Testes em Go - Do básico ao avançado

2019

mercado livre

Quem sou eu?

- **Software Developer** @ Mercado Libre
- Formado pela Fatec Zona Leste em 2016
- 5 anos de experiência profissional
- GitHub/Insta: BrunoDM2943
- Linkedin

- Asserts em Go
- Mock em Go
- Testes de:
  - Repositório
  - Service
  - Controller
- Testes de Benchmark

#ENGINEERING

mercado livre

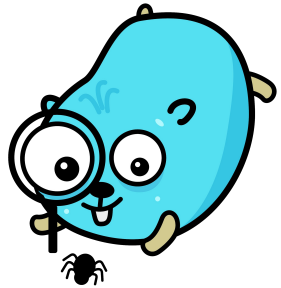
#ENGINEERING

Vamos começar?

mercado livre

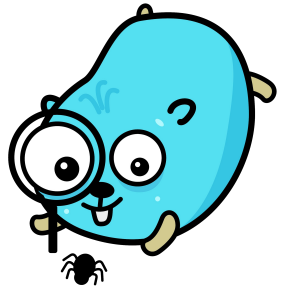
Asserts em Go

- Um assert nada mais que um comando utilizado para validar determinada condição.
- Diferente de um IF, ele é usado para garantir que a condição seja verdadeira, do contrário, a execução do programa é finalizada.
- No nosso cenário de testes, os asserts são usados para checar se uma condição posterior ao teste foi atingida



Asserts em Go

- A biblioteca nativa não fornece suporte para assert. Então para quem veio de Java, isso pode parecer uma dificuldade.
- A biblioteca mais famosa que temos para suportar essa falta é a testify
- Asserts disponíveis
  - assert.NotEqual
  - assert.Equal
  - assert.NotNil
  - assert.Nil



Asserts em Go

```go
package yours

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

func TestSomething(t *testing.T) {
  assert := assert.New(t)

  // assert equality
  assert.Equal(123, 123, "they should be equal")

  // assert inequality
  assert.NotEqual(123, 456, "they should not be equal")

  // assert for nil (good for errors)
  assert.Nil(object)

  // assert for not nil (good when you expect something)
  if assert.NotNil(object) {

    // now we know that object isn't nil, we are safe to make
    // further assertions without causing any errors
    assert.Equal("Something", object.Value)
  }
}
```

Mas e na vida real?

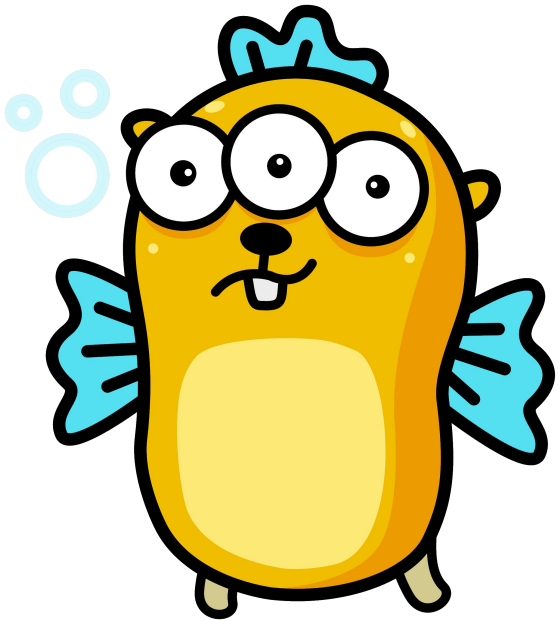
panic(err)

Mas e na vida real?

