INTRODUCCIÓN A LA CLASE

GO TESTING

Objetivos de esta clase

- Comprender el concepto de Functional Testing.
- Realizar nuestro primer Functional Testing.
- Comprender el concepto de TDD.
- Realizar nuestro primer TDD.
- Comprender el concepto de BDD.

FUNCTIONAL TESTING

GO TESTING

// Functional Testing

Es un tipo de test de caja negra que tiene como objetivo probar un requerimiento funcional específico del software.

IT BOARDING

BOOTCAMP

Características

- El objetivo es probar un requerimiento funcional concreto, ejemplo alta de producto.
- Para realizar estos tests es necesario que se integren múltiples componentes.
- Son más costosos de realizar, ya que es necesario que el sistema completo esté operativo.
- Son lentos a comparación de los unit tests.

httptest

//Package

IT BOARDING

BOOTCAMP

// httptest

Es un paquete de testing de Go que nos permite construir Functional Tests, End to End Tests e Integration Tests. Una de las ventajas es que puede ser usado junto con librerías nativas de GO como con librerías de terceros (Gin).

httptest.NewRequest

Sirve para generar un Request, se puede definir el método HTTP, el body y el header.

```
{}
req := httptest.NewRequest("GET", "http://example.com/foo", nil)
```

httptest.NewRecorder

Es básicamente un Response. Se pasa en el handler del Server y con este se puede obtener la respuesta HTTP después de la ejecución.

{}

```
var response *httptest.ResponseRecorder = httptest.NewRecorder()
```

¿Cómo lo integramos con Gin?

Después de crear el server Gin y configurar las rutas, se puede usar el método “ServerHTTP” para indicar que un determinado Request sea atendido y el resultado sea devuelto en el Response indicado.

{}

```
func (engine *Engine) ServeHTTP(w http.ResponseWriter, req *http.Request)
```

Realizando nuestro primer Test Funcional

Elegir dos funcionalidades del proyecto Go-Web del BOOTCAMP

Alta de producto:

```
{}  func (c *Product) Store() gin.HandlerFunc {
```

Obtener todos los productos:

```
{}  func (c *Product) GetAll() gin.HandlerFunc {
```

Realizando nuestro primer Test Funcional

Primeros pasos:

- Crear un nuevo package, el nombre es indistinto, en este ejemplo ”test”.
- Crear el archivo products_test.go.

Realizando nuestro primer Test Funcional

Función para crear el Server y definir las Rutas:

{}

```
package test
func createServer() *gin.Engine {
    _ = os.Setenv("TOKEN", "123456")
    db := store.New(store.FileType, "./products.json")
    repo := products.NewRepository(db)
    service := products.NewService(repo)
    p := handler.NewProduct(service)
    r := gin.Default()

    pr := r.Group("/products")
    pr.POST("/", p.Store())
    pr.GET("/", p.GetAll())
    return r
}
```

Realizando nuestro primer Test Funcional

Función para generar el Request y Response según nuestras necesidades. Como parámetro se le puede pasar el metodo HTTP, la URL y el body de forma opcional. En el header se agrega el token y el content type.

{}

```go
package test

func createRequestTest(method string, url string, body string)
(*http.Request, *httptest.ResponseRecorder) {
    req := httptest.NewRequest(method, url, bytes.NewBuffer([]byte(body)))
    req.Header.Add("Content-Type", "application/json")
    req.Header.Add("token", "123456")

    return req, httptest.NewRecorder()
}
```

Realizando nuestro primer Test Funcional

Se obtienen todos los productos y se valida la respuesta.

{}

```go
package test
func Test_GetProduct_OK(t *testing.T) {
    // crear el Server y definir las Rutas
    r := createServer()
    // crear Request del tipo GET y Response para obtener el resultado
    req, rr := createRequestTest(http.MethodGet, "/products/", "")

    // indicar al servidor que pueda atender la solicitud
    r.ServeHTTP(rr, req)

    assert.Equal(t, 200, rr.Code)
    err := json.Unmarshal(rr.Body.Bytes(), &objRes)
    assert.Nil(t, err)
    assert.True(t, len(objRes.Data) > 0)
}
```

Realizando nuestro primer Test Funcional

Se da de alta un producto y se valida la creación exitosa.

