



Go-Lang Unit Tests

Eco Kurniawan Khannedy

Eco Kurniawan Khannedy

- Technical architect at one of the biggest e-commerce companies in Indonesia
- 10+ years experience
- youtube.com/c/ProgrammerZamanNow





Eco Kurniawan Khannedy

- Telegram : @khannedy
- Facebook : fb.com/ProgrammerZamanNow
- Instagram: instagram.com/programmerzamannow
- Youtube: youtube.com/c/ProgrammerZamanNow
- Telegram Channel: <https://t.me/ProgrammerZamanNow>
- Email : echo.khannedy@gmail.com



Before Studying

- Basic Go-Lang
- Go-Lang Modules



Agenda

- Introduction to Software Testing
- testing Package
- Unit Tests
- Assertion
- Mock, and
- Benchmarks

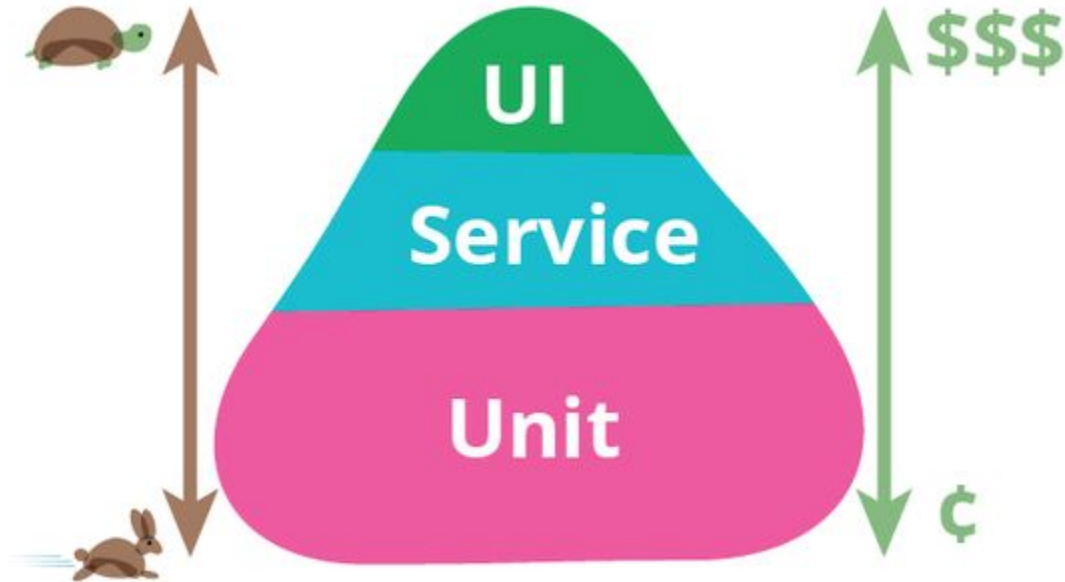
Introduction to Software Testing



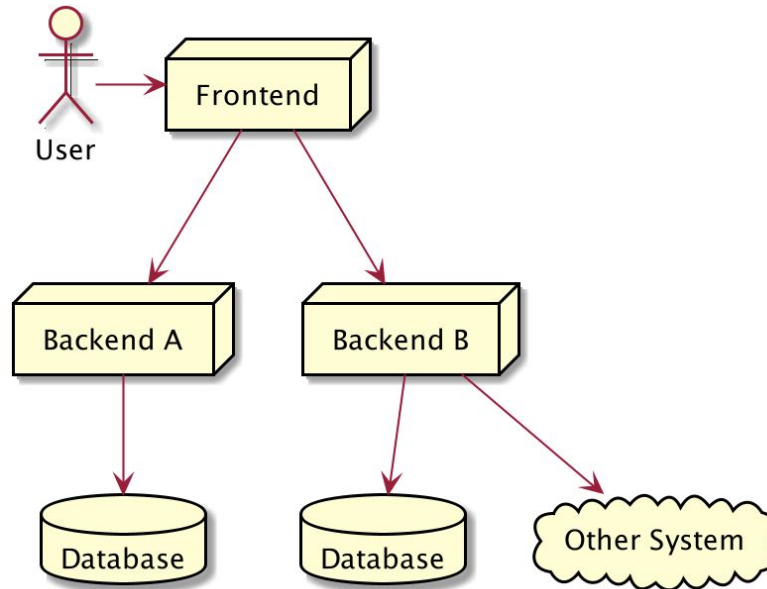
Introduction to Software Testing

- Software testing is one of the scientific disciplines in software engineering
- The main goal of software testing is to ensure the quality of our code and applications is good
- The knowledge of software testing itself is very broad, in this material we will only focus on unit testing

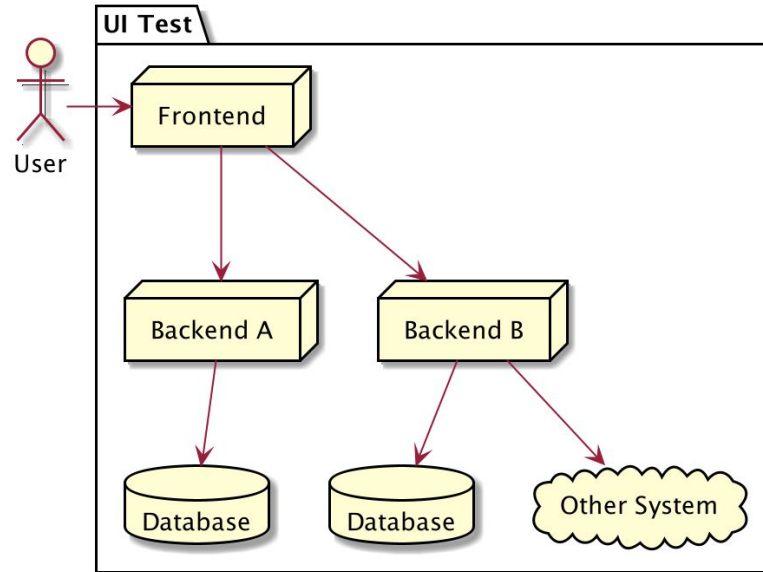
Pyramid Test



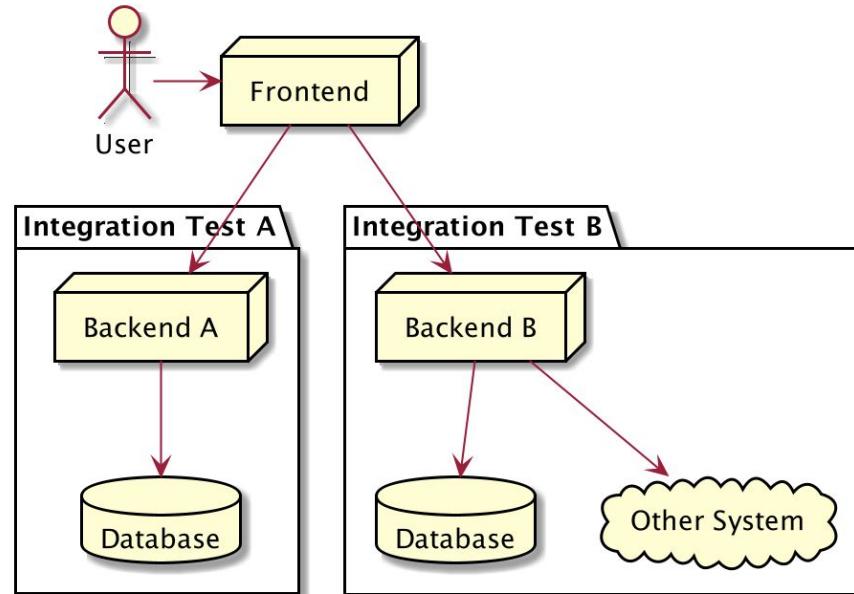
Example of High Level Application Architecture



