Scalable Web Service With Golang
sesi 11

Unit Test

Unit Test

Bahasa Go telah menyediakan suatu package bernama *testing*, yang dapat kita gunakan untuk membuat pada aplikasi bahasa Go kita. Terdapat aturan dari penamaan file test yang harus kita ikuti guna untuk membuat file testing pada bahasa Go. Berikut merupakan aturannya:

- File testing harus diakhiri dengan penamaan _test.go
- Misalkan kita mempunyai sebuah file bernama calculation.go, dan ketika kita ingin membuat file testing untuk file calculation.go, maka perlu membuat file dengan nama calculation_test.go. Perlu diingat disini bahwa penaman file testing boleh bebas, asalkan di akhiri dengan penaman _test.go

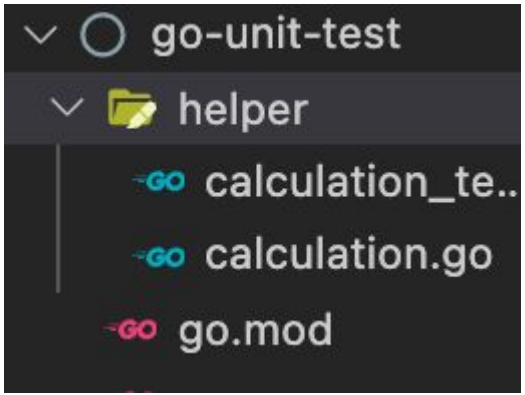
Terdapat juga aturan untuk membuat function testing pada bahasa Go. Berikut merupakan aturannya:

- Function untuk testing harus diwali dengan penamaan *Test*.
- Misalkan kita mempunyai suatu function bernama *sum* dan kita ingin membuat unit test dari function tersebut, maka kita dapat memberikan nama untuk function unit test kita seperti *TestSum*.
- Perlu diingat disini bahwa tidak ada aturan untuk nama belakang dari suatu function testing, yang terpenting adalah harus diawali dengan penaman *Test*.
- Kemudian function testing harus menerima satu parameter dengan tipe data **testing.T* dan tidak mengembalikan suatu return value apapun. Tipe data **testing.T* merupakaan sebuah *struct* dari package *testing* untuk membuat suatu unit test.

Setup unit test workspace

Mari kita belajar sama-sama tentang unit test pada bahasa Go. Untuk itu buatlah satu folder bernama **go-unit-test** dan jalankan perintah **go mod init go-unit-test** pada terminal di dalam folder tersebut. Kemudian buatlah suatu folder bernama **helper** di dalam folder **go-unit-test**. Kemudian buatlah 2 file didalam folder helper dengan nama **calculation.go** dan **calculation_test.go**.

Jika sudah maka folder tree kita akan terlihat seperti pada gambar dibawah ini.



Create Unit Test #1

Pertama-tama kita akan membuat suatu function bernama *Sum* pada file **calculation.go**. Function Sum akan kita gunakan untuk melakukan penjumlahan yang sangat sederhana. Contohnya seperti pada gambar dibawah ini.

```go
package helper

func Sum(a, b int) int {
    return a + b
}

```

Create Unit Test #2

Kemudian kita akan membuat function yang bernama *TestSum* pada file calculation_test.go. Function ini akan kita gunakan untuk membuat unit test terhadap function *Sum*. Caranya seperti pada gambar disebelah kanan.

Bisa dilihat pada line 6, kita memanggil function *TestSum* dan memberikan dua angka sebagai argumennya berupa 10 dan 10.

Kemudian pada line 8 kita melakukan pengecekan hasil dari unti testnya. Jika hasilnya bukan angka 20, maka kita akan menggagalkan testnya dengan function *panic*.

```
1  package helper
2
3  import "testing"
4
   run test | debug test
5  func TestSum(t *testing.T) {
6      result := Sum(10, 10)
7
8      if result != 20 {
9          panic("Result should be 20")
10     }
11 }
```

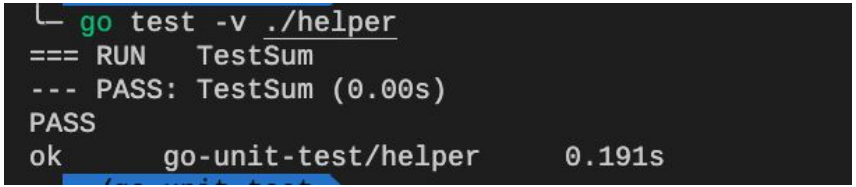
Create Unit Test #3

Kemudian agar kita dapat menjalankan file testing kita, maka kita perlu menjalankan perintah **go test**.

