INTRODUCCIÓN A LA CLASE

GO TESTING

UNIT TEST

GO TESTING

Objetivos de esta clase

- Afianzar el concepto de Test Unitario
- Conocer las herramientas necesarias para el testing.
- Diseñar y ejecutar tests unitarios con Go.
- Evaluar los resultados.

// Unit Test

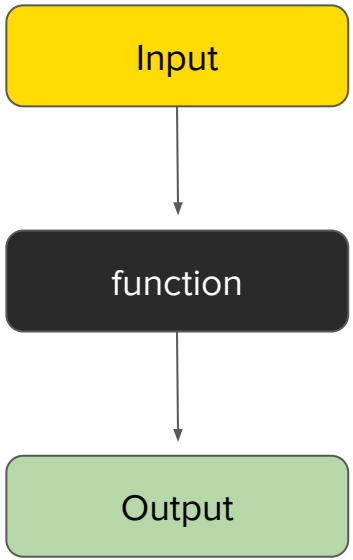
Un test unitario es la forma de probar una parte pequeña del código, lo más atomizada posible. Usualmente estos son cada método o cada función.

IT BOARDING

BOOTCAMP

Características

- Pueden ser ejecutados en cualquier orden.
- No requieren acceso a ningún repositorio de datos.
- Deben probar bloques de código atómicos.
- Son legibles y fáciles de comprender
- Los resultados arrojados deben ser claros y concisos.



Test Result ⟺ Output Validation

// ¿Cuándo hacer un Unit Test?

Se considera que el Unit Testing es parte del proceso de programación y desarrollo. Es decir, empezar a programar es empezar a crear test unitarios.

PACKAGE TESTING

GO TESTING

// Package “testing”

Es una librería nativa de Go, que provee las herramientas necesarias para el diseño e implementación de tests. También se encarga de la ejecución de forma automatizada de los test diseñados a través del comando `go test`

Nuestro primer test unitario.

Para nuestro primer test, tomaremos el siguiente código como objeto de pruebas.

{}

```go
package calculadora

// Función que recibe dos enteros y retorna la suma resultante
func Sumar(num1, num2 int) int {
    return num1 + num2
}

// Función que recibe dos enteros y retorna la resta o diferencia resultante
func Restar(num1, num2 int) int {
    return num1 - num2
}
```

Nuestro primer test unitario.

{}

```go
package calculadora

// Se importa el package testing
import "testing"

func TestSumar(t *testing.T) {
    // Se inicializan los datos a usar en el test (input/output)
    num1 := 3
    num2 := 5
    resultadoEsperado := 8

    // Se ejecuta el test
    resultado := Sumar(num1, num2)

    // Se validan los resultados
    if resultado != resultadoEsperado {
        t.Errorf("Funcion suma() arrojo el resultado = %v, pero el esperado es  %v", resultado, resultadoEsperado)
    }
}
```

Nuestro primer test unitario.

Finalmente procedemos a la ejecución del test:

```
$ go test
```

```
output
PASS
ok
go-testing/calc 1.776s
```

Consideraciones importantes

- El archivo calculadora.go y **calculadora_test.go** pertenecen al mismo package "**calculadora**". Esto permite acceder directamente a las funciones testeadas.

- La firma de cada función de prueba inicia con el nombre "**Test**". Esto permite que la función pueda ser reconocida por la rutina automatizada que los ejecuta.

- El método **t.Errorf()** es el encargado de ***registrar el error*** así como mostrar formateados los argumentos usados.

Testeando el error

Hagamos ahora el mismo test pero cambiando el comportamiento de la función Suma.

{}

```
package calculadora

// Función que recibe dos enteros y retorna la suma resultante
func Sumar(num1, num2 int) int {
    // Esta función ahora devuelve un resultado INCORRECTO
    return num1 - num2
}

// Función que recibe dos enteros y retorna la resta o diferencia resultante
func Restar(num1, num2 int) int {
    return num1 - num2
}
```

Testeando el error

```
$ go test
```

```
output
--- FAIL: TestSumar (0.00s)
    calculadora_test.go:17: Funcion suma() arrojo el
resultado = -2, pero el esperado es  8
FAIL
exit status 1
FAIL    go-testing/calc 0.635sPASS
ok
go-testing/calc 1.776s
```

PACKAGE TESTIFY

GO TESTING

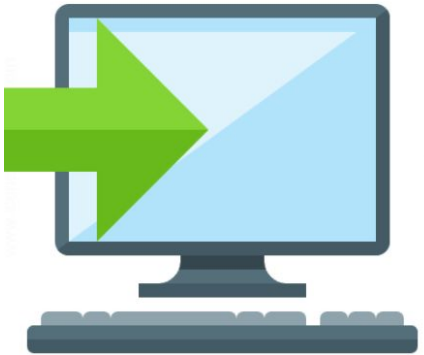
// Package “testify”

Es un paquete que facilita el desarrollo y la implementación de los test.

