Go-lang Unit Test

BRYAN AGAM KOTTAMA

Overview

- Pengenalan software testing
- Testing package
- Unit test
- Assertion
- Mock
- Benchmark

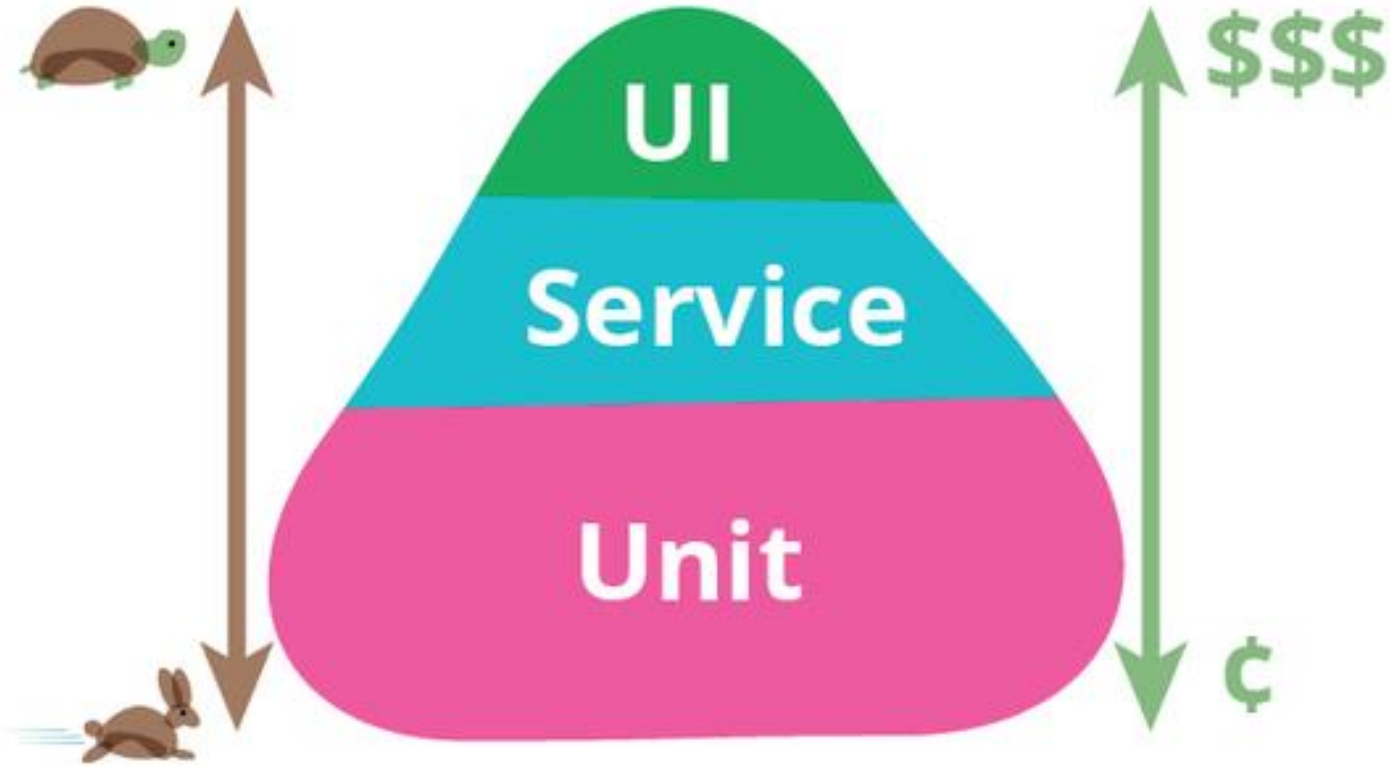
Pengenalan Software Testing

Pengenalan Software Testing

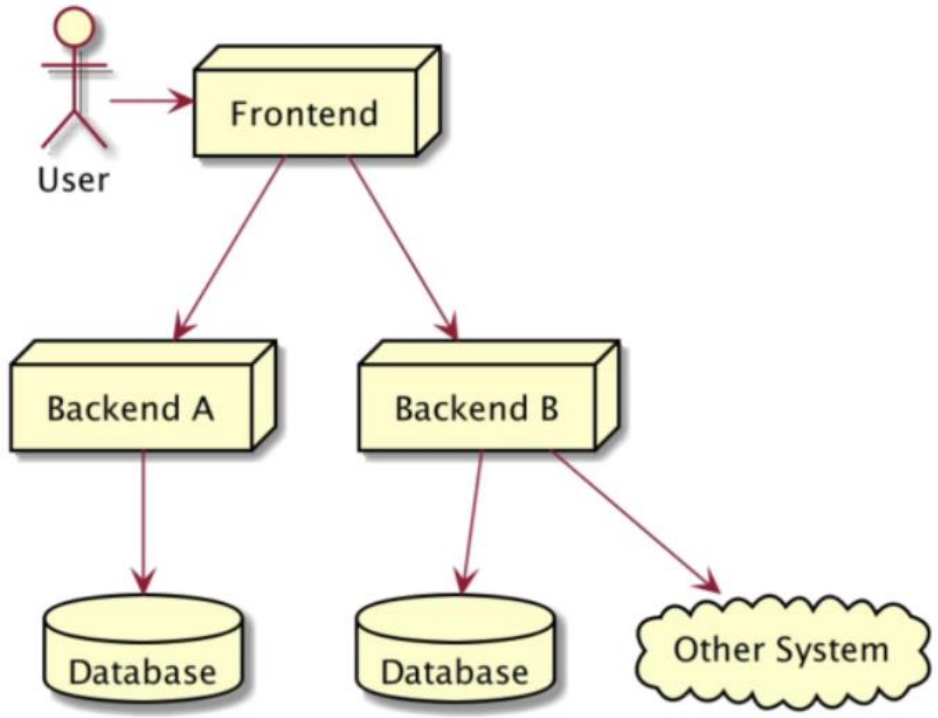
Software testing adalah salah satu disiplin ilmu dalam software engineering

Tujuan utama dari software testing adalah memastikan kualitas kode dan aplikasi kita baik