Tetapi karena file testing kita berada didalam package *helper* dan kita ingin menjalankan file testing kita dari root direktori kita, maka kita dapat menjalankan perintah **go test ./helper**.

Maka hasilnya akan seperti pada gambar pertama di sebelah kanan. Gambar pertama disebelah kanan menunjukkan bahwa unit test kita telah berhasil yang ditandai dengan kalimat **ok**.

Kemudian jika kita ingin lebih mengetahui secara detail function unit test yang mana yang kita eksekusi, maka kita dapat menjalankan perintah **go test -v ./helper**. Maka hasilnya akan seperti pada gambar kedua.



Create Failed Unit Test

Untuk menggagalkan sebuah unit test, kita dapat menggunakan 4 method yang telah disediakan oleh package *testing.* Berikut merupakan penjelasan dari ke -4 method-methodnya:

- t.Fail() : Digunakan untuk menggagalkan sebuah unit test namun proses eksekusi dari unit test nya tidak akan terhenti.
- t.FailNow() : Digunakan untuk menggagalkan sebuah unit test namun dan proses eksekusi nya akan terhenti pada unit test tersebut dan unit test lainnya tidak akan tereksekusi.
- t.Error(...args) : Digunakan untuk menggagalkan sebuah unit test sekaligus melakukan logging dari errornya agar kita dapat membuat pesan dari error yang terjadi.
- t.Fatal(...args) : Fungsi nya mirip seperti t.FailNow(), namun t.Fatal(...args) dapat melakukan logging dari pesan errornya.

Fail method

Mari kita buat suatu function unit test yang bernama *TestFailSum* yang akan kita gunakan untuk membuat unit test yang gagal. Caranya seperti pada gambar pertama disebelah kanan.

Bisa dilihat bahwa kita membuat function *TestFailSum* diatasnya function *TestSum* yang telah kita buat sebelumnya. Hal ini dikarenakan kita akan mencoba untuk membuat test yang gagal, dan kita akan mencoba method testing untuk menggagalkan sebuah test yang dapat menghentikan eksekusi function unit test di bawahnya.

Bisa dilihat pada line 11, kita sengaja untuk menggagalkan unit test nya dengan membuat ekspektasi yang salah. Lalu kita menggunakan method *Fail* untuk menggagalkan test nya.

Jika kita jalankan pada terminal kita, maka hasilnya akan seperti pada gambar kedua. Bisa dilihat bahwa function *TestFailSum* gagal tetapi karena kita memaki method *Fail*, maka eksekusi tidak akan dihentikan. Hal ini dapat dilihat dari *fmt.Println* pada line 15 masih tereksekusi.

```go
1  package helper
2
3  import (
4    "fmt"
5    "testing"
6  )
7
   run test | debug test
8  func TestFailSum(t *testing.T) {
9    result := Sum(10, 10)
10
11   if result != 40 {
12     t.Fail()
13   }
14
15   fmt.Println("TestFailSum Eksekusi Terhenti")
16 }
17
   run test | debug test
18 func TestSum(t *testing.T) {
19   result := Sum(10, 10)
20
21   if result != 20 {
22     t.FailNow()
23   }
24
25   fmt.Println("TestSum Eksekusi Terhenti")
26 }
```

```
=== RUN   TestFailSum
TestFailSum Eksekusi Terhenti
--- FAIL: TestFailSum (0.00s)
=== RUN   TestSum
TestSum Eksekusi Terhenti
--- PASS: TestSum (0.00s)
FAIL
FAIL    go-unit-test/helper     0.185s
FAIL
```

FailNow method

Sekarang kita memakai method *FailNow*, jika kita jalankan maka hasilnya akan seperti pada gambar kedua.