Problemas

- ➔ Dependências externas nos testes (como chamadas a APIs)
- ➔ Dependências de negócio para testar caminhos felizes
- ➔ Arquiteturas/camadas que não temos suporte para ambientes de testes (como uma base de dados)

Solução



Mock em Go

- ➔ gomock é uma das principais bibliotecas para isso.
- ➔ Para utilizar os Mocks em Go, precisamos desenvolver o código voltado a interfaces
- ➔ As interfaces em Go funcionam como uma definição de funções. Diferente de outras linguagens, não é necessário dizer no código que a nossa classe (ou aqui, struct) implementa a interface. A própria linguagem detecta isso.
- ➔ Para realizar o mock, anotamos em cima da definição da interface o comando que vai ser executado pelo generate, dizendo como vai ser gerado o mock e o seu destino.

Show me the code

Mock em Go

```
type Service interface {
    SaveAccount(person *domain.Account) error
    GetAccount(ID int64) (*domain.Account, error)
}
```

Mock em Go

```
→  account git:(master) go get github.com/golang/mock/gomock
go: finding github.com/golang/mock/gomock latest
→  account git:(master) go install github.com/golang/mock/mockgen
→  account git:(master) mockgen -source=./accountService.go -destination=./
mock/mock_accountService.go
→  account git:(master)
```

Mock em Go

```
//Service for account services
//go:generate mockgen -source=./accountService.go -
type Service interface {
    SaveAccount(person *domain.Account) error
    GetAccount(ID int64) (*domain.Account, error)
}
```

Testando um repo

- Dificilmente conseguimos testar a nossa camada de acesso a dados
- Se temos acesso a uma base de dados, vamos precisar de um ambiente para testes automatizados.
  - Isso é difícil de manter, porque precisamos pensar em scripts que mantenham a base em um estado para que todos os testes funcionem 100% a cada execução.

Testando um repo

- Se temos acesso a uma API externa, precisamos mockar às chamados e retornos. E isso não é algo fácil...
  - Nem todos os frameworks dão suporte para Mocks no RestClient, então o melhor a fazer, é mockar as nossas funções que fazem essa chamada.
  - Temos um restclient no Meli que é público e ajuda muito nisso =)

Testando um repo

```
func TestSaveAccountOnDBWihoutID(t *testing.T) {
    db, _ := gorm.Open("sqlite3", "../gotest.db")
    db.DropTable(&domain.Account{})
    db.AutoMigrate(&domain.Account{})
    repo := NewAccountRepo(db)
    defer db.Close()
    account := &domain.Account{
        User: "Test",
    }
    assert.Nil(t, repo.SaveAccount(account))
    assert.NotEqual(t, 0, account.ID)
}
```

Testando um repo

```
func TestSaveAccountOnDBError(t *testing.T) {
    db, _ := gorm.Open("sqlite3", "../gotest.db")
    db.DropTable(&domain.Account{})
    repo := NewAccountRepo(db)
    defer db.Close()
    account := &domain.Account{
        User: "Test",
        ID:   19,
    }
    assert.NotNil(t, repo.SaveAccount(account))
}
```

Testando um service

- O teste da camada de serviço é o mais simples que temos para fazer, pois é baseado diretamente no nosso **negócio**.
- Se temos funções com as responsabilidades bem definidas, o teste se torna mais fácil! É importante definir bem cada função para que seja mais fácil a execução
- Como a camada de serviço tem chamadas a outros objetos de acesso a dados (ou outros serviços), é importante o uso de mocks para conseguirmos testar unitariamente todos os fluxos!

Testando um service

```go
func TestSaveAccountOK(t *testing.T) {
    ctrl := gomock.NewController(t)
    defer ctrl.Finish()
    mockDao := mock.NewMockRepository(ctrl)
    service := NewAccountService(mockDao)
    account := &domain.Account{
        User:    "bdm2943",
        Balance: 100,
    }
    mockDao.EXPECT().SaveAccount(account).Return(nil).AnyTimes()
    err := service.SaveAccount(account)
    assert.Nil(t, err)
}
```