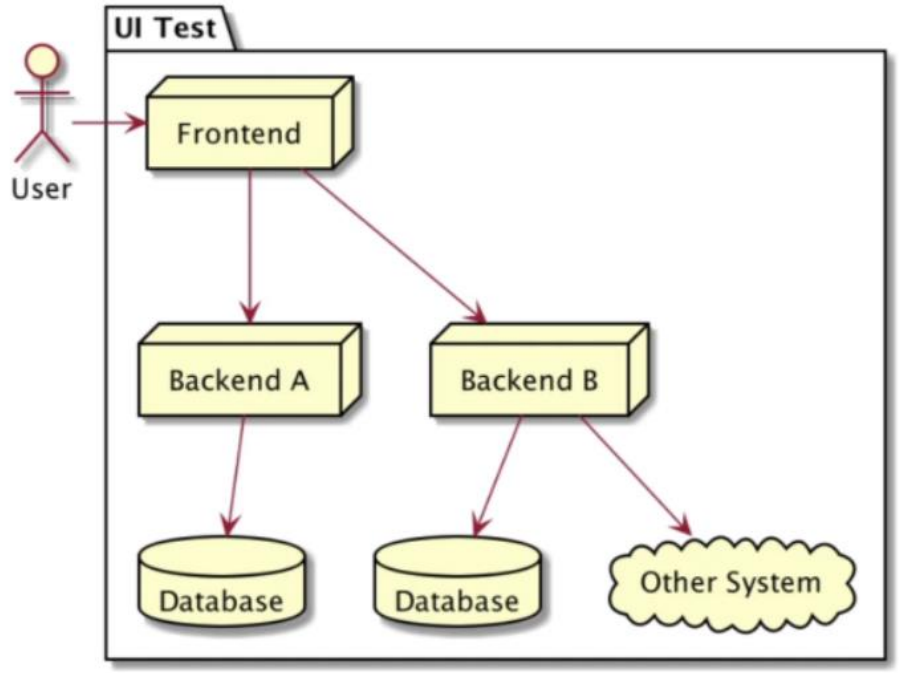
Test Pyramid



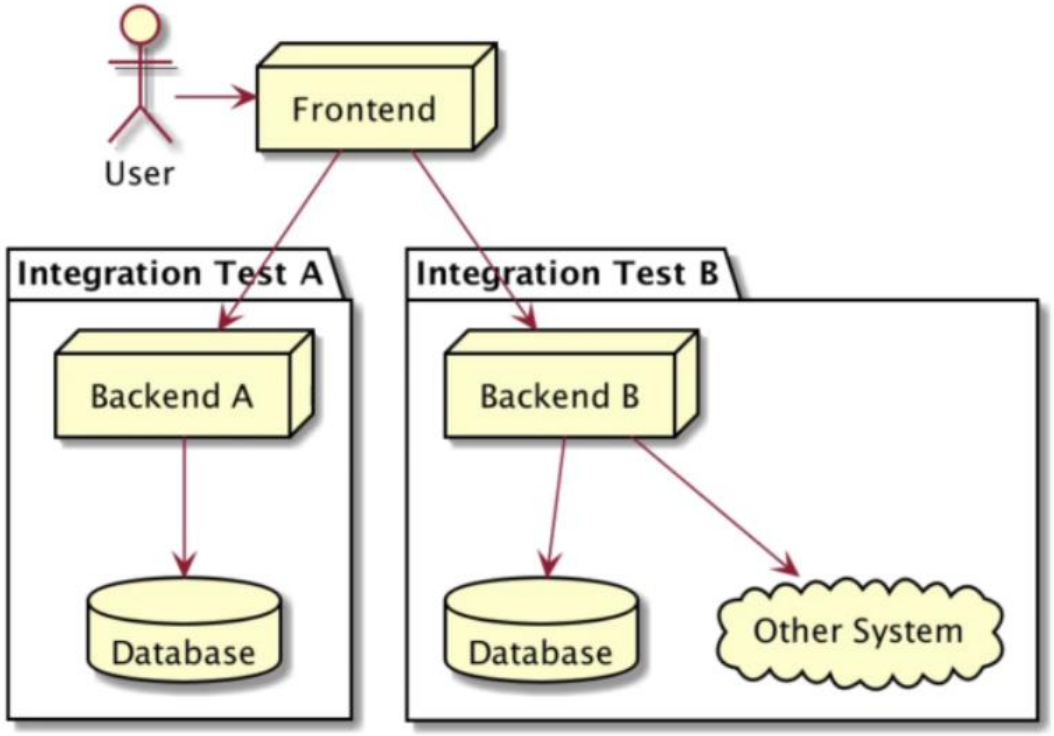
Contoh High Level Architecture Aplikasi



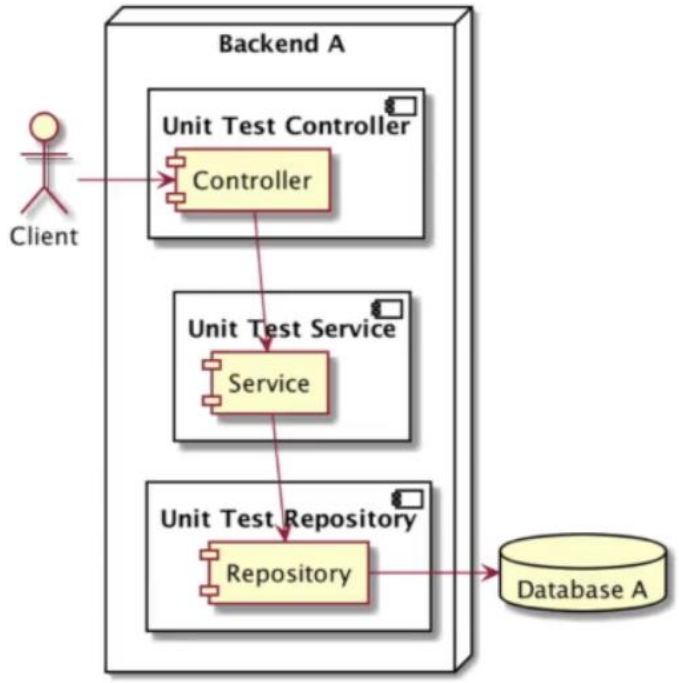
UI Test/End to End Test



Service Test/Integration Test



Unit Test


Backend A
Unit Test Controller
Controller
Client
Unit Test Service
Service
Unit Test Repository
Repository
Database A

Unit Test

Test akan focus menguji bagian kode program terkecil, biasanya menguji sebuah method

Unit test biasanya dibuat kecil dan cepat, oleh karena itu biasanya kadang unit test lebih banyak dari kode program aslinya, karena semua scenario pengujian akan dicoba di unit test

Unit test bisa digunakan sebagai cara untuk meningkatkan kualitas kode program kita

Testing Package

Testing Package

Unit testing pada Bahasa pemrograman lain, biasanya membutuhkan library tambahan atau framework khusus