Bisa dilihat bahwa *fmt.Println* pada line 15 tidak tereksekusi karena method *FailNow* akan memberhentikan eksekusi dari sebuah function unit test.

```go
package helper

import (
  "fmt"
  "testing"
)

run test | debug test
func TestFailSum(t *testing.T) {
  result := Sum(10, 10)

  if result != 40 {
    t.FailNow()
  }

  fmt.Println("TestFailSum Eksekusi Terhenti")
}

run test | debug test
func TestSum(t *testing.T) {
  result := Sum(10, 10)

  if result != 20 {
    t.FailNow()
  }

  fmt.Println("TestSum Eksekusi Terhenti")
}
```

```
=== RUN   TestFailSum
--- FAIL: TestFailSum (0.00s)
=== RUN   TestSum
TestSum Eksekusi Terhenti
--- PASS: TestSum (0.00s)
FAIL
FAIL    go-unit-test/helper     0.188s
FAIL
```

Error method

Sekarang perhatikan ketika kita memakai method *Error*. Method *Error* memiliki fungsi yang sama seperti method *Fail,* tetapi kita dapat melakukan logging untuk menampilkan pesan error di terminal.

Jika kita jalankan maka hasilnya akan seperti pada gambar kedua.

```go
package helper

import (
  "fmt"
  "testing"
)

run test | debug test
func TestFailSum(t *testing.T) {
  result := Sum(10, 10)

  if result != 40 {
    t.Error("Result has to be 40")
  }

  fmt.Println("TestFailSum Eksekusi Terhenti")
}

run test | debug test
func TestSum(t *testing.T) {
  result := Sum(10, 10)

  if result != 20 {
    t.FailNow()
  }

  fmt.Println("TestSum Eksekusi Terhenti")
}
```

```
=== RUN   TestFailSum
    calculation_test.go:12: Result has to be 40
TestFailSum Eksekusi Terhenti
--- FAIL: TestFailSum (0.00s)
=== RUN   TestSum
TestSum Eksekusi Terhenti
--- PASS: TestSum (0.00s)
FAIL
FAIL    go-unit-test/helper     0.423s
FAIL
```

Fatal method

Method *Fatal* akan memiliki fungsi yang sama dengan method *FailNow,* tetapi method *Fatal* dapat membuat logging untuk menampilkan pesan error.

Jika kita jalankan maka hasilnya akan seperti pada gambar kedua.

```go
package helper

import (
  "fmt"
  "testing"
)

run test | debug test
func TestFailSum(t *testing.T) {
  result := Sum(10, 10)

  if result != 40 {
    t.Error("Result has to be 40")
  }

  fmt.Println("TestFailSum Eksekusi Terhenti")
}

run test | debug test
func TestSum(t *testing.T) {
  result := Sum(10, 10)

  if result != 20 {
    t.FailNow()
  }

  fmt.Println("TestSum Eksekusi Terhenti")
}
```

```
=== RUN   TestFailSum
    calculation_test.go:12: Result has to be 40
TestFailSum Eksekusi Terhenti
--- FAIL: TestFailSum (0.00s)
=== RUN   TestSum
TestSum Eksekusi Terhenti
--- PASS: TestSum (0.00s)
FAIL
FAIL    go-unit-test/helper     0.423s
FAIL
```

Testify

Agar kita tidak perlu menggunakan banyak kondisional *if else* pada unit test kita, maka kita perlu menggunakan mekanisme *Assertion*.

Untuk melakukan *assertion*, maka kita dapat menginstall terlebih dahulu suatu library bernama Testify.

Testify adalah suatu library pada Go untuk melakukan assertion, mocking dan masih banyak lainnya.

Jalankan perintah dibawah ini untuk menginstall library Testify.

```
go get github.com/stretchr/testify
```

Testify (assert and require)

Sekarang kita akan mencoba untuk menerapkan package *assert* dan *require* yang berasal dari library Testify.

Perbedaan dari package *assert* dan require adalah ketika terjadi sebuah error.

Package *assert* tidak akan menghentikan eksekusi function unit test ketika terjadi sebuah error.

Package require akan menghentikan eksekusi function unit test ketika terjadi sebuah error.

Contoh penerapannya sepertin pada gambar disebelah kanan berikut ini.

```go
package helper

import (
  "fmt"
  "testing"

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

run test | debug test
func TestFailSum(t *testing.T) {
  result := Sum(10, 10)

  require.Equal(t, 40, result, "Result has to be 40")

  fmt.Println("TestFailSum Eksekusi Terhenti")
}

run test | debug test
func TestSum(t *testing.T) {
  result := Sum(10, 10)

  assert.Equal(t, 20, result, "Result has to be 20")

  fmt.Println("TestSum Eksekusi Terhenti")
}
```

Testify (assert and require)

Pada line 14, kita menggunakan package *require* dengan function *Equal*. Function *Equal* pada package *require* ataupun *package assert* menerima 4 parameter.

