Unit testing in Go

Introduction

Unit testing is a way of testing your code to ensure that it is working as expected. Unit testing is a very important part of the development process. It allows you to verify that your code is working as expected. It also allows you to verify that your code is not working as expected. Unit testing is a very important part of the development process. It allows you to verify that your code is working as expected. It also allows you to verify that your code is not working as expected.

We use in Go the package testing to write unit tests. The testing package provides a framework for writing unit tests.

Testing principles

There are a few principles that we should follow when writing tests:

- Tests should be isolated from each other
- Tests should be repeatable
- Tests should be deterministic
- Tests should be fast
- Tests should be readable
- Tests should be maintainable
- Tests should be reliable
- Tests should be independent

Testing package

The main function in the testing package is the Test function. The Test function takes a testing.T as a parameter. The testing.T is used to report the results of the test. The testing.T has a number of methods that can be used to report the results of the test. The testing.T has a number of methods that can be used to report the results of the test.

Test function

```
func Test(t *testing.T) {
    // ...
}
```

To notify the testing framework that a test has failed, we can use the Error method on the testing.T.;

Error methods:

- Log: Log formats its arguments using default formatting, analogous to Println, and records the text in the error log.

- Logf: Logf formats its arguments according to the format, analogous to fmt.Printf, and records the text in the error log.
- Error: Error is equivalent to Log followed by Fail.
- Errorf: Errorf formats its arguments according to the format, analogous to fmt.Printf, and records the text in the error log.
- Fail: Fail marks the function as having failed but continues execution.
- FailNow: FailNow marks the function as having failed and stops its execution by calling runtime.Goexit (which then runs all deferred calls in the current goroutine).
- Failed: Failed reports whether the function has failed.
- Fatal: Fatal is equivalent to Log followed by FailNow.
- Fatalf: Fatalf is equivalent to Logf followed by FailNow.
- Warning: Warning is equivalent to Log followed by Skip.
- Warningf: Warningf is equivalent to Logf followed by Skip.
- Skip: Skip is equivalent to Log followed by SkipNow.
- SkipNow: SkipNow marks the function as having failed and stops its execution by calling runtime.Goexit (which then runs all deferred calls in the current goroutine).
- Skipf: Skipf is equivalent to Logf followed by SkipNow.
- Skipped: Skipped reports whether the function has been skipped.

Error method

```
func Test(t *testing.T) {
    // ...
    t.Error("Expected 'Hello, world.'")
    // ...
}
```

Scaffolding

For this example, we will use the following scaffolding:

$ tree -L 1 .

├── main.go ├── main_test.go └── README.md

0 directories, 3 files

main.go

main.go

```
package main

import (
    "fmt"
)

func helloWorld() string {
    return "Hello, world."
}

func main() {
    fmt.Println("Hello, world.")
}
```

main_test.go

main_test.go

```
package main

import (
    "testing"
)

func TestMain(t *testing.T) {
    main()
}
```

Running the tests

```
$ go test
PASS
ok      _/home/username/go/src/github.com/username/hello-world 0.001s
```

Adding a test

Let’s add a test to our code:

main_test.go

```
package main

import (
    "testing"
)

func TestMain(t *testing.T) {
    main()
}

func TestHelloWorld(t *testing.T) {
    if helloWorld() != "Hello, world." {
        t.Error("Expected 'Hello, world.'")
    }
}
```

Testing blocks

There are a few blocks that we should use when writing tests:

- Setup: The setup block is used to prepare the test for execution. This is where we will create any objects that we need for the test.
- Exercise: The exercise block is where we will execute the code that we are testing.
- Verify: The verify block is where we will verify that the code that we are testing is working as expected.
- Teardown: The teardown block is where we will clean up any objects that we created in the setup block.

Example setup block

```
func TestHelloWorld(t *testing.T) {
    // Setup
    expected := "Hello, world."

    // Exercise
    actual := helloWorld()

    // Verify
    if actual != expected {
        t.Error("Expected 'Hello, world.'")
    }

    // Teardown

}
```

Assertions

Assertions are used to verify that the code that we are testing is working as expected. Assertions is a package that provides a number of assertions that we can use when writing tests.

To install the assertions package

```
$ go get github.com/stretchr/testify/assert
```

There are a few assertions that we can use when writing tests:

- Equal: The Equal assertion is used to verify that two values are equal.
- Nil: The Nil assertion is used to verify that a value is nil.
- True: The True assertion is used to verify that a value is true.
- False: The False assertion is used to verify that a value is false.
- Contains: The Contains assertion is used to verify that a string contains a substring.
- Error: The Error assertion is used to verify that a function returns an error.
- Panics: The Panics assertion is used to verify that a function panics.

Example with several assertions assertion.method

```
package main

import (
    "testing"
    "github.com/stretchr/testify/assert"
)

func TestHelloWorld(t *testing.T) {
    // Setup
    expected := "Hello, world."

    // Exercise
    actual := helloWorld()

    // Verify
    assert.Equal(t, expected, actual)
    assert.Nil(t, nil)
    assert.True(t, true)
    assert.False(t, false)
    assert.Contains(t, "Hello, world.", "world")
    assert.Error(t, nil)
    assert.Panics(t, func() { panic("Panic!") })

    // Teardown

}

func helloWorld() string {
    return "Hello, world."
}
```

Testing resources

- official documentation