



INTRODUCCIÓN A LA CLASE

GO TESTING

Objetivos de esta clase

- Conocer qué es un Test de Integración.
- Implementar Test de Integración con Go.
- Aplicar Test de Integración sobre el Proyecto Go Web.





TEST DE INTEGRACIÓN

GO TESTING

// ¿Qué es Test de Integración?

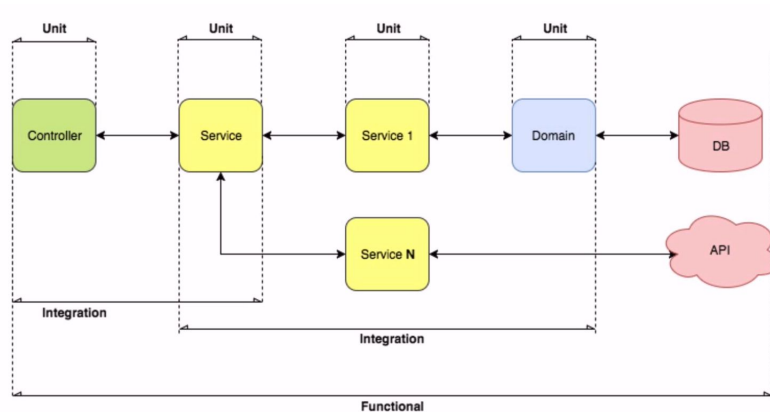
Son los test que comprueban el comportamiento entre dos módulos, packages o capas de la aplicación.

IT BOARDING

BOOTCAMP

Test de Integración

Una de las principales diferencias con el **Test Unitario**, es que en el **Test de Integración** se pone a prueba la comunicación e entre distintas unidades o bloques de código. Tienden a ser menos exhaustivos que un test unitario, pero son requeridos para validar que la interacción entre capas sea correcta.



Importancia del test de integración

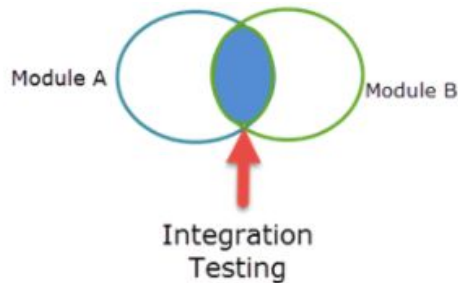
El **Test de Integración** es tan importante como el **Test Unitario**. Una aplicación podría tener **test unitarios** en todos los componentes, pero no podemos asegurar completamente la calidad del código si no se han aprobado las pruebas de Integración. Es posible que un componente funcione bien de forma unitaria, pero falle en la integración. Esto hace que sea tan importante asegurar la integración.



Objetivos del test de integración







Generalmente el Test de Integración se aplica para:

- Verificar que las interfaces estén correctamente diseñadas e implementadas.
- Prevenir fallos potenciales durante las siguientes etapas de Testing.
- Detectar vulnerabilidades de seguridad.
- Detectar errores de interacción.



Diseño del test de integración

Para diseñar un Test de Integración, también será necesario la aplicación de Doubles previamente vistos, de la misma forma que también será necesario disponer de los packages “testing” y “testify” para el diseño de los mismos. Por lo que Dummy, Stub, Spy, Mock y Fake seguirán siendo parte imprescindible de los tests que diseñemos.

	Unit Testing	Integration Testing
		
		