- Parameter pertama: Menerima sebuah interface bernama *TestingT* yang dimaan interface ini mempunyai 2 *method* yaitu *Errorf* dan *FailNow*.
- Parameter kedua: Menerima hasil yang kita harapkan atau ekspektasi dari unit test.
- Parameter ketiga: Menerima operasi dari function yang ingin kita test.
- Parameter keempat: Menerima pesan error yang nantinya akan ditampilkan ketika terjadi sebuah error.

```
1  package helper
2
3  import (
4    "fmt"
5    "testing"
6
7    "github.com/stretchr/testify/assert"
8    "github.com/stretchr/testify/require"
9  )
10
   run test | debug test
11 func TestFailSum(t *testing.T) {
12   result := Sum(10, 10)
13
14   require.Equal(t, 40, result, "Result has to be 40")
15
16   fmt.Println("TestFailSum Eksekusi Terhenti")
17 }
18
   run test | debug test
19 func TestSum(t *testing.T) {
20   result := Sum(10, 10)
21
22   assert.Equal(t, 20, result, "Result has to be 20")
23
24   fmt.Println("TestSum Eksekusi Terhenti")
25 }
26
```

Testify (assert and require)

Package *require* akan menghentikan proses eksekusi function unit test ketika terjadi sebuah error, yang berarti function *fmt.Println* pada line 16 tidak akan dieksekusi.

Mari kita jalankan kembali file testing kita, maka hasilnya akan seperti pada gambar kedua.

```go
package helper

import (
  "fmt"
  "testing"

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

run test | debug test
func TestFailSum(t *testing.T) {
  result := Sum(10, 10)

  require.Equal(t, 40, result, "Result has to be 40")

  fmt.Println("TestFailSum Eksekusi Terhenti")
}

run test | debug test
func TestSum(t *testing.T) {
  result := Sum(10, 10)

  assert.Equal(t, 20, result, "Result has to be 20")

  fmt.Println("TestSum Eksekusi Terhenti")
}
```

```
=== RUN   TestFailSum
    calculation_test.go:14:
                Error Trace:    calculation_test.go:14
                Error:          Not equal:
                                expected: 40
                                actual  : 20
                Test:           TestFailSum
                Messages:       Result has to be 40
--- FAIL: TestFailSum (0.00s)
=== RUN   TestSum
TestSum Eksekusi Terhenti
--- PASS: TestSum (0.00s)
FAIL
exit status 1
FAIL    go-unit-test/helper     0.917s
```

HACKTIV8

Table Test

Perlu teman-teman ketahui bahwa function *TestTableSum* pada gambar di sebelah kanan masih dikerjakan pada file yang sama seperti function unit test yang kita kerjakan sebelumnya.

Perhatikan pada line 11 - 32, kita membuat *slice of struct* yang berisikan field-field/property untuk keperluan unit test kita.

- request: Digunakan untuk menaruh operasi dari function yang akan kita test.
- expected: Digunakan untuk menaruh nilai/hasil yang kita ekspektasikan.
- errMsg: Digunakan untuk menaruh pesan error yang akan ditampilkan jika terjadi error.

```
11  func TestTableSum(t *testing.T) {
12    tests := []struct {
13      request  int
14      expected int
15      errMsg   string
16    }{
17      {
18        request:  Sum(10, 10),
19        expected: 20,
20        errMsg:   "Result has to be 20",
21      },
22      {
23        request:  Sum(20, 20),
24        expected: 40,
25        errMsg:   "Result has to be 40",
26      },
27      {
28        request:  Sum(25, 25),
29        expected: 50,
30        errMsg:   "Result has to be 50",
31      },
32    }
33
34    for i, test := range tests {
35      t.Run(fmt.Sprintf("test-%d", i), func(t *testing.T) {
36        require.Equal(t, test.expected, test.request, test.errMsg)
37      })
38    }
39  }
40
```

Table Test

Pada line 34-38, kita melakukan looping menggunakan *range loop*, yang dimana kita mengeksekusi unit test didalam looping tersebut.