Berbeda dengan Go-Lang, karena package unit test sudah disediakan bernama **testing** beserta dengan commandnya

https://golang.org/pkg/testing

testing.T

Go-Lang menyediakan sebuah struct yang bernama testing.T

Struct ini digunakan untuk unit test di Go-Lang

testing.M

testing.M adalah struct yang disediakan oleh Go-Lang untuk mengatur lifecycle testing

testing.B

testing.B adalah struct yang disediakan oleh Go-Lang untuk melakukan benchmarking

Benchmarking (mengukur kecepatan kode program) pada Go-Lang juga sudah disediakan, sehingga kita tidak perlu menggunakan framework tambahan

Kode: Hello World Function hello_world.go

```go
package helper


func HelloWorld(name string) string {

    return "Hello " + name

}
```

Aturan File Test

Go-Lang memiliki aturan cara membuat file khusus untuk unit test

Untuk membuat file unit test, kita harus menggunakan akhiran **_test**

Jadi, misalkan kita membuat file hello_world.go artinya untuk membuat file unit testnya kita harus membuat file hello_world**_test**.go

Aturan function unit test

Selain aturan nama file, di Go-Lang juga sudah diatur untuk nama function unit test

Nama function untuk unit test harus diawali dengan nama **Test**

Misal jika kita ingin melakukan testing function HelloWorld, maka kita akan membuat function unit test dengan nama **Test**HelloWorld

Tak ada aturan untuk nama belakang function unit test harus sama dengan nama function yang akan ditest

Selanjutnya harus memiliki parameter (t *testing.T) dan tidak mengembalikan return value

Kode: Hello world unit test hello_world_test.go

```go
package helper

import "testing"

func TestHelloWorld(t *testing.T) {

    result := HelloWorld("Bro")

    if result != "Hello Bro" {

        // unit test failed

        panic("Result is not Hello Bro")

    }

}
```

Menjalankan Unit Test

Untuk menjalankan unit test kita bisa menggunakan perintah: go test

Jika kita ingin lihat lebih detail fungsi test apa saja yang sudah di running, kita bisa gunakan perintah: go test –v

Jika kita ingin melakukan testing spesifik ke fungsi tertentu gunakan perintah :
go test –v –run TestNamaFunction

Menjalankan semua unit test

Jika kita ingin menggunakan semua unit test yang ada di top folder module, gunakan perintah:
go test ./..

Menggagalkan Unit Test

Menggagalkan unit test menggunakan panic bukanlah hal yang bagus

Go-Lang sendiri sudah menyediakan cara untuk menggagalkan unit test menggunakan testing.T

Terdapat function Fail(), FailNow(), Error() dan Fatal() jika kita ingin menggagalkan unit testing

t.Fail() dan t.FailNow()

Terdapat dua function untuk menggagalkan unit test, yaitu Fail() dan FailNow()

Fail() akan menggagalkan unit test, namun tetap melanjutkan eksekusi unit test. Namun diakhir Ketika selesai maka unit test tersebut dianggap gagal

FailNow() akan menggagalkan unit test saat itu juga, tanpa melanjutkan unit test

t.Error(args..) dan t.Fatal(args..)

Selain Fail() dan FailNow(), ada juga Error() dan Fatal()

Error() function lebih seperti melakukan log(print) error, namun setelah melakukan log error, dia akan secara otomatis memanggil function Fail(), sehingga mengakibatkan unit test gagal

Namun karena hanya memanggil Fail(), artinya eksekusi unit test akan tetap berjalan sampai selesai

Fatal() mirip dengan Error(), hanya saja setelah melakukan log error, dia akan memanggil FatalNow() sehingga mengakibatkan eksekusi unit test berhenti seketika

Kode: Error

```
func TestHelloWorldAgam(t *testing.T) {
    result := HelloWorld("Agam")
    if result != "Hello Agam" {
        t.Error("Harusnya Hello Agam")
    }
    fmt.Println("Dieksekusi")
}
```

Kode: Fatal

```
func TestHelloWorldAgam(t *testing.T) {
    result := HelloWorld("Agam")
    if result != "Hello Agam" {
        t.Fatal("Harusnya Hello Agam")
    }
    fmt.Println("Tidak Dieksekusi")
}
```

Assertion

Assertion

Melakukan pengecekan di unit test secara manual menggunakan if else sangatlah menyebalkan

