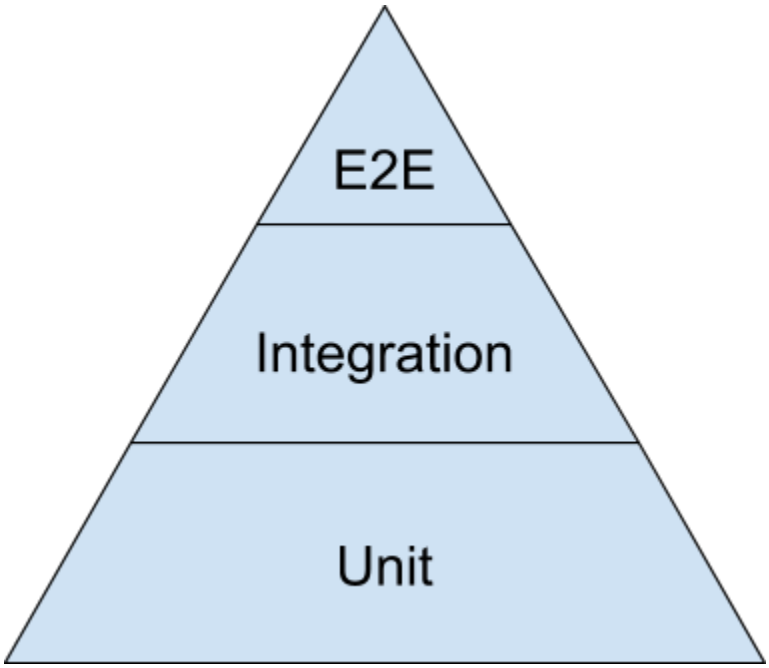
GO-06 01: Unit-тестирование в Go

Описание:

Этот модуль будет посвящен улучшению качества наших программ при помощи различных проверок (Тестирование / Бенчмаркинг / Профилирование), а начнем мы с самого важного из этих методов.
Тестирование
Тесты - это то, без чего нельзя представить себе работающий код. Мы пишем тесты, чтобы проверить, что наша программа делает то, что от нее ждут, а также не делает того, чего от нее не ждут.
Существуют разные типы тестов:

- Unit - тестирование кода программы без каких-либо зависимостей (баз данных или других сервисов).
- Integration - это тесты, в которых тестируется уже не только сам код, но и связка его с другими частями системы или инфраструктуры (БД, брокеры сообщений, сторонние сервисы и т.д.).
- E2E (End-to-End) - или их еще называют UI (User Interface) тесты, это тесты всей системы целиком. На данном уровне тестируются пользовательские сценарии. Например, покупка товара в магазине, а именно - полный процесс, от входа в личный кабинет до оплаты заказа по карте. Здесь уже тестируется не просто наш кусочек кода, а то, как этот кусочек кода гармонирует со всей остальной системой.

Для описания уровней тестов была даже придумана пирамида уровней. Выглядит она примерно так (в разных вариациях по-разному).



Мы же в этом модуле посмотрим, как в Golang писать Unit тесты, так как в большинстве случаев именно их мы и будем использовать в наших проектах.
Также, за редким исключением, мы можем писать еще и integration тесты, но, как правило, во многих компаниях эти тесты передают уже на уровень QA-специалистов.
Тесты в Go
Тесты в Golang чаще всего лежат рядом с файлами имплементации и имеют имя [file_name]_test.go.

```
|-package
|---- my_func_impl.go
|---- my_func_impl_test.go
```

Этот стандартный префикс _test нужен для того, чтобы утилита go test понимала, в каких файлах лежат тесты, а в какие файлы можно не заглядывать.
Сами тестовые программы выглядят практически также, как и обычные, за исключением того, что тестовые функции должны начинаться со слова Test, а дальше иметь название тестируемой функции. Обычно такие функции имеют вид:

```
package sum

import "testing"

func TestSum(t *testing.T) {
      if sum(1, 1) != 2 {
            t.Fatal("fail")
      }
}
```

Где t *testing.T - это единственный аргумент в тестовых функциях, а пакет testing - это пакет из стандартной библиотеки go, в котором собраны различные вспомогательные функции для тестов.
Метод t.Fatal("fail") при вызове дает утилите go test понять, что тест завершен неудачно.
Пакет testify
Единственное, чего недостает пакету testing, - это хороших функций проверок, а для тестов это очень важно. Например, допустим, что у нас есть функция:

```
func GetOrders(limit int) ([]Order, error) {
    res, err := httpClient.Operations.GetOrders(limit)
    if err != nil {
        return nil, err
    }

    return res.Payload.Orders, nil
}
```