Jika kita perhatikan apda line 35, terdapat sebuah hal baru yang belum dijelaskan pada halaman-halaman sebelumnya.

t.Run merupakan sebuah method dari struct *T* yang digunakan untuk melakukan sub-test.

Dengan seperti ini maka kita dapat melakukan lebih dari satu test pada satu function unit test.

```
11  func TestTableSum(t *testing.T) {
12    tests := []struct {
13      request  int
14      expected int
15      errMsg   string
16    }{
17      {
18        request:  Sum(10, 10),
19        expected: 20,
20        errMsg:   "Result has to be 20",
21      },
22      {
23        request:  Sum(20, 20),
24        expected: 40,
25        errMsg:   "Result has to be 40",
26      },
27      {
28        request:  Sum(25, 25),
29        expected: 50,
30        errMsg:   "Result has to be 50",
31      },
32    }
33
34    for i, test := range tests {
35      t.Run(fmt.Sprintf("test-%d", i), func(t *testing.T) {
36        require.Equal(t, test.expected, test.request, test.errMsg)
37      })
38    }
39  }
40
```

Table Test

Method *t.Run* menerima 2 parameter:

- Parameter pertama: Menerima sebuah nilia dengan tipe data *string* untuk menaruh nama dari sub-test nya.
- Parameter kedua: Menerima sebuah function callback yang kita gunakan untuk mengeksekusi proses unit test kita.

```go
func TestTableSum(t *testing.T) {
  tests := []struct {
    request  int
    expected int
    errMsg   string
  }{
    {
      request:  Sum(10, 10),
      expected: 20,
      errMsg:   "Result has to be 20",
    },
    {
      request:  Sum(20, 20),
      expected: 40,
      errMsg:   "Result has to be 40",
    },
    {
      request:  Sum(25, 25),
      expected: 50,
      errMsg:   "Result has to be 50",
    },
  }

  for i, test := range tests {
    t.Run(fmt.Sprintf("test-%d", i), func(t *testing.T) {
      require.Equal(t, test.expected, test.request, test.errMsg)
    })
  }
}
```

Table Test

Sekarang mari kita coba untuk menjalankan function *TestTableSum* saja.

Ketika kita ingin menjalankan hanya satu function unit test saja, maka kita dapat mengeksekusi perintah berikut:

go test -v ./helper -run=TestTableSum.

Maka hasilnya akan seperti pada gambar di bawah ini..

```
└─ go test -v ./helper -run=TestTableSum
=== RUN   TestTableSum
=== RUN   TestTableSum/test-0
=== RUN   TestTableSum/test-1
=== RUN   TestTableSum/test-2
--- PASS: TestTableSum (0.00s)
    --- PASS: TestTableSum/test-0 (0.00s)
    --- PASS: TestTableSum/test-1 (0.00s)
    --- PASS: TestTableSum/test-2 (0.00s)
PASS
ok      go-unit-test/helper     1.535s
```

Mock

Mock merupakan sebuah object yang sudah kita program dengan eksepektasi tertentu sehingga ketika object tersebut dipanggil, maka ia akan menghasilkan data sesuai dengan yang sudah kita progam diawal.

Mock juga merupakan salah satu teknik pada unit test, dimana kita dapat membuat object dari suatu object yang memang sulit untuk di testing.

Contohnya ketika kita ingin membuat unit test, tetapi ternyata kode dari program kita perlu memanggil *API* dari pihak ketiga dan kita mungkin tidak tahu apakah hasil yang kita dapatkan dari *API* tersebut akan selalu sama atau tidak.

Maka untuk kasus tersebut, kita dapat membuat object mock untuk melakukan *mocking* terhadap API tersebut.

Mock (Create Entity)

Saat ini kita akan mencoba untuk membuat contoh kasus yang dimana kita akan melakukan query kepada database.

Kita akan membuat sebuah layer service sebagai business logic nya, dan sebuah layer repository untuk jembatan kepada database.

Sekarang buatlah sebuah folder dengan nama **entity**, setelah itu buatlah sebuah file di dalamnya dengan nama **product.go**. Kemudian isilah dengan kode seperti pada gambar di sebelah kanan.

```
1  package entity
2
3  type Product struct {
4      Id   string
5      Name string
6  }
7
```

Mock (Create Repository Interface)

Kemudian kita akan membuat interface untuk repository kita.

Maka dari itu buat lah sebuah folder dengan nama **repository** dan buat lah satu file dengan nama **product_repository.go**.