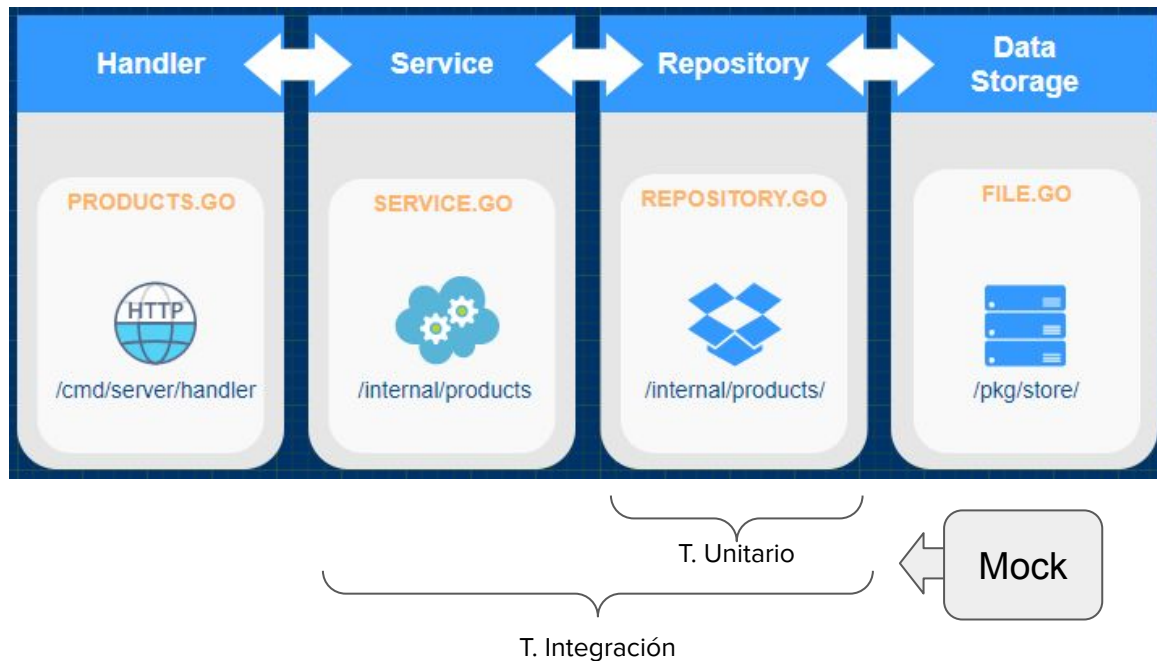


IMPLEMENTACIÓN

GO TESTING

Implementación

Vamos a implementar nuestro primer Test de Integración. Para esto haremos un Mock de **file.go** y probar unitariamente el repository, para luego si probar la integración de **Service con Repository**.



Implementación

En el archivo file.go que hace de emulador de DataStorage, también ofrece la forma de “Mockear” data específica, lo que nos permite diseñar el Stub que usaremos para los test unitarios y de integración.

{}

```
type FileStore struct {
    FileName string
    Mock *Mock
}
type Mock struct {
    Data []byte
    Err error
}
func (fs *FileStore) AddMock(mock *Mock) {
    fs.Mock = mock
}
func (fs *FileStore) ClearMock() {
    fs.Mock = nil
}
func (fs *FileStore) Read(data interface{}) error {
    if fs.Mock != nil {
        if fs.Mock.Err != nil {
            return fs.Mock.Err
        }
        return json.Unmarshal(fs.Mock.Data, data)
    }
    file, err := ioutil.ReadFile(fs.FileName)
    if err != nil {
        return err
    }
    return json.Unmarshal(file, data)
}
```



Implementación

En el archivo file.go que contiene la logica para guardar nuestros datos, este funciona a través un interfaz Store que realiza una lectura de un archivo .json como base de datos. Vamos a realizar un Mock de este para poder utilizar en test unitarios.

```
{}
```

```
type Store interface {  
    Read(data interface{}) error  
    Write(data interface{}) error  
}  
  
func NewStore(fileName string) Store {  
    return &fileStore{fileName}  
}  
  
type fileStore struct {  
    FilePath string  
}  
  
func (fs *fileStore) Write(data interface{}) error {  
    fileData, err := json.Marshal(data)  
    if err != nil {  
        return err  
    }  
    return os.WriteFile(fs.FilePath, fileData, 0644)  
}  
  
func (fs *fileStore) Read(data interface{}) error {  
    file, err := os.ReadFile(fs.FilePath)  
    if err != nil {  
        return err  
    }  
    return json.Unmarshal(file, &data)  
}
```