Apabila jika result data yang harus di cek itu banyak

Oleh karena itu, sangat disarankan menggunakan Assertion untuk melakukan pengecekan

Sayangnya, Go-Lang tidak menyediakan package untuk Assertion, sehingga kita butuh menambahkan library untuk melakukan assertion ini

Testify

Salah satu library assertion yang paling popular di Go-Lang adalah Testify

Kita bisa menggunakan library ini untuk melakukan assertion terhadap result data di unit test

https://github.com/stretchr/testify

Tambahkan pada go module:
go get github.com/strechr/testify

Kode: Assertion

```
import (
    "fmt"
    "runtime"
    "testing"

    "github.com/stretchr/testify/assert"
)


func TestHelloWorldAssertion(t *testing.T) {

    result := HelloWorld("Bro")

    assert.Equal(t, "Hello Bro", result, "Result must be 'Hello Bro'")

    fmt.Println("TestHelloWorld with Assertion Done")

}
```

Assert vs Require

Testify menyediakan dua package untuk assertion, yaitu assert dan require

Saat kita menggunakan assert, jika pengecekan gagal maka assert akan memanggil Fail() artinya eksekusi unit test akan tetap dilanjutkan

Sedangkan jika menggunakan require, jika pengecekan gagal maka require akan memanggil FailNow() artinya eksekusi unit test tidak akan dilanjutkan

Kode: Require

```
func TestHelloWorldRequire(t *testing.T) {
    result := HelloWorld("Bro")
    require.Equal(t, "Hello Bro", result, "Result must be 'Hello Bro'")
    fmt.Println("TestHelloWorld with Require Done")
}
```

Skip Test

Skip Test

Kadang dalam keadaan tertentu, kita ingin membatalkan eksekusi unit test

Di Go-Lang juga kita bisa membatalkan eksekusi unit test jika kita mau

Untuk membatalkan unit test kita bisa menggunakan function Skip()

Kode: Test Skip

```
func TestSkip(t *testing.T) {
    if runtime.GOOS == "windows" {
        t.Skip("Can not run on Windows")
    }

    result := HelloWorld("Bro")
    require.Equal(t, "Hello Bro", result, "Result must be Hello Bro")
}
```

Before and After Test

Before and After Test

Biasanya dalam unit test, kadang kita ingin melakukan sesuatu sebelum dan setelah sebuah unit test dieksekusi

Jikalau kode yang kita lakukan sebelum dan setelah selalu sama antar unit test function, maka membuat manual di unit test functionnya adalah hal yang membosankan dan akan terlalu banyak code duplikat

Go-Lang menyediakan fitur yang bernama testing.M

Fitur ini bernama Main, dimana digunakan untuk mengatur eksekusi unit test, namun hal ini juga bisa kita gunakan untuk melakukan Before dan After di unit test

testing.M

Untuk mengatur eksekusi unit test, kita cukup membuat sebuah function bernama TestMain dengan parameter testing.M

Jika terdapat function TestMain tersebut, maka secara otomatis Go-Lang akan mengeksekusi function ini tiap kali akan menjalankan unit test di sebuah package

Dengan ini kita bisa mengatur Before dan After unit test sesuai dengan yang kita mau

Ingat, function TestMain itu dieksekusi hanya sekali per Go-Lang package, bukan per tiap function unit test

Kode: TestMain

```go
func TestMain(m *testing.M) {
    // before
    fmt.Println("Sebelum Unit Test")
    // eksekusi test semua testing dalam package
    m.Run()
    // after
    fmt.Println("Setelah Unit Test")
}
```

Sub Test

Sub Test

Go-Lang mendukung fitur pembuatan function unit test di dalam function unit test

Fitur ini memang sedikit aneh dan jarang sekali dimiliki di unit test di Bahasa Pemrograman yang lainnya

Untuk membuat sub test, kita bisa menggunakan function Run()

Kode: SubTest

```go
func TestSubTest(t *testing.T) {

    t.Run("Bro", func(t *testing.T) {

        result := HelloWorld("Bro")

        require.Equal(t, "Hello Bro", result)

    })


    t.Run("Agam", func(t *testing.T) {

        result := HelloWorld("Agam")

        require.Equal(t, "Hello Agam", result)

    })

}
```

Menjalan Hanya Sub Test

Kita sudah tahu jika ingin menjalankan sebuah unit test function, kita bisa menggunakan perintah: go test –run TestNamaFunction

