {
        t.Run(tc.name, func(t *testing.T) {
            assert := assert.New(t)
            got := wrapAngle(tc.angle)

            assert.Equal(tc.expected, got)
        })
    }
}
```

Testing commandos

Test all test_packages

```
>>go test -v ./...
```

Test coverage

```
>>go test ./... -cover
```

Coverage report

```
>>go test ./... -coverprofile=coverage.out
```

Show report function-by-function

```
>>go tool cover -func=coverage
```