Lalu isilah dengan kode seperti pada gambar di sebelah kanan pada file **product_repository.go**.

```go
package repository

import "go-unit-test/entity"

type ProductRepository interface {
    FindById(id string) *entity.Product
}
```

Mock (Create Interface)

Kemudian kita akan membuat *struct* untuk service kita dan juga method untuk mendapatkan satu data product.

Maka dari itu buat lah sebuah folder dengan nama **service** dan buat lah satu file dengan nama **product_service.go**.

Lalu isilah dengan kode seperti pada gambar di bawah pada file **product_service.go**.

```go
package service

import (
  "errors"
  "go-unit-test/entity"
  "go-unit-test/repository"
)

type ProductService struct {
  Repository repository.ProductRepository
}

func (service ProductService) GetOneProduct(id string) (*entity.Product, error) {
  product := service.Repository.FindById(id)
  if product == nil {
    return nil, errors.New("product not found")
  }

  return product, nil
}
```

Mock (Create Interface)

Kemudian kita akan membuat *struct* untuk service kita dan juga method untuk mendapatkan satu data product.

Maka dari itu buat lah sebuah folder dengan nama **service** dan buat lah satu file dengan nama **product_service.go**.

Lalu isilah dengan kode seperti pada gambar di bawah pada file **product_service.go**.

```go
package service

import (
  "errors"
  "go-unit-test/entity"
  "go-unit-test/repository"
)

type ProductService struct {
  Repository repository.ProductRepository
}

func (service ProductService) GetOneProduct(id string) (*entity.Product, error) {
  product := service.Repository.FindById(id)
  if product == nil {
    return nil, errors.New("product not found")
  }

  return product, nil
}
```

Mock (Create Mock Object)

Tipe testing yang kita lakukan saat ini adalah unit test, dan pada unit test idealnya adalah kita tidak perlu menjalankan sistem dari pihak ketiga yang dimana salah satu sistem tersebut adalah database.

Maka dari itu, kita akan membuat object mock untuk melakukan mocking terhadap interface product repository kita, sehingga service yang telah kita buat dapat di test.

Buatlah sebuah file dengan nama product_repository_mock.go dan isilah kode pada gambar di sebelah kanan.

```go
package repository

import (
  "go-unit-test/entity"

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

type ProductRepositoryMock struct {
  Mock mock.Mock
}

func (repository *ProductRepositoryMock) FindById(id string) *entity.Product {
  arguments := repository.Mock.Called(id)

  if arguments.Get(0) == nil {
    return nil
  }

  product := arguments.Get(0).(entity.Product)

  return &product
}
```

Mock (Create Mock Object)

Jika kita perhatikan, kita menambahkan field bernama *Mock* pada struct *ProductRepositoryMock*.

Field *Mock* tersebut memiliki tipe data struct *mock.Mock* yang berasal dari pacakge *mock* dari library Testify.

Struct *mock.Mock* tersebut memiliki berbagai macam method untuk keperluan proses mocking.

Pada line 14, kita memanggil method *Called* untuk memanggil object mock dan akan mereturn sebuah tipe data mock.Arguments yang merupakan tipe data aliase dari *slice of interface*.

Karena method *FindById* tersebut digunakan untuk mengambil satu data product berdasarkan id product, maka dari itu pada line 16, kita menggunakan method *Get* untuk mengambil data berdasarkan index dari pemanggilan method *Called*.

```
1  package repository
2
3  import (
4    "go-unit-test/entity"
5
6    "github.com/stretchr/testify/mock"
7  )
8
9  type ProductRepositoryMock struct {
10   Mock mock.Mock
11 }
12
13 func (repository *ProductRepositoryMock) FindById(id string) *entity.Product {
14   arguments := repository.Mock.Called(id)
15
16   if arguments.Get(0) == nil {
17     return nil
18   }
19
20   product := arguments.Get(0).(entity.Product)
21
22   return &product
23 }
```

Mock (Create Service Unit Test)

Sekarang kita akan membuat testing untuk service yang telah kita buat.

Buatlah sebuah file didalam folder **service** dengan nama **product_service_test.go**, dan isilah dengan kode seperti pada gambar disebelah kanan untuk file tersebut.

Testing yang kita buat sekarang adalah testing response error terhadapt method *GetOneProduct* pada service kita.