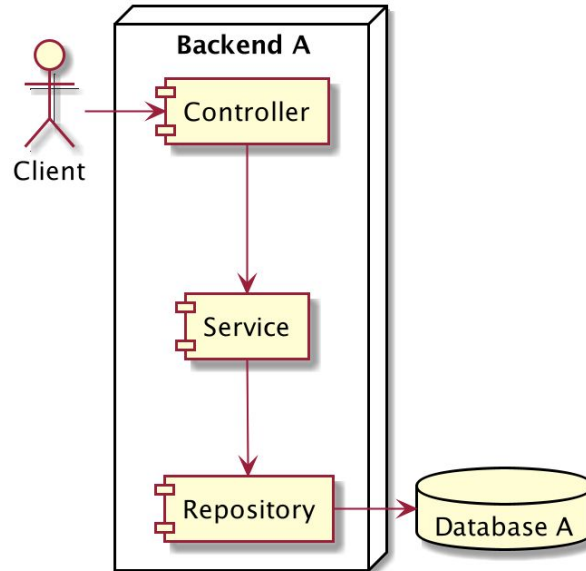
UI Test / End to End Test



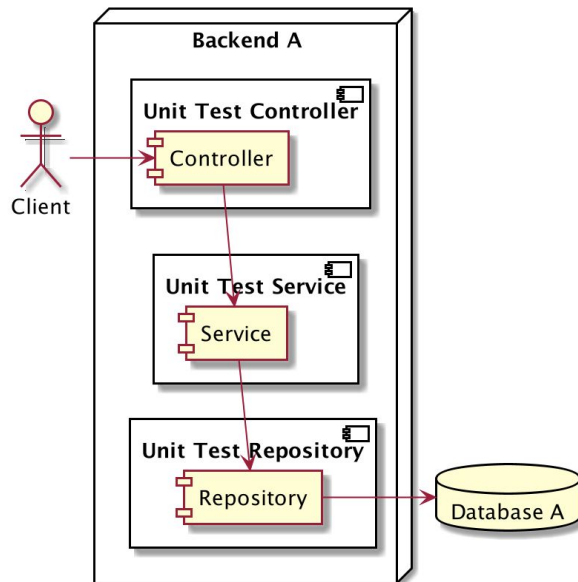
Service Test / Integration Test



Example of Application Internal Architecture



Unit Tests





Unit Tests

- Unit tests will focus on testing the smallest part of program code, usually testing a method
- Unit tests are usually made small and fast, therefore sometimes there is more unit test code than the original program code, because all test scenarios will be tried in the unit test
- Unit tests can be used as a way to improve the quality of our program code

Introduction to testing packages



testing Package

- In other programming languages, usually to implement unit tests, we need a special library or framework for unit tests
- In contrast to Go-Lang, in Go-Lang, a special package called testing has been provided for unit tests
- Apart from that, to run unit tests, Go-Lang also provides commands
- This makes implementing unit testing in Golang very easy compared to other programming languages
- <https://golang.org/pkg/testing/>



testing.T

- Go-Lang provides a struct called testing.T
- This struct is used for unit tests in Go-Lang



testing.M

- testing.M is a struct provided by Go-Lang to manage the testing life cycle
- We will discuss this material later in the Main chapter



testing.B

- testing.B is a struct provided by Go-Lang for benchmarking
- In Go-Lang, benchmarking (measuring the speed of program code) is also provided, so we don't need to use additional libraries or frameworks.