Testando um service

```
func TestSaveAccountNotOK(t *testing.T) {
    ctrl := gomock.NewController(t)
    defer ctrl.Finish()
    mockDao := mock.NewMockRepository(ctrl)
    service := NewAccountService(mockDao)
    account := &domain.Account{
        User:    "bdm2943",
        Balance: 100,
    }
    mockDao.EXPECT().SaveAccount(account).Return(errors.New("Error")
    err := service.SaveAccount(account)
    assert.NotNil(t, err)
}
```

Testando um controller

- O controller é o nosso front de comunicação. Basicamente possui código responsáveis por receber o input, validar que a entrada esteja bem, executar o negócio e voltar a resposta.
- Novamente, com a ajuda dos mocks, podemos diretamente mockar a camada de serviço e nos preocuparmos somente com as regras dessa camada!

Testando um controller

- Vale lembrar que: go possui um pacote chamado httptest
- Esse pacote é tão poderoso nos testes de API que pode ser usado diretamente nos testes de libs terceiras, como gin-gonic/gin e gorilla/mux

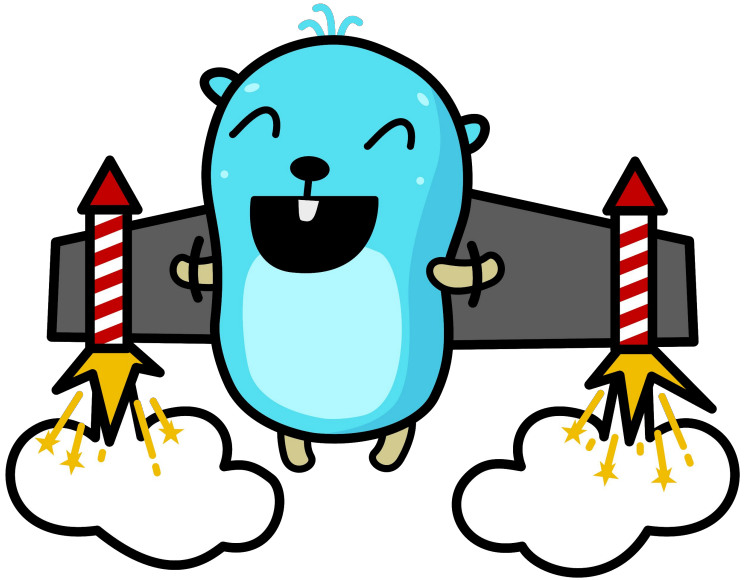
Testando um controller

```
run test | debug test
func TestPostAccountOK(t *testing.T) {
    ctrl := gomock.NewController(t)
    defer ctrl.Finish()
    service := mock.NewMockService(ctrl)
    personHandler := NewAccountHandler(
        service,
    )

    service.EXPECT().SaveAccount(gomock.Any()).DoAndReturn(func(account *domain.Account) error {
        account.ID = 1
        return nil
    })

    req, _ := http.NewRequest("POST", "/api/account", bytes.NewBufferString(`
        {
            "user": "bdm2943"
        }
    `))
    rr := httptest.NewRecorder()
    handler := http.HandlerFunc(personHandler.PostAccount)
    handler.ServeHTTP(rr, req)
    assert.Equal(t, http.StatusCreated, rr.Code)
}
```

Testes de Benchmark



Testes de Benchmark

- Diferente dos testes normais já mostrados até aqui, esse se preocupa 100% com a performance do código!
- Um bom uso é validar o quanto estamos gastando de memória com algum trecho, e quanto tempo estamos levando para realizar a operação!

