



INTRODUCCIÓN A LA CLASE

GO TESTING

Objetivos de esta clase

- Entender el concepto de **Linter**.
- Usar ***golangci-lint***.
- Entender los distintos conceptos de **Coverage**.
- Generar el *informe* de Coverage.
- Entender ***Go Benchmark***.





LINTER

GO TESTING

// Linter

“Un Linter es una herramienta para realizar análisis automático y estático del código. Permite detectar de forma temprana errores y posibles malas prácticas.”

IT BOARDING

BOOTCAMP

Características

- Ayuda a seguir las buenas prácticas del lenguaje.
- Permite detectar errores de forma temprana.
- Mejora la calidad del código.



Go linters

- Staticcheck (<https://github.com/dominikh/go-tools>)
- golint ([golang/lint: \[mirror\] This is a linter for Go source code](#)) * deprecated
- **golangci-lint** (<https://github.com/golangci/golangci-lint>)
- govet ([vet - The Go Programming Language \(golang.org\)](#))
- gofmt ([gofmt - The Go Programming Language \(golang.org\)](#))
- goimports ([goimports · pkg.go.dev](#))
- errcheck ([kisielk/errcheck: errcheck checks that you checked errors. \(github.com\)](#))
- y muchos otros más...



El linter golint fue deprecado y l@s desarrollador@s del lenguaje Go recomiendan usar Staticcheck

☰ README.md

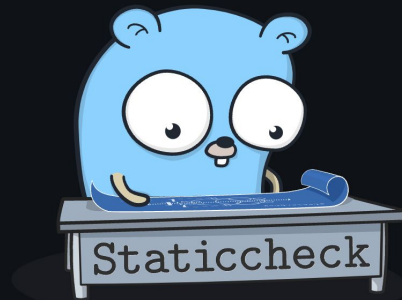
NOTE: Golint is [deprecated and frozen](#). There's no drop-in replacement for it, but tools such as [Staticcheck](#) and `go vet` should be used instead.

Golint is a linter for Go source code.

`GO` reference build passing

A partir de Go 1.17, la forma más sencilla de instalar **Staticcheck** es ejecutando:

```
go install honnef.co/go/tools/cmd/staticcheck@latest
```



The advanced Go linter

Staticcheck is a state of the art linter for the [Go programming language](#). Using static analysis, it finds bugs and performance issues, offers simplifications, and enforces style rules.

Financial support by [private and corporate sponsors](#) guarantees the tool's continued development. Please [become a sponsor](#) if you or your company rely on Staticcheck.

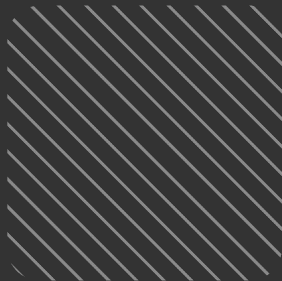


golangci-lint

// Tool

IT BOARDING

BOOTCAMP



// golangci-lint

“Es un Linter Runner para Go, combina distintos linters de Go para mejorar la calidad del código.”

IT BOARDING

BOOTCAMP

Características

- Combina muchos de los Linter Go más conocidos.
- Super rápido. Ya que ejecuta los análisis en paralelo.
- Fácil de integrar y configurar.
- Sencillo de usar.



Instalar golangci-lint

Para linux y windows:

\$

```
curl -sSfL https://raw.githubusercontent.com/golangci/golangci-lint/master/install.sh | sh -s --  
-b $(go env GOPATH)/bin v1.41.1
```

Para macOS:

\$

```
brew install golangci-lint  
brew upgrade golangci-lint
```

Con docker:

\$

```
docker run --rm -v $(pwd):/app -w /app golangci/golangci-lint:v1.41.1  
golangci-lint run -v
```



Validar versión y ejecutar golangci-lint

Para validar la versión lo hacemos con el siguiente comando:

```
$ golangci-lint --version
```

Para correr el linter lo hacemos de la siguiente manera parados en la carpeta raíz del proyecto.

```
$ golangci-lint run
```



¿Donde esta el error?