Instalando Testify

De la misma manera que instalamos cualquier módulo o librería, podemos instalar testify a través del comando “go get”

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



Implementando Testify

Sobre el mismo file en el que venimos trabajando “calculadora_test.go” **implementamos testify** de la siguiente manera;

{}

```go
package calculadora

import (
    "testing"


    "github.com/stretchr/testify/assert" // Se importa testify
)

func TestSumar(t *testing.T) {
    // Se inicializan los datos a usar en el test (input/output)
    num1 := 3
    num2 := 5
    resultadoEsperado := 8

    // Se ejecuta el test
    resultado := Sumar(num1, num2)

    // Se validan los resultados
    assert.Equal(t, resultadoEsperado, resultado, "deben ser iguales")

}
```

Entendiendo Testify

Las validaciones se hacen a través del package“assert”, que nos ofrece de forma sencilla efectuar las comparaciones y validaciones. Las funciones principales disponibles son:

// **Validar Igualdad** -> **assert.Equal**(t, 123, 123, "deberían ser iguales")

// Validar **Desigualdad** -> **assert.NotEqual**(t, 123, 456, "no deberían ser iguales")

// Validar **Nulo Esperado** (Bueno para errores) -> **assert.Nil**(t, object)

//Validar **No Nulo Esperado** (Bueno para cuando esperamos algo) -> **assert.NotNil**(t, object)

Ejecutando con testify

Si tomamos nuestro código original, con la función Sumar() funcionando correctamente y ejecutamos el test en el que implementamos testify, este será el resultado:

```
$ go test
```

```
output
PASS
ok
go-testing/calc 1.776s
```

Ejecutando con testify

Tal como pudimos comprobar, al implementar testify también pudimos testear la función Sumar() satisfactoriamente. Con la ventaja en este caso, que ***se desarrollaron menos líneas de código para hacer exactamente el mismo test***.

En un test simple como el de este ejemplo, ahorrar dos líneas puede que no represente una carga importante. Pero a medida que los test toman complejidad, se hace mucho más provechoso el uso de testify.

Testeando el error con Testify

Intentemos ahora la detección de errores implementando testify. Este es nuestro código defectuoso

```
{}
```

```go
package calculadora

// Función que recibe dos enteros y retorna la suma resultante
func Sumar(num1, num2 int) int {
    // Esta función ahora devuelve un resultado INCORRECTO
    return num1 - num2
}

// Función que recibe dos enteros y retorna la resta o diferencia resultante
func Restar(num1, num2 int) int {
    return num1 - num2
}
```

Testeando el error con Testify

```
$ go test
```

output

```
--- FAIL: TestSumar (0.00s)
    calculadora_test.go:19:
                Error Trace:    calculadora_test.go:19
                Error:          Not equal:
                                expected: 8
                                actual  : -2
                Test:           TestSumar
                Messages:       deben ser iguales
FAIL
exit status 1
FAIL    go-testing/calc 0.575s
```

Testeando el error con Testify

Si detallamos la salida de la consola en la ejecución con testify, se evidencia que testify ofrece una forma más clara y precisa de leer los resultados del test. Esto también es una ventaja de la implementación de testify.

output

```
--- FAIL: TestSumar (0.00s)
    calculadora_test.go:19:
                Error Trace:    calculadora_test.go:19
                Error:          Not equal:
                                expected: 8
                                actual  : -2
                Test:           TestSumar
                Messages:       deben ser iguales
FAIL
exit status 1
FAIL    go-testing/calc 0.575s
```

Comparando el código

Hasta este punto vimos dos formas de implementar un test unitario, que se diferencian en el uso de testify.

Sin complementos

```
func TestSumar(t *testing.T) {
    // Se inicializan los datos a usar en el test (input/output)
    num1 := 3
    num2 := 5
    resultadoEsperado := 8

    // Se ejecuta el test
    resultado := Sumar(num1, num2)

    // Se validan los resultados
    if resultado != resultadoEsperado {
        t.Errorf("Funcion suma() arrojo el resultado = %v, pero el
esperado es  %v", resultado, resultadoEsperado)
    }
}
```

Con Testify

```
func TestSumar(t *testing.T) {
    // Se inicializan los datos a usar en el test (input/output)
    num1 := 3
    num2 := 5
    resultadoEsperado := 8

    // Se ejecuta el test
    resultado := Sumar(num1, num2)

    // Se validan los resultados
    assert.Equal(t, resultadoEsperado, resultado, "deben ser iguales")

}
```

PANICS DURANTE TESTS

GO TESTING

Panic durante un test

Panic son aquellas alertas que se generan para indicar que algo salió mal inesperadamente. Evidencia un fallo que no debería ocurrir durante la ejecución o funcionamiento de un proceso, y para el cual, no se diseñó una rutina que maneje adecuadamente dicha excepción.