Kemudian kita menentukan response dari pemanggilan object mock menggunakan method On seperti pada line 16.

```
1  package service
2
3  import (
4    "go-unit-test/entity"
5    "go-unit-test/repository"
6    "testing"
7
8    "github.com/stretchr/testify/assert"
9    "github.com/stretchr/testify/mock"
10 )
11
12 var productRepository = &repository.ProductRepositoryMock{Mock: mock.Mock{}}
13 var productService = ProductService{Repository: productRepository}
14
   run test | debug test
15 func TestProductServiceGetOneProductNotFound(t *testing.T) {
16   productRepository.Mock.On("FindById", "1").Return(nil)
17
18   product, err := productService.GetOneProduct("1")
19
20   assert.Nil(t, product)
21   assert.NotNil(t, err)
22   assert.Equal(t, "product not found", err.Error(), "error response has to be 'product not
     found'")
23 }
24
```

Mock (Create Service Unit Test)

Method On menerima 2 parameter:

- Parameter pertama: Menerima nama dari method yang akan kita mocking.
- Parameter kedua: Menerima argument yang ingin kita berikan pada method yang ingin kita mocking.

Kemudian kita melakukan chaining dengan menyambungkan method *On* dengan method *Return*.

Method *Return* kita gunakan untuk menentukan response yang akan dihasilkan oleh object yang akan kita mocking. (Pada kasus kita sekarang, object tersebut berarti adalah method *FindById* milik struct ProductRepositoryMock).

```go
package service

import (
  "go-unit-test/entity"
  "go-unit-test/repository"
  "testing"

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

var productRepository = &repository.ProductRepositoryMock{Mock: mock.Mock{}}
var productService = ProductService{Repository: productRepository}

run test | debug test
func TestProductServiceGetOneProductNotFound(t *testing.T) {
  productRepository.Mock.On("FindById", "1").Return(nil)

  product, err := productService.GetOneProduct("1")

  assert.Nil(t, product)
  assert.NotNil(t, err)
  assert.Equal(t, "product not found", err.Error(), "error response has to be 'product not found'")
}
```

Mock (Create Service Unit Test)

Kemudian kita melakukan assertion yang dimana kita berekspektasi bahwa variable *err* pada line 18 tidak akan menjadi nil, variable *product* pada line 18 akan menjadi nil, dan kita akan mendapatkan pesan error berupa “product not found”.

```
1  package service
2
3  import (
4    "go-unit-test/entity"
5    "go-unit-test/repository"
6    "testing"
7
8    "github.com/stretchr/testify/assert"
9    "github.com/stretchr/testify/mock"
10 )
11
12 var productRepository = &repository.ProductRepositoryMock{Mock: mock.Mock{}}
13 var productService = ProductService{Repository: productRepository}
14
   run test | debug test
15 func TestProductServiceGetOneProductNotFound(t *testing.T) {
16   productRepository.Mock.On("FindById", "1").Return(nil)
17
18   product, err := productService.GetOneProduct("1")
19
20   assert.Nil(t, product)
21   assert.NotNil(t, err)
22   assert.Equal(t, "product not found", err.Error(), "error response has to be 'product not
     found'")
23 }
24
```

Mock (Create Service Unit Test)

Mari kita jalankan function unit test dari service kita dengan mengkesekusi perintah dibawah ini:

go test -v ./service

Maka hasilnya akan seperti pada gambar di bawah ini.

```
└─ go test -v ./service
=== RUN   TestProductServiceGetOneProductNotFound
--- PASS: TestProductServiceGetOneProductNotFound (0.00s)
PASS
ok      go-unit-test/service    0.499s
```

Mock (Create Service Unit Test)

Jika kita ingin membuat test untuk response sukses. Maka kita dapat menambahkan function unit test kita seperti pada gambar di bawah ini.

```go
func TestProductServiceGetOneProduct(t *testing.T) {
  product := entity.Product{
    Id:   "2",
    Name: "Kaca mata",
  }

  productRepository.Mock.On("FindById", "2").Return(product)

  result, err := productService.GetOneProduct("2")

  assert.Nil(t, err)

  assert.NotNil(t, result)

  assert.Equal(t, product.Id, result.Id, "result has to be '2'")
  assert.Equal(t, product.Name, result.Name, "result has to be 'Kaca mata'")
  assert.Equal(t, &product, result, "result has to be a product data with id '2'")
}
```