Mock de file.go

Esta es la implementación de un Mock de file.go, lo que se busca es imitar el comportamiento de Store real, pero escribiendo en una lista dentro de memoria únicamente. Adicionalmente tiene 2 atributos para forzar un caso de error en la lectura y uno en la escritura

{}

```
type MockStorage struct {
    dataMock []domain.Product
    errWrite string
    errRead  string
}

func (m *MockStorage) Read(data interface{}) error {
    if m.errRead != "" {
        return fmt.Errorf(m.errRead)
    }
    a := data.(*[]domain.Product)
    *a = m.dataMock
    return nil
}

func (m *MockStorage) Write(data interface{}) error {
    if m.errWrite != "" {
        return fmt.Errorf(m.errWrite)
    }
    a := data.(*[]domain.Product)
    m.dataMock = append(m.dataMock, a[len(a)-1])
    return nil
}
```





Test Unitario

GO TESTING

// ¿Qué es un unit Test?

“Es la forma de probar una parte pequeña del código, atomizada lo mayormente posible. Usualmente estos son por cada método o función.”

IT BOARDING

BOOTCAMP

Test Unitario de Repository

Se crea el archivo repository_test.go en el mismo folder en el que está repository.go.

Aquí se guarda el mock antes visto y se crea la función “TestGetAll”.

Para inicializar es necesario crear el mock. Primero se crea el conjunto de productos que deseamos usar como base de datos.

{}

```
func TestGetAll(t *testing.T) {  
    // arrange  
    database := []domain.Product{  
        {  
            ID: 1,  
            Name: "CellPhone",  
            Type: "Tech",  
            Count: 3,  
            Price: 250,  
        }, {  
            ID: 2,  
            Name: "Notebook",  
            Type: "Tech",  
            Count: 10,  
            Price: 1750.5,  
        }}  
    mockStorage := MockStorage{  
        dataMock: database,  
        errWrite: "",  
        errRead: "",  
    }  
    ... continua ...  
}
```



Test Unitario de Repository

Para probar el Repository, es necesario instanciarlo con el mock de Storage. Posteriormente se ejecuta el test y se valida que la información que devuelve el repository sea igual a la que definimos en el mock del Storage. Esto sucede solo si GetAll se ejecuta correctamente.

{ }

... continuación...

```
// act
repo := NewRepository(&mockStorage)
result, err := repo.GetAll()
// assert
assert.Nil(t, err)
assert.Equal(t, mockStorage.dataMock, result)
}
```



Test Unitario de Repository

Finalmente procedemos a la ejecución del test.

Ejecutaremos el comando con el flag “v” que significa verbose, para mostrar información detallada del test:

```
$ go test -v
```

```
output === RUN   TestGetAll
--- PASS: TestGetAll (0.00s)
PASS
ok
github.com/ncostamagna/meli-bootcamp/internal/products
0.200s
```