Si se genera un Panic durante la ejecución de pruebas, el test finaliza de forma abrupta, pero es registrado con status “Failed”.

Panic durante un test

Hagamos la prueba con una nueva función llamada Dividir()

{}

```go
package calculadora

// Función que recibe dos enteros y retorna la suma resultante
func Sumar(num1, num2 int) int {
    // Esta función ahora devuelve un resultado INCORRECTO
    return num1 - num2
}

// Función que recibe dos enteros y retorna la resta o diferencia resultante
func Restar(num1, num2 int) int {
    return num1 - num2
}

// Función que recibe dos enteros (numerador y denominador) y retorna la división resultante
func Dividir(num, den int) int {
    return num / den
}
```

Panic durante un test

Diseñemos el test unitario de la función Dividir(). En este test la validación para determinar que el test está OK, es simplemente que la función Dividir retorne un objeto distinto de Nil. Pero la variable num2 la inicializamos en 0 para forzar el error.

{}

```
func TestDividir(t *testing.T) {
    // Se inicializan los datos a usar en el test (input/output)
    num1 := 3
    num2 := 0

    // Se ejecuta el test
    resultado := Dividir(num1, num2)

    // Se validan los resultados aprovechando testify
    assert.NotNil(t, resultado)

}
```

Panic durante un test

$

```
go test
```

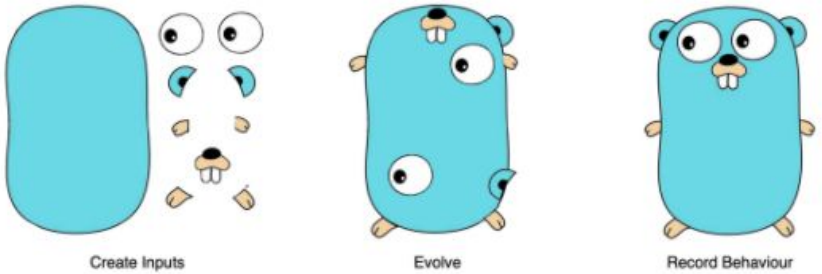
output

```
--- FAIL: TestDividir (0.00s)
panic: runtime error: integer divide by zero [recovered]
        panic: runtime error: integer divide by zero

goroutine 7 [running]:
testing.tRunner.func1.1(0xc8aac0, 0xe0bde0)
        c:/go/src/testing/testing.go:1072 +0x310
testing.tRunner.func1(0xc000037380)
        c:/go/src/testing/testing.go:1075 +0x43a
panic(0xc8aac0, 0xe0bde0)
        c:/go/src/runtime/panic.go:969 +0x1c7
go-testing/calc.Dividir(...)
        C:/Users/A308071/go/src/go-testing/calc/calculadora.go:15
go-testing/calc.TestDividir(0xc000037380)
        C:/Users/A308071/go/src/go-testing/calc/calculadora_test.go:29 +0x12
testing.tRunner(0xc000037380, 0xcd16e0)
        c:/go/src/testing/testing.go:1123 +0xef
created by testing.(*T).Run
        c:/go/src/testing/testing.go:1168 +0x2b3
exit status 2
FAIL    go-testing/calc 0.522s
```

Panic durante un test

Se evidencia que ante cualquier fallo inesperado que interrumpe el test, el status será “Failed”.



Manejo de errores

El manejo de errores es una característica esencial del código sólido.

Todo software contiene o contendrá en algún punto errores. Para el manejo de estos, algunos lenguajes utilizan excepciones, sin embargo Go las funciones y métodos admiten múltiples valores de retorno y, por convención, esta capacidad se usa comúnmente para devolver el resultado de la función junto con una variable de error.

Go resuelve el problema de informar a los programadores cuando las cosas han salido mal y se reserva el panic por lo verdaderamente excepcional.

Gracias.

IT BOARDING

BOOTCAMP

Autor: Nelber Mora
Email: nelber.mora@digitalhouse.com
Última fecha de actualización: 06-07-21

IT BOARDING
BOOTCAMP