Create Unit Tests



Code: Hello World Function

```
hello_world.go x
1 package helper
2
3 func HelloWorld(name string) string {
4     return "Hello " + name
5 }
6
```



Test File Rules

- Go-Lang has rules for creating special files for unit tests
- To create a unit test file, we must use the `_test` suffix
- So, for example, we create the `hello_world.go` file, meaning that to create the unit test, we have to create the `hello_world_test.go` file



Function Unit Test Rules

- Apart from file name rules, Go-Lang also has settings for unit test function names
- Function names for unit tests must begin with the name Test
- For example, if we want to test the HelloWorld function, we will create a unit test function with the name TestHelloWorld
- There is no rule that the last name of the unit test function must be the same as the name of the function to be tested, the important thing is that it must start with the word Test
- Next it must have a parameter (t *testing.T) and not return a return value

Code: Hello World Unit Test

```
hello_world_test.go x
1  >> package helper
2
3  import "testing"
4
5  func TestHelloWorld(t *testing.T) {
6      result := HelloWorld("Eko")
7      if result != "Hello Eko" {
8          // unit test failed
9          panic("Result is not Hello Eko")
10     }
11 }
12
```




Running Unit Tests

- To run unit tests we can use the command: `go test`
- If we want to see in more detail what function tests have been run, we can use the command:
`go test -v`
- And if we want to choose which unit test function we want to run, we can use the command:

`go test -v -run TestFunctionName`



Running All Unit Tests

- If we want to run all unit tests from the top module folder, we can use the command: `go test ./...`

Failed the Test



Failing Unit Tests

- Failing unit tests using panic is not a good thing
- Go-Lang itself has provided a way to fail unit tests using testing.T
- There are functions Fail(), FailNow(), Error() and Fatal() if we want to fail the unit test



t.Fail() and t.FailNow()

- There are two functions to fail unit tests, namely Fail() and FailNow(). So what's the difference?
- Fail() will fail the unit test, but continue executing the unit test. However, when it is finished, the unit test is deemed to have failed
- FailNow() will fail the unit test at this time, without continuing to execute the unit test



t.Error(args...) and t.Fatal(args...)

- Besides Fail() and FailNow(), there are also Error() and Fatal()
- The Error() function is more like logging (printing) errors, but after logging the error, it will automatically call the Fail() function, resulting in the unit test being considered failed.
- However, because it only calls Fail(), it means the unit test execution will continue to run until it finishes
- Fatal() is similar to Error(), only after logging the error, it calls FailNow(), causing unit test execution to stop.



Code: Error

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



Code: Fatal

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

Assertion



Assertion

- Carrying out checks in unit tests manually using if else is very annoying
- Especially if there is a lot of result data that needs to be checked
- Therefore, it is highly recommended to use Assertion to carry out checks
- Unfortunately, Go-Lang does not provide a package for assertions, so we need to add a library to perform this assertion



Testify

- One of the most popular assertion libraries in Go-Lang is Testify
- We can use this library to perform assertions on result data in unit tests
- <https://github.com/stretchr/testify>
- We can add it to our Go module: go get github.com/stretchr/testify



Code: Assertion

```
import (  
    "fmt"  
    "github.com/stretchr/testify/assert"  
    "testing"  
)  
  
func TestHelloWorldAssertion(t *testing.T) {  
    result := HelloWorld("Eko")  
    assert.Equal(t, "Hello Eko", result)  
    fmt.Println("Dieksekusi")  
}
```



assert vs require

- Testify provides two packages for assertion, namely assert and require
- When we use assert, if the check fails, then assert will call Fail(), meaning that unit test execution will continue
- Meanwhile, if we use require, if the check fails, then require will call FailNow(), meaning that unit test execution will not continue

Code: Require

```
"github.com/stretchr/testify/require"  
"testing"
```

```
)
```

```
func TestHelloWorldRequire(t *testing.T) {  
    result := HelloWorld("Eko")  
    require.Equal(t, "Hello Eko", result)  
    fmt.Println("Tidak Dieksekusi")  
}
```

Skip Test



Skip Test

- Sometimes under certain circumstances, we want to cancel unit test execution
- In Go-Lang we can also cancel unit test execution if we want
- To cancel a unit test we can use the `Skip()` function



Code: Skip Test

```
func TestSkip(t *testing.T) {  
    if runtime.GOOS == "darwin" {  
        t.Skip("Unit test tidak bisa jalan di Mac")  