TEST DE INTEGRACIÓN

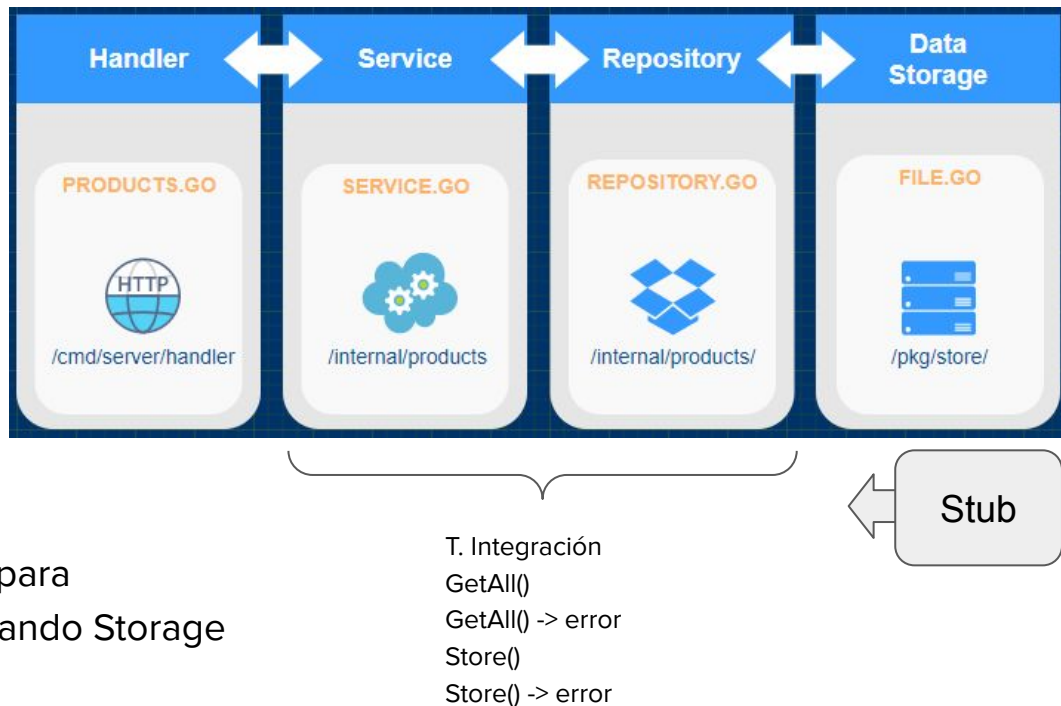
GO TESTING

Test de Integración

El test de integración se diseñará usando como objeto de prueba la capa “service” y su integración con “repository”.

De la misma forma que en el test anterior, usaremos un Mock de Storage.

Serán testeados GetAll y Store, para respuestas válidas y también cuando Storage devuelve error.





INTEGRACIÓN: GETALL

GO TESTING

Test de Integración

De forma análoga al mock que se diseñó para el test unitario, hacemos lo mismo para el test de integración entre el repository y el service.

{}

```
func TestServiceGetAll(t *testing.T) {
    input := []Product{
        {
            ID:      1,
            Name:    "CellPhone",
            Type:    "Tech",
            Count:  3,
            Price:  250,
        }, {
            ID:      2,
            Name:    "Notebook",
            Type:    "Tech",
            Count:  10,
            Price:  1750.5,
        },
    }
    dataJson, _ := json.Marshal(input)
    dbMock := store.Mock{
        Data: dataJson,
    }
    storeStub := store.FileStore{
        FileName: "",
        Mock:     &dbMock,
    }
    ... continúa...
```



Test de Integración

Para invocar la ejecución del test, es necesario instanciar el Service con el repositorio que contiene el stub del storage. Posteriormente, se ejecuta el test y se valida en este caso que los resultados sean exactamente igual a lo esperado y que el error sea nil.

```
{}
```

```
... continuación...
```

```
myRepo := NewRepository(&storeStub)
```

```
myService := NewService(myRepo)
```

```
result, err := myService.GetAll()
```

```
assert.Equal(t, input, result)
```

```
assert.Nil(t, err)
```

```
}
```



Test de Integración

De forma análoga al mock que se diseñó para el test unitario, hacemos lo mismo para el test de integración entre el repository y el service.

{}

```
func TestServiceIntegrationGetAll(t *testing.T) {  
    // arrange  
    database := []domain.Product{  
        {  
            ID: 1,  
            Name: "CellPhone",  
            Type: "Tech",  
            Count: 3,  
            Price: 250,  
        }, {  
            ID: 2,  
            Name: "Notebook",  
            Type: "Tech",  
            Count: 10,  
            Price: 1750.5,  
        }  
    }  
    mockStorage := MockStorage{  
        dataMock: database,  
        errWrite: "",  
        errRead: "",  
    }  
    ... continúa...
```



Test de Integración

Para invocar la ejecución del test, es necesario instanciar el Service con el repositorio que contiene el mock del storage.