{}

```
package test
func Test_SaveProduct_OK(t *testing.T) {
    // crear el Server y definir las Rutas
    r := createServer()
    // crear Request del tipo POST y Response para obtener el resultado
    req, rr := createRequestTest(http.MethodPost, "/products/", `{
        "nombre": "Tester","tipo": "Funcional","cantidad": 10,"precio": 99.99
    }`)

    // indicar al servidor que pueda atender la solicitud
    r.ServeHTTP(rr, req)

    assert.Equal(t, 200, rr.Code)
}
```

TEST DRIVEN DEVELOPMENT

GO TESTING

// Test-Driven Development (TDD)

Es una técnica para el desarrollo de software que tiene como objetivo entender los casos de uso, escribir los tests y finalmente de forma iterativa implementar la solución.

Características

- La filosofía es primero escribir el test y después el código. De esta forma los tests dirigen el desarrollo.
- Toda línea de código debe tener un unit test asociado.
- Con TDD se puede alcanzar una cobertura del 100% ya que cada línea de código tiene un unit test asociado.
- Promueve el refactor continuo.

El ciclo Red-Green-Refactor

Paso 1: Elegir la funcionalidad a desarrollar y analizar los casos de prueba.

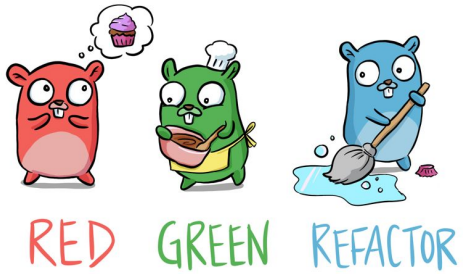
Paso 2: Escribir un unit test para el caso de prueba.

Paso 3: Escribir el mínimo código posible para que el Test pase.

Paso 4: Ejecutar todos los unit tests. Todos deberían pasar.

Paso 5: Refactor, mejorar el código.

Paso 6: Repetir paso 2 y 5 para cada caso de prueba.



Pasos TDD [1/6]

Elegir la funcionalidad a desarrollar y analizar los casos de prueba:
Funcionalidad: Calcular el factorial de un número. El factorial de un número N es la multiplicación de los enteros positivos hasta llegar N. La fórmula es N! = N*(N-1)!

Casos de prueba:

A. 0 ! = 1
B. 5 ! = 5 x 4 x 3 x 2 x 1= 120

Pasos TDD [2/6]

Escribir un unit test para un caso de prueba: **Caso de prueba A** 0! = 1

{}

```go
func TestFactorial(t *testing.T) {
    tests := []struct {
        arg  int
        want int
    }{{0, 1}}

    for i, d := range tests {
        got := factorial(d.arg)
        if got != d.want {
            t.Errorf("Test[%d]: factorial(%d) returned %d, want %d",
                i, d.arg, got, d.want)
        }
    }
}
```

Pasos TDD [3/6]

Escribir el mínimo código posible para que el test pase:

{}

```
func factorial(number int) int {
    if number == 0 {
        return 1
    }

    return 0
}
```

Pasos TDD [4/6]

Ejecutar todos los unit tests:

output

```
go test -v -run TestFactorial
=== RUN   TestFactorial
--- PASS: TestFactorial (0.00s)
PASS
ok      poc-golang/go-test      0.341s
```

Pasos TDD [5/6]

Refactor, mejorar el código: en este momento no hay nada que mejorar.

Pasos TDD [6/6]

Repetir paso 2 y 5 para cada caso de prueba:

Pasos TDD [2/6]

Escribir un unit test para un caso de prueba: **Caso de prueba B** 5 ! = 120

{}

```go
func TestFactorial(t *testing.T) {
    tests := []struct {
        arg  int
        want int
    }{{0, 1}, {5, 120}}

    for i, d := range tests {
        got := factorial(d.arg)
        if got != d.want {
            t.Errorf("Test[%d]: factorial(%d) returned %d, want %d",
                i, d.arg, got, d.want)
        }
    }
}
```

Pasos TDD [2/6]

Seguimos en el paso 2, vamos a ejecutar el nuevo caso de prueba para ver cómo se comporta nuestro algoritmo:

output

```
go test -v -run TestFactorial
=== RUN   TestFactorial
    factorial_test.go:14: Test[1]: factorial(5) returned 0, want 120
--- FAIL: TestFactorial (0.00s)
FAIL
```

Evidentemente falló, es hora de implementar una solución más robusta.

Pasos TDD [3/6]

Escribir el mínimo código posible para que el test pase:

{}

```
func factorial(number int) int {
    if number == 0 {
        return 1
    }

    return number * factorial(number-1)
}
```

Pasos TDD [4/6]

Ejecutar todos los unit tests:

output

```
go test -v -run TestFactorial
=== RUN   TestFactorial
--- PASS: TestFactorial (0.00s)
PASS
ok      poc-golang/go-test      0.306s
```

Pasos TDD [5/6]

Refactor: una función recursiva podría ser ineficiente, vamos a cambiarlo por una iteración tradicional.

