GO

BOOTCAMP

Clase en vivo

//Go Bases

IT BOARDING

BOOTCAMP

Objetivos de esta clase

- Conocer cómo crear un test en Golang.
- Aprender a diseñar tests unitarios.
- Conocer las librerías nativas (standard libraries) usadas para ejecutar los test y sus validaciones.
- Entender la importancia de los tests.

Índice

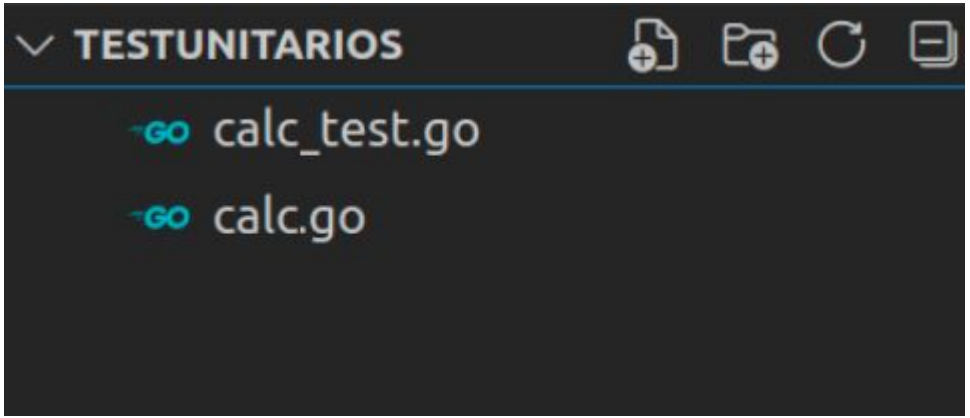
IT BOARDING
BOOTCAMP

Repaso

Convenciones en la escritura de los test

La primera convención en la escritura de test es escribir el archivo con el código de los tests en el mismo directorio donde se encuentra el código que se está probando.

Los archivos de test utilizan el nombre del archivo a prueba con el sufijo _test. De forma que el archivo calc.go tiene a su vez un archivo de test calc_test.go y se localiza en el mismo directorio.



Convenciones en la escritura de los test

La segunda convención al escribir pruebas es que las funciones deben empezar con la palabra Test.

{}

```
package calculator

import "testing"

func TestAddition(t *testing.T) {
    if Add(2, 2) != 4 {
        t.Fatal("The addition of 2 + 2 must equal 4")
    }
}
```

2 | Testing Nativo

IT BOARDING

BOOTCAMP

// 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 calculator

// Addition is a function that receives two integers and returns the resulting sum.
func Addition(num1, num2 int) int {
    return num1 + num2
}

// Subtraction is a function that receives two integers and returns the difference.
func Substraction(num1, num2 int) int {
    return num1 - num2
}
```

Nuestro primer test unitario.

{}

```go
package calculator

import "testing"

func TestAddition(t *testing.T) {
    num1 := 3
    num2 := 5
    expectedResult:= 8


    result := Addition(num1, num2)


    if result != expectedResult{
        t.Errorf("Addition() function gave the result = %v, but the expected result is %v", result, expectedResult)
    }
}
```

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 **calc.go** y **calc_test.go** pertenecen al mismo package “calculadora”. Esto permite acceder directamente a las funciones testeadas.

- El nombre de cada función de prueba inicia con el nombre “Test”. Esto permite que la función puede 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.

{}

```go
package calculator

// Addition is a function that receives two integers and returns the resulting sum.
func Addition(num1, num2 int) int {
    return num1 + num2
}

// Subtraction is a function that receives two integers and returns the difference.
func Substraction(num1, num2 int) int {
    return num1 - num2
}
```

Testeando el error

```
$ go test
```

output

```
--- FAIL: TestAddition(0.00s)
    calc_test.go:17: Addition() function gave the
result = -2, but the expected result is 8
FAIL
exit status 1
FAIL    go-testing/calc 0.635sPASS
ok
go-testing/calc 1.776s
```

A codear...

3 | Testify

IT BOARDING

BOOTCAMP

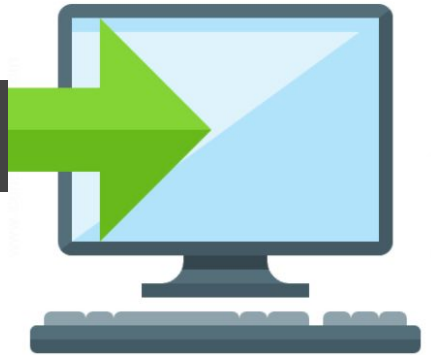
// Package “testify”

Es un paquete que facilita el desarrollo y la implementación de los test.
Su uso es ventajoso, porque entre otras cosas, nos ayuda a mejorar la calidad del código de pruebas, haciéndolo mucho más legible y entendible.

Instalando Testify (require)

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/require
```



Implementando Testify

Sobre el mismo file en el que venimos trabajando “calc_test.go” implementamos testify de la siguiente manera:

{}

```go
package calculator

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

func TestAddition(t *testing.T) {
    num1 := 3
    num2 := 5
    expectedResult:= 8

    obtainedResult:= Addition(num1, num2)


    require.Equal(t, expectedResult, obtainedResult)
}
```

Entendiendo Testify

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

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

// Validar desigualdad -> require.NotEqual(t, 123, 456, "no deberían ser iguales")

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

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

Ejecutando con testify

Si tomamos nuestro código original, con la función Addition() 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 **Addition()** 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 tomen 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 calculator

// Addition is a function that receives two integers and returns the
resulting sum.
func Addition(num1, num2 int) int {
    return num1 + num2
}

// Subtraction is a function that receives two integers and returns the
difference.
func Substraction(num1, num2 int) int {
    return num1 - num2
}
```

Testeando el error con Testify

```
$ go test
```

output

```
--- FAIL: TestSumar (0.00s)
    calc_test.go:19:
                Error Trace:    calc_test.go:19
                Error:          Not equal:
                                expected: 8
                                actual  : -2
                Test:           TestSAddition
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)
    calc_test.go:19:
                Error Trace:    calc_test.go:19
                Error:          Not equal:
                                expected: 8
                                actual  : -2
                Test:           TestSumar
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 TestAddition(t *testing.T) {
    num1 := 3
    num2 := 5
    expectedResult:= 8


    result := Addition(num1, num2)

    if result != expectedResult{
        t.Errorf("Addition() function gave the result = %v, but the
expected result is %v", result, expectedResult)
    }
}
```

Con Testify

```
func TestAddition(t *testing.T) {
    num1 := 3
    num2 := 5
    expectedResult:= 8


    result := Addition(num1, num2)

    require.Equal(t, expectedResult, result)
}
```

A codear...

Conclusiones

Hoy aprendimos a testear de manera unitaria nuestras porciones de código.

Aprendimos también el uso de testing nativo como el uso de la librería **“testify/require”** que nos permite distintas herramientas para mejorar nuestros Tests y organizarlos.

Recuerden que esta clase fue un primer acercamiento a Testing, a lo largo del Bootcamp seguiremos profundizando en contenidos de los test.

Gracias.

IT BOARDING

BOOTCAMP