Она возвращает количество ордеров, которое мы попросили, передав туда аргумент limit или ошибку в случае некорректного ответа от другого сервиса или неполадок с сетью.
Тест для такой функции будет выглядеть примерно вот так:

```
func TestGetOrder(t *testing.T) {
    orders, err := GetOrder(5)
    if err != nil {
        t.Errorf("fail to get orders: %w", err)
    }

    if len(orders) != 5 {
        t.Error("fail orders length")
    }

    if orders[0].ID == 1015 {
        t.Error("invalid order ID")
    }
}
```

И так может продолжаться и дальше, в зависимости от условий использования функций, краевых случаев и т.д. Плюс ко всему эта функция еще очень простая. Возможны и такие варианты, где функционал будет куда сложнее и от этого будут страдать тесты. Ведь нам надо описывать логику проверок, а это сложно читается, так как надо вчитываться в контекст и разбираться, как написан и работает тест, а не в том, что он тестирует.
Но есть отличное решение, как писать все эти проверки в более понятном, декларативном стиле. Это пакет testify.
Пакет testify расширяет стандартную библиотеку для тестирования и добавляет много нового и интересного функционала (assert/require, mock, suite и т.д.), который, в свою очередь, облегчает написание и понимание тестов.
После установки и добавления этого пакета в наш тест он сразу начинает выглядеть более понятно.

```
package orders

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

func TestGetOrder(t *testing.T) {
    req := require.New(t)
    orders, err := GetOrder(5)
    req.NoError(err)
```

```
    req.Len(orders, 5)
    req.NotEqual(1015, orders[0].ID)
}
```

Использование этого пакета считается неким best practices в наше время, так как позволяет делать тесты, которые сможет прочитать даже разработчик не на go.
Запуск тестов
Запуск тестов происходит через вызов команды go test.
go test [build/test flags] [packages] [build/test flags & test binary flags]
Важно отметить, что go test может быть запущен в разных режимах:

- В режиме локального каталога. В этом режиме тесты запускаются только в текущем каталоге, а также результаты тестов не кешируются. Для того чтобы запустить тесты в этом режиме, достаточно не указывать аргументы пакетов (т.е. не указывать, конкретно из каких пакетов запускать тесты).

Пример запуска в режиме локального каталога - go test -v.

- В режиме списка пакетов. В этом режиме пакеты указываются явно, а положительные результаты тестов кешируются (это сделано для того, чтобы не запускать лишний раз положительно прошедшие тесты, так как пакетов может быть много и каждый прогон тестов без кеширования будет занимать много времени). Важно отметить, что результаты будут кешироваться всегда, кроме случая, когда указаны флаги, которые не относятся к разряду кешируемых тестовых флагов -cpu, -list, -parallel, -run, -short и -v.
    - Запуск конкретного пакета - go test mypackage
    - Запуск тестов всех пакетов текущего каталога - go test ./...
    - Запуск тестов в режиме списка пакетов, без кеширования - go test ./... -count=1

При успешном прохождении тестов вы увидите примерно вот такую картину:

```
~# go test
PASS
ok    _/home/fanyshu/Projects/learn/go-test 0.002s
```

Будут указаны все пакеты, в которых были запущены тесты и, если все тесты пройдены, то напротив них будет стоять ok.
Если же тесты провалились, то мы увидим следующую картину:

```
~# go test
--- FAIL: TestSum (0.00s)
    test_test.go:7: fail
FAIL
exit status 1
FAIL  _/home/fanyshu/Projects/learn/go-test 0.001s
```

А при использовании testify вывод будет более информативным (в этом еще один плюс testify):

```
~# go test
--- FAIL: TestSum (0.00s)
    test_test.go:11:
            Error Trace:    test_test.go:11
            Error:          Not equal:
                            expected: 1
                            actual  : 2
            Test:           TestSum
FAIL
exit status 1
FAIL    _/home/fanyshu/Projects/learn/go-test 0.002s
```

Полезные ссылки:

- Golang book testing
- Godoc package testing
- Testify package
- Команды go: go test, тестировать пакеты

Задание:

1. Сделайте форк проекта module06 в группу golang_users_repos/<your_gitlab_id>.
2. Создайте в своем проекте module06 ветку module06_01.
3. Создайте рядом с файлом util.go файл util_test.go в каталоге ./internal/pkg/util.
4. В данном задании вам необходимо написать тесты для функции ReverseInt(x interface{}).
5. Напишите несколько тест-кейсов:
    - Что будет, если передать туда обычное число? Например, 123 ?
    - Что будет, если передать туда отрицательное число? Например, -649 ?
    - Что будет, если передать туда 0 ?
6. Придумайте и напишите несколько тест-кейсов самостоятельно.
7. Зафиксируйте изменения в репозитории (сделайте коммит с выполненными задачами).
8. Подключите библиотеку testify и перепишите тесты с использованием require/assert.
9. Запустите и удостоверьтесь, что тесты работают, как нужно.
10. В качестве ответа пришлите ссылку на merge request в ветку master вашего проекта ветки module06_01.