Testes de Benchmark

```
run benchmark | debug benchmark
func BenchmarkProcessPayments(b *testing.B) {
    for n := 0; n < b.N; n++ {
        processPayments(mockList(1000))
    }
}

run benchmark | debug benchmark
func BenchmarkProcessPaymentsFaster(b *testing.B) {
    for n := 0; n < b.N; n++ {
        processPaymentsFaster(mockListPointer(1000))
    }
}
```

Testes de Benchmark

```go
func processPayments(payments []domain.Payment) error {
    file, _ := os.Create("/tmp/paymentsPointer")
    for _, payment := range payments {
        b, _ := json.Marshal(payment)
        file.Write(b)
    }
    return nil
}

func processPaymentsFaster(payments []*domain.Payment) error {
    file, _ := os.Create("/tmp/paymentsValue")
    for _, payment := range payments {
        b, _ := json.Marshal(payment)
        file.Write(b)
    }
    return nil
}
```

Testes de Benchmark

```
Running tool: /usr/bin/go test -benchmem -run=^$ github.com/BrunoDM2943/pock_bank-api/
payment -bench ^(BenchmarkProcessPayments|BenchmarkProcessPaymentsFaster)$

goos: linux
goarch: amd64
pkg: github.com/BrunoDM2943/pock_bank-api/payment
BenchmarkProcessPayments-4                 50      29407560 ns/op     4805119 B/op
40030 allocs/op
BenchmarkProcessPaymentsFaster-4           50      27932185 ns/op     2947024 B/op
50027 allocs/op
PASS
ok      github.com/BrunoDM2943/pock_bank-api/payment    2.922s
Success: Benchmarks passed.
```

Testes de Benchmark

- ➔ Passando os dados por ponteiro temos uma maior performance de operações por nano segundo.
- ➔ Porém, com isso, temos mais operações de alocações na memória
- ➔ É um tradeoff

Testes de Benchmark

```go
func benchmarkFib(i int, b *testing.B) {
    for n := 0; n < b.N; n++ {
        Fib(i)
    }
}

run benchmark | debug benchmark
func BenchmarkFib1(b *testing.B)  { benchmarkFib(1, b) }
run benchmark | debug benchmark
func BenchmarkFib2(b *testing.B)  { benchmarkFib(2, b) }
run benchmark | debug benchmark
func BenchmarkFib3(b *testing.B)  { benchmarkFib(3, b) }
run benchmark | debug benchmark
func BenchmarkFib10(b *testing.B) { benchmarkFib(10, b) }
run benchmark | debug benchmark
func BenchmarkFib20(b *testing.B) { benchmarkFib(20, b) }
run benchmark | debug benchmark
func BenchmarkFib40(b *testing.B) { benchmarkFib(40, b) }
```

Testes de Benchmark

```
Running tool: /usr/bin/go test -benchmem -run=^$ github.com/BrunoDM2943/pock_bank-api/
fibonnaci -bench ^(BenchmarkFib1|BenchmarkFib2|BenchmarkFib3|BenchmarkFib10|
BenchmarkFib20|BenchmarkFib40)$

goos: linux
goarch: amd64
pkg: github.com/BrunoDM2943/pock_bank-api/fibonnaci
BenchmarkFib1-4     1000000000           2.30 ns/op        0 B/op          0 allocs/op
BenchmarkFib2-4     200000000            6.04 ns/op        0 B/op          0 allocs/op
BenchmarkFib3-4     100000000           10.2 ns/op         0 B/op          0 allocs/op
BenchmarkFib10-4     5000000           384 ns/op           0 B/op          0 allocs/op
BenchmarkFib20-4       30000         46297 ns/op           0 B/op          0 allocs/op
BenchmarkFib40-4           2     700951709 ns/op           0 B/op          0 allocs/op
PASS
ok      github.com/BrunoDM2943/pock_bank-api/fibonnaci  11.688s
Success: Benchmarks passed.
```