Posteriormente, se ejecuta el test y se valida en este caso que los resultados sean exactamente igual a lo esperado y que el error sea nil.

En esta oportunidad validamos los resultados respecto a lo que retorno la llamada al service, que internamente llamó a repo.GetAll

{ }

... continuación...

```
// act
repo := NewRepository(&mockStorage)
service := NewService(repo)
result, err := service.GetAll()
// assert
assert.Nil(t, err)
assert.Equal(t, mockStorage.dataMock, result)
}
```



Test de Integración

Finalmente procedemos a la ejecución del test.

```
$ go test -v
```

output

```
=== RUN   TestGetAll  
--- PASS: TestGetAll (0.00s)  
PASS  
ok  
github.com/ncostamagna/meli-bootcamp/internal/products  
0.200s
```





INTEGRACIÓN: GETALL ERROR

GO TESTING

Test de Integración

Ahora vamos a probar la integración cuando la respuesta desde el Storage es errónea.

Para esto debemos usar otro Mock. Donde se establece que la data en el Storage es nil y el error es igual a “expectedError”.

```
{}
```

```
func TestServiceGetAllError(t *testing.T) {
    expectedError := errors.New("error for GetAll")
    dbMock := store.Mock{
        Err: expectedError,
    }
    storeStub := store.FileStore{
        FileName: "",
        Mock:     &dbMock,
    }
    myRepo := NewRepository(&storeStub)
    myService := NewService(myRepo)

    result, err := myService.GetAll()

    assert.Equal(t, expectedError, err)
    assert.Nil(t, result)
}
```



Test de Integración

Ahora vamos a probar la integración cuando la respuesta desde el Storage es errónea.

Para esto debemos usar otro Mock. Donde se establece que la data en el Storage es nil y el error es igual a “expectedError”.

```
{}
```

```
func TestServiceIntegrationGetAllFail(t *testing.T) {  
    // arrange  
    expectedError := errors.New("cant read database")  
    mockStorage := MockStorage{  
        dataMock: nil,  
        errWrite: "",  
        errRead: "cant read database",  
    }  
    // act  
    repo := NewRepository(&mockStorage)  
    service := NewService(repo)  
    result, err := service.GetAll()  
    // assert  
    assert.Equal(t, expectedError, err)  
    assert.Nil(t, result)  
}
```



Test de Integración

Finalmente procedemos a la ejecución del test.

```
$ go test -v
```

output

```
=== RUN   TestServiceGetAllError  
--- PASS: TestServiceGetAllError (0.00s)  
PASS  
ok
```





INTEGRACIÓN: STORE

GO TESTING

Test de Integración

En este test de integración se comprobará que desde el service se pueda almacenar información correctamente.

Para esto se define el Stub inicial vacío y se ejecuta el método Store. La respuesta debe retornar un producto con las mismas características y con ID = 1.

{}

```
func TestStore(t *testing.T) {  
    testProduct := Product{  
        Name: "CellPhone",  
        Type: "Tech",  
        Count: 3,  
        Price: 52.0,  
    }  
    dbMock := store.Mock{}
```

}

```
    storeStub := store.FileStore{  
        FileName: "",  
        Mock:      &dbMock,  
    }  
    myRepo := NewRepository(&storeStub)  
    myService := NewService(myRepo)  
    result, _ := myService.Store(testProduct.Name,  
                                testProduct.Type,  
                                testProduct.Count,  
                                testProduct.Price)  
    assert.Equal(t, testProduct.Name, result.Name)  
    assert.Equal(t, testProduct.Type, result.Type)  
    assert.Equal(t, testProduct.Price, result.Price)  
    assert.Equal(t, 1, result.ID)  
}
```



Test de Integración

En este test de integración se comprobará que desde el service se pueda almacenar información correctamente.