Jika kita ingin menjalankan hanya salah satu sub test, kita bisa gunakan perintah:
go test –run TestNamaFunction/NamaSubTest

Atau untuk semua test smua sub test di semua function, kita bisa gunakan perintah:
go test –run /NamaSubTest

Table Test

Table Test

Sebelumnya kita sudah belajar tentang sub test

Jika diperintahkan, sebenarnya dengan sub test kita bisa membuat test secara dinamis

Fitur sub test, bias digunakan untuk membuat test dengan konsep table test

Table test yaitu dimana kita menyediakan data berupa slice yang berisi parameter dan ekspektasi hasil dari unit test

Lalu slice tersebut kita iteras menggunakan sub test

Kode: Table Test

```
func TestHelloWorldTable(t *testing.T) {
    tests := []struct {
        name     string
        request  string
        expected string
    }{
        {
            name:     "Bro",
            request:  "Bro",
            expected: "Hello Bro",
        },
        {
            name:     "Bryan",
            request:  "Bryan",
            expected: "Hello Bryan",

        },
        {
            name:     "Agam",
            request:  "Agam",
            expected: "Hello Agam",
        },
    }

    for _, test := range tests {

        t.Run(test.name, func(t *testing.T) {

            result := HelloWorld(test.request)

            require.Equal(t, test.expected, result)

        })

    }
}
```

Mock

Mock

Mock adalah object yang sudah kita program dengan ekspektasi tertentu sehingga ketika dipanggil, dia akan menghasilkan data yang sudah kita program diawal

Mock adalah salah satu Teknik dalam unit testing, dimana kita bisa membuat mock object dari suatu object yang memang sulit untuk di testing

Misal kita ingin membuat unit test, namun ternyata ada kode program kita yang harus memanggil API call ke third party service. Hal ini sangat sulit untuk di test, karena unit testing kita harus selalu memanggil third party service, dan belum tentu responsenya sesuai dengan apa yang kita mau

Pada kasus seperti ini, cocok sekali untuk menggunakan mock object

Testify Mock

Untuk membuat mock object, tidak ada fitur bawaan dari GOO, namun kita bisa menggunakan library testify yang sebelumnya kita gunakan untuk assertion

Testify mendukung pembuatan mock object, sehingga cocok untuk kita gunakan ketika ingin membuat mock object

Namun, perlu diperhatikan jika desain kode program kita jelek akan sulit untuk melakukan mocking. Jadi pastikan kita melakukan pembuatan desain kode program kita dengan baik

Aplikasi Query ke Database

Kita akan coba contoh kasus dengan membuat contoh aplikasi golang yang melakukan query ke database

Dimana kita akan buat layer Service sebagai business logic dan layer Repository sebagai jembatan ke database

Agar kode kita mudah untuk di test, disarankan agar membuat kontrak berupa interface

Kode: Category Entity category.go

```go
package entity


type Category struct {
    Id   string
    Name string
}
```

Kode: Category Repository category_repository.go

```go
package repository

import "github.com/bryanagamk/golang-unit-test/entity"

type CategoryRepository interface {
    FindById(id string) *entity.Category
}
```

Kode: Category Service

```go
package service

import (
    "errors"
    "github.com/bryanagamk/golang-unit-test/entity"
    "github.com/bryanagamk/golang-unit-test/repository"
)

type CategoryService struct {
    Repository repository.CategoryRepository
}
```

```go
func (service CategoryService) Get(id string)
(*entity.Category, error) {
    category := service.Repository.FindById(id)
    if category == nil {
        return category, errors.New("Category Not Found")
    } else {
        return category, nil
    }
}
```

Kode : Category Repository Mock

```go
package repository

import (
    "github.com/bryanagamk/golang-unit-test/entity"
    "github.com/stretchr/testify/mock"
)

type CategoryRepositoryMock struct {
    Mock mock.Mock
}
```

```go
func (c *CategoryRepositoryMock) FindById(id string) *entity.Category
{
    arguments := c.Mock.Called(id)
    if arguments.Get(0) == nil {
        return nil
    } else {
        category := arguments.Get(0).(entity.Category)
        return &category
    }
}
```