{}

```
func factorial(number int) int {
    if number == 0 {
        return 1
    }

    f := 1
    for i := 1; i <= number; i++ {
        f *= i
    }

    return f
}
```

Pasos TDD [5/6]

Después del refactor se valida que los tests sigan pasando satisfactoriamente.

output

```
go test -v -run TestFactorial
=== RUN   TestFactorial
--- PASS: TestFactorial (0.00s)
PASS
```

BEHAVIOR DRIVEN DEVELOPMENT

GO TESTING

// Behavior-Driven Development (BDD)

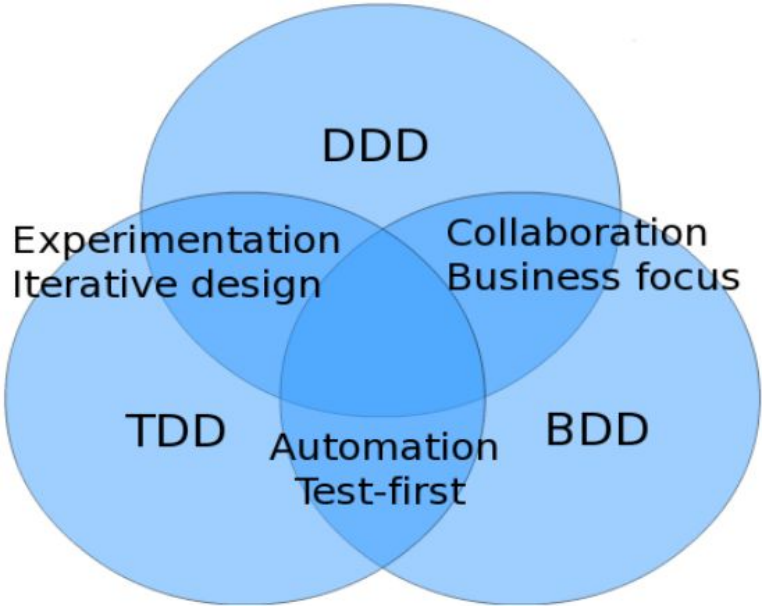
Es un proceso de desarrollo de software enfocado en el comportamiento del sistema que el usuario final espera experimentar.

Características

- BDD surge a partir de TDD, utiliza el principio de Red, Green and Refactor.
- Describe cómo el usuario final utiliza el software. Por este motivo se apoya en el dominio del experto.
- Uno de los objetivos es crear tests que sean entendibles tanto desde el punto de vista funcional como técnico.
- Los BDD tests están definidos por el formato GWT (Given, When, Then).

BDD vs TDD

TDD es una técnica utilizada por los programadores, en cambio BDD toma los conceptos de TDD y vas más allá definiendo un proceso de desarrollo de software en el cual pone especial énfasis en el negocio, el usuario final, el dominio del experto, la comunicación en el equipo y la automatización.



Given-When-Then

GWT es un formato para escribir y definir el criterio de aceptación de los tests en BDD.

- **Given**: Estado inicial del contexto y parámetros de entrada necesarios (Precondiciones).
- **When**: Ejecución de un proceso particular del sistema (Acción).
- **Then**: Los resultados y consecuencias después de la Acción del paso anterior (Resultados).

Given-When-Then

Un ejemplo sería:

Feature: Alta de estudiantes.

Scenario: El usuario solicita dar de alta a un estudiante.

Given La clase tiene 30 estudiantes.

When Se solicita dar de alta a un estudiante.

Then Se debería tener 31 estudiantes.

Frameworks BDD

- Cucumber es uno de los mejores frameworks BDD, te permite trabajar en diferentes tecnologías ejemplo java, python, etc. BDD Testing & Collaboration Tools for Teams | Cucumber

- Ginkgo es una muy buena opción para Golang developer. Ginkgo (onsi.github.io)

Los invitamos a completar la siguiente encuesta sobre el módulo Testing
¡Es muy muy importante para nosotros contar con su feedback!
Solamente les tomará unos minutos completarla :)

Link a la encuesta

Gracias.

IT BOARDING

BOOTCAMP

Autor: Omar Barra
Email: omar.barra@digitalhouse.com
Última fecha de actualización: 16-07-21

IT BOARDING
BOOTCAMP