Testes de Benchmark

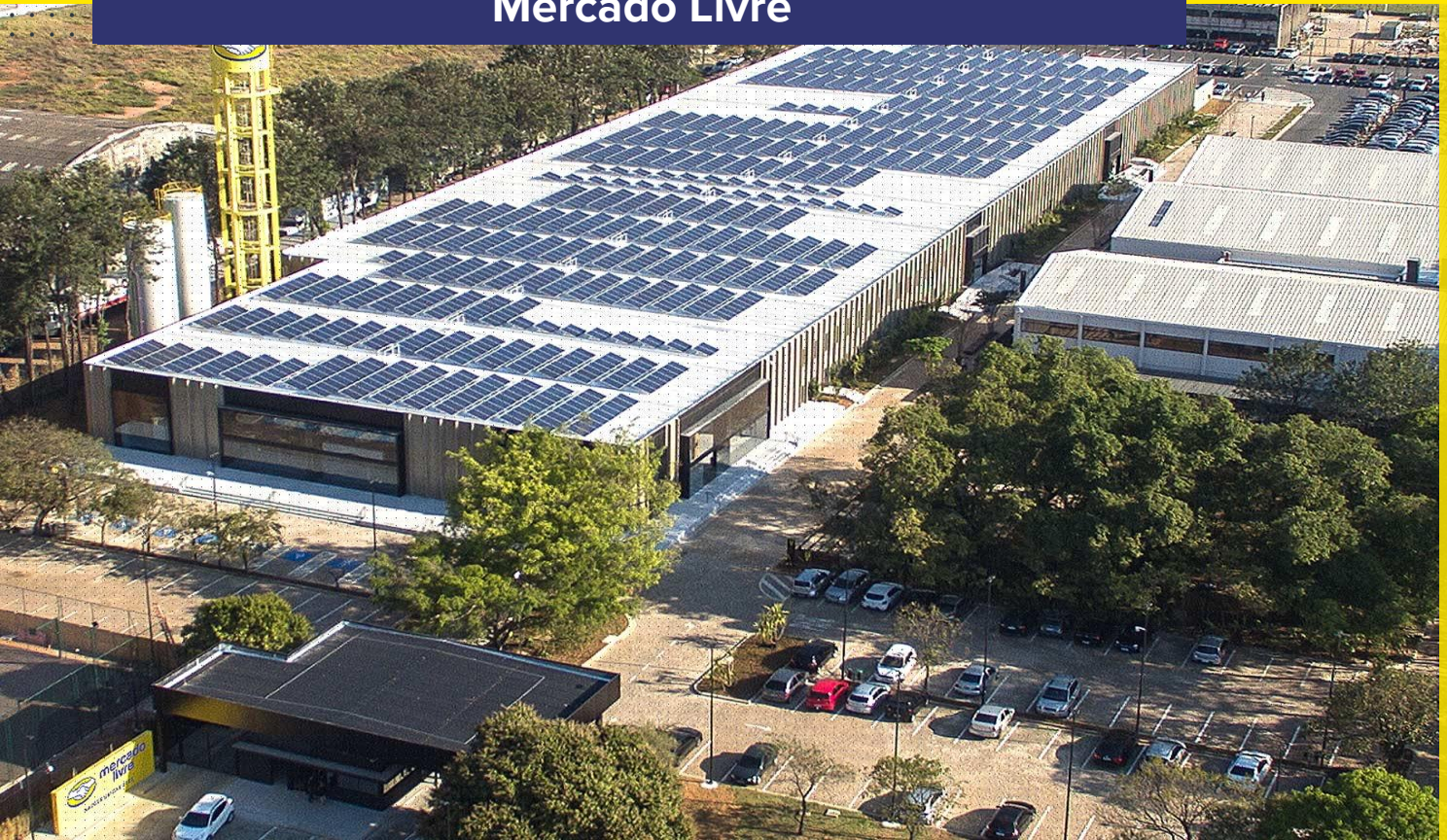
- Fibonnaci de um foi executado 1000000000 vezes com um tempo de 2ns por operação
- Fibonnaci de 10 foi executado 5000000 vezes com um tempo de 447ns por operação

#ENGINEERING

Muito obrigado!

mercado livre

Mercado Livre



+7000
funcionários

+2700
no brasil

+2400
devs

mercado livre

#ENGINEERING

./we-are-hiring.sh

mercado livre