{}

```
func HandleHealth(w http.ResponseWriter, _ *http.Request) {  
    js, err := json.Marshal(map[string]interface{}{  
        "name": "DH-Service",  
        "info": "It is ok",  
    })  
    if err != nil {  
        http.Error(w, err.Error(), http.StatusInternalServerError)  
        return  
    }  
  
    w.Header().Set("Content-Type", "application/json")  
    w.WriteHeader(http.StatusOK)  
    w.Write(js)  
}
```



Usar golangci-lint

Parado en la raíz del proyecto ejecutar:

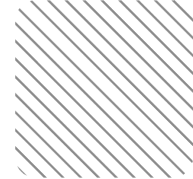
```
$ golangci-lint run
```

Se está ignorando un posible error:

```
$ example.go:21:9: Error return value of `w.Write` is not checked  
(errcheck)  
    w.Write(js)
```



Chequear los errores



```
{}
```

```
func HandleHealth(w http.ResponseWriter, _ *http.Request) {  
    js, err := json.Marshal(map[string]interface{}{  
        "name": "DH-Service",  
        "info": "It is ok",  
    })  
    if err != nil {  
        http.Error(w, err.Error(), http.StatusInternalServerError)  
        return  
    }  
  
    w.Header().Set("Content-Type", "application/json")  
    w.WriteHeader(http.StatusOK)  
    _, err = w.Write(js)  
    if err != nil {  
        http.Error(w, err.Error(), http.StatusInternalServerError)  
    }  
}
```





COVERAGE

GO TESTING

// Code Coverage

“Es una métrica que nos permite saber cuánto del código fuente de un software fue testeado.”

IT BOARDING

BOOTCAMP

// Coverage Report

“Es un reporte que además del Code Coverage también brinda información para saber qué partes del software fueron cubiertas por las pruebas.”

IT BOARDING

BOOTCAMP

Características

- Brinda una medida de la calidad del software.
- Ofrece un informe que permite identificar las partes del software que no fueron testeados.
- Podría ayudar a detectar código innecesario en el software, ya que no se ejecuta.



// ¿Por qué es importante el Coverage?

“Permite saber el grado de calidad del software que estamos construyendo.”

IT BOARDING

BOOTCAMP

¿Qué cobertura tiene?

Función Type Number:

{}

```
func TypeNumber(a int) string {  
    switch {  
    case a < 0:  
        return "negative"  
    case a > 0:  
        return "positive"  
    default:  
        return "zero"  
    }  
}
```



¿Qué cobertura tiene?

Unit Test de la función Type Number, tiene dos casos de test:

```
type Test struct {  
    in  int  
    out string  
}  
  
var tests = []Test{ {-1, "negative"}, {5, "positive"} }  
  
{}  
func TestTypeNumber(t *testing.T) {  
    for i, test := range tests {  
        size := TypeNumber(test.in)  
        if size != test.out {  
            t.Errorf("#%d: Size(%d)=%s; want %s", i, test.in, size, test.out)  
        }  
    }  
}
```



Generar el code coverage

Parado en la raíz del proyecto ejecutar:

```
$ go test -cover ./...
```

El 75% de nuestro código está cubierto por tests:

```
$ ok      poc-golang/go-test    0.289s  coverage: 75.0% of  
statements
```



Generar el coverage report

Generar el coverage report, ejecutar:

```
$ go test -cover -coverprofile=coverage.out ./...
```

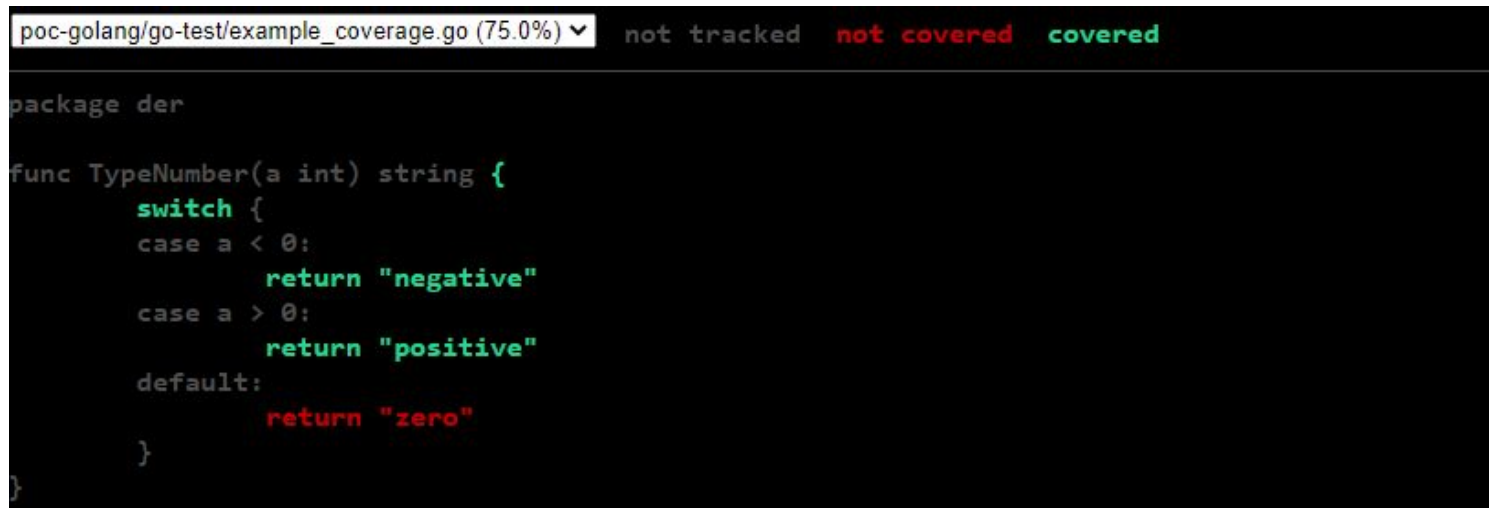
Mostrar en el navegador el coverage report generado, ejecutar:

```
$ go tool cover -html=coverage.out
```