Para esto se define el Mock inicial vacío y se ejecuta el método Store. La respuesta debe retornar un producto con las mismas características y con ID = 1.

{}

```
func TestServiceIntegrationStore(t *testing.T) {  
    // arrange  
    newProduct := domain.Product{  
        ID:    0,  
        Name:  "Tablet",  
        Type:  "Tech",  
        Count: 5,  
        Price: 1050.99,  
    }  
    mockStorage := MockStorage{  
        dataMock: nil,  
        errWrite:  "",  
        errRead:   "",  
    }  
    // act  
    repo := NewRepository(&mockStorage)  
    service := NewService(repo)  
    result, err := service.Store(  
        newProduct.Name,  
        newProduct.Type,  
        newProduct.Count,  
        newProduct.Price)  
    // assert  
    assert.Nil(t, err)  
    assert.Equal(t, mockStorage.dataMock[0], result)  
    assert.Equal(t, mockStorage.dataMock[0].ID, 1)  
}
```



Test de Integración

Finalmente procedemos a la ejecución del test.

```
$ go test -v
```

output

```
=== RUN   TestStore  
--- PASS: TestStore (0.00s)  
PASS  
ok
```





INTEGRACIÓN: STORE ERROR

GO TESTING

Test de Integración

Con esta integración comprobaremos que si ocurre un error durante la escritura de FileStore, el service reciba del repositorio el error correcto y además que retorne un objeto vacío de Product.

{}

```
func TestStoreError(t *testing.T) {
    testProduct := Product{
        Name:  "CellPhone",
        Type:  "Tech",
        Count: 3,
        Price: 52.0,
    }
    expectedError := errors.New("error for Storage")
    dbMock := store.Mock{
        Err: expectedError,
    }
    storeStub := store.FileStore{
        FileName: "",
        Mock:     &dbMock,
    }
    myRepo := NewRepository(&storeStub)
    myService := NewService(myRepo)
    result, err := myService.Store(testProduct.Name,
                                   testProduct.Count,
                                   testProduct.Price)
    assert.Equal(t, expectedError, err)
    assert.Equal(t, Product{}, result)
}
```



Test de Integración

Con esta integración comprobaremos que si ocurre un error durante la escritura de Storage, el **service** reciba del **repositorio** el error correcto y además que retorne un objeto vacío de Product.

{}

```
func TestServiceIntegrationStoreFail(t *testing.T) {  
    // arrange  
    newProduct := domain.Product{  
        ID:      0,  
        Name:    "Tablet",  
        Type:    "Tech",  
        Count:   5,  
        Price:   1050.99,  
    }  
    writeErr := fmt.Errorf("cant write database")  
    expectedError := fmt.Errorf("error creating product: %w",  
writeErr)  
    mockStorage := MockStorage{  
        dataMock: nil,  
        errWrite:  "cant write database",  
        errRead:   "",  
    }  
    // act  
    repo := NewRepository(&mockStorage)  
    service := NewService(repo)  
    result, err := service.Store(  
        newProduct.Name,  
        newProduct.Type,  
        newProduct.Count,  
        newProduct.Price)  
    // assert  
    assert.Equal(t, expectedError, err)  
    assert.Equal(t, domain.Product{}, result)
```



Test de Integración

Finalmente procedemos a la ejecución del test.

```
$ go test -v
```

```
output === RUN   TestStoreError  
--- PASS: TestStoreError (0.00s)  
PASS  
ok
```



Observaciones Finales

A efectos del abordaje de esta clase, se hizo Test sobre un par de métodos de la integración, pero la cobertura (o coverage) debe ser amplia y procurar que se compruebe el comportamiento de todos los métodos y sus distintos flujos internos. La integración ejecutada alcanzó las capas Service/Repo/Db.





Gracias.

IT BOARDING

BOOTCAMP



Autor: Nelber Mora

Email: nelber.mora@digitalhouse.com

Última fecha de actualización: 08-07-21

IT BOARDING

BOOTCAMP