Kode: Category Service Unit Test 1

```
var categoryRepository = &repository.CategoryRepositoryMock{Mock: mock.Mock{}}
var categoryService = CategoryService{Repository: categoryRepository}

func TestCategoryService_GetNotFound(t *testing.T) {
    // program mock
    categoryRepository.Mock.On("FindById", "1").Return(nil)

    category, err := categoryService.Get("1")
    assert.Nil(t, category)
    assert.NotNil(t, err)
}
```

Kode: Category Service Unit Test 2

```
func TestCategoryService_GetFound(t *testing.T) {

    category := entity.Category{

        Id:   "1",

        Name: "Laptop",

    }

    categoryRepository.Mock.On("FindById", "2").Return(category)


    result, err := categoryService.Get("2")

    assert.Nil(t, err)

    assert.NotNil(t, result)

    assert.Equal(t, category.Id, result.Id)

    assert.Equal(t, category.Name, result.Name)

}
```

Benchmark

Benchmark

Selain unit test, Go-Lang testing package juga mendukung melakukan benchmark

Benchmark adalah mekanisme menghitung kecepatan performa kode aplikasi kita

Benchmark di Go-Lang dilakukan dengan cara otomatis melakukan iterasi kode yang kita panggil berkali-kali sampai waktu tertentu

Kita tidak perlu menentukan jumlah iterasi dan lamany, karena itu sudah diatur oleh testing.B bawaan dari testing package

testing.B

testing.B adalah struct yang digunakan untuk melakukan benchmark

testing.B mirip dengan testing.T terdapat function Fail(), FailNow(), Error(), Fatal() dll

Yang membedakan ada beberapa attribute dan function tambahan yang digunakan untuk melakukan benchmark

Salah satunya adalah attribute N, ini digunakan untuk melakukan total iterasi sebuah benchmark

Cara kerja Benchmark

Cara kerja benchmark di Go-Lang sangat sederhana

Kita hanya perlu membuat perulangan sejumlah N attribute

Secara otomatis, lalu mendeteksi berapa lama proses tersebut berjalan dan disimpulkan performa benchmarknya dalam waktu

Benchmark function

Mirip seperti unit test, untuk benchmark di Go-Lang sudah ditentukan nama functionnya, harus diawali dengan kata Benchmark, missal BenchmarkHelloWorld

Selain itu, harus memiliki parameter (b *testing.B)

Dan tidak boleh mengembalikan return value

Untuk nama file, sama seperti dengan unit test dimana harus diakhiri dengan _test, misal hello_world_test.go

Kode: Benchmark function

```
func BenchmarkHelloWorld(b *testing.B) {
    for i := 0; i < b.N; i++ {
        HelloWorld("Bro")
    }
}
```

Menjalankan Benchmark

Untuk menjalankan seluruh benchmark di module: go test –v –bench=.

Menjalankan benchmark tanpa unit test: go test –v –run=NotMathUnitTest –bench=.

Menjalankan spesifik benchmark tanpa unit test: go test –v –run=NotMathUnitTest –bench=BenchmarkTest

Menjalankan benchmark di root module dan semua module dijalankan: go test –v –bench=./..

Sub Benchmark

Sama seperti testing.T, di testing.B juga bisa membuat sub benchmark menggunakan function Run()

Kode: Sub Benchmark

```
func BenchmarkHelloWorldSub(b *testing.B) {
    b.Run("Bro", func(b *testing.B) {
        for i := 0; i < b.N; i++ {
            HelloWorld("Bro")
        }
    })
    b.Run("Kottama", func(b *testing.B) {
        for i := 0; i < b.N; i++ {
            HelloWorld("Kottama")
        }
    })
}
```

Menjalankan Sub Benchmark

Menjalankan salah satu sub benchmark: go test –v –bench=BenchmarkNama/NamaSub

Table Benchmark

Table Benchmark

Sama seperti di unit test, Go-Lang developer terbiasa membuat table benchmark

Ini berfungsi untuk melakukan performance test dengan kombinasi data berbeda tanpa harus membuat banyak benchmark function

Kode: Table Test

```
func BenchmarkTable(b *testing.B) {

    persons := []struct {

        name string

    }{
        {
            name: "Bro",
        },
        {
            name: "Bryan",
        },
        {
            name: "Kottama",
        },
        {
            name: "Candra Gupta",
        },
    }

    for _, person := range persons {

        b.Run(person.name, func(b *testing.B) {

            for i := 0; i < b.N; i++ {

                HelloWorld(person.name)

            }

        })

    }

}
```