Generar el coverage report

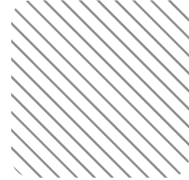
En rojo se pueden ver las líneas de código que no tienen cobertura. En verde las líneas cubiertas por test:



```
poc-golang/go-test/example_coverage.go (75.0%) ▼ not tracked not covered covered  
  
package der  
  
func TypeNumber(a int) string {  
    switch {  
    case a < 0:  
        return "negative"  
    case a > 0:  
        return "positive"  
    default:  
        return "zero"  
    }  
}
```



¿Como hacer para cumplir con la cobertura?



- No se trata de alcanzar un porcentaje de cobertura. El objetivo es escribir Tests que aseguren el cumplimiento de las funcionalidades y aporten a la calidad del software.
- El grado de cobertura depende de una serie de factores, cada equipo debería conocer esos factores y definir el grado de cobertura ideal.
- Puede existir un código malo con el 100% de cobertura. ***Las métricas no hacen un buen código.***





BENCHMARK

GO TESTING

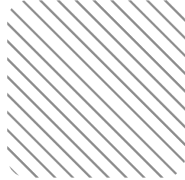
// Benchmark

“Es un tipo de test que permite examinar y poner a prueba la eficiencia del software. Go provee las herramientas nativas para poder realizar este tipo de pruebas.”

IT BOARDING

BOOTCAMP

Consideraciones importantes



- El archivo que contiene al benchmark test debe tener el sufijo “**_test**”, ejemplo **checksum_test.go**.
- El nombre de cada función de prueba debe iniciar con el nombre “**Benchmark**”. Esto permite que la función pueda ser reconocida como un benchmark test.
- El parámetro de la función de prueba debe ser “**b *testing.B**”.
- No es necesario chequear el resultado, ya que ***el objetivo es validar la performance***.



Miembros de testing.B

- “**b.N**” es una variable que define la cantidad de iteraciones, Go es el encargado de definir el valor de esta variable.
- “**b.StartTimer, b.StopTimer, b.ResetTimer**” son funciones que nos permiten poder manipular el contador de tiempo.
- “**b.RunParallel**” es una función que nos permite paralelizar el test.



Nuestro primer benchmark test

Para el primer benchmark test, se deben examinar dos algoritmos para generar checksum implementados por Go en el paquete “crypto”:

```
{  
    const Size = 32  
  
    const Size224 = 28  
  
    // Sum returns the SHA-1 checksum of the data.  
    func Sum(data []byte) [Size]byte  
  
    // Sum224 returns the SHA224 checksum of the data.  
    func Sum224(data []byte) (sum224 [Size224]byte)
```



Nuestro primer benchmark test

Escribimos un nuevo archivo llamado “checksum_test.go”. Se crean 2 benchmark test para comparar el tiempo de procesamiento de cada algoritmo:

```
{  
func BenchmarkSum256(b *testing.B) {  
    data := []byte("Digital House impulsando la transformacion digital")  
    for i := 0; i < b.N; i++ {  
        sha256.Sum256(data)  
    }  
}  
func BenchmarkSum(b *testing.B) {  
    data := []byte("Digital House impulsando la transformacion digital")  
    for i := 0; i < b.N; i++ {  
        sha1.Sum(data)  
    }  
}
```



Nuestro primer benchmark test

Ejecutar todos los test incluyendo los benchmark test:

```
$ go test -bench .
```

Notar que el algoritmo sha1.Sum es más rápido, ya que tarda 129 ns por iteración:

output	BenchmarkSum256-8	6316264	182 ns/op
	BenchmarkSum-8	8954107	129 ns/op





Gracias.

IT BOARDING

BOOTCAMP





Autor: Omar Barra

Email: omar.barra@digitalhouse.com

Última fecha de actualización: 09-07-21

IT BOARDING

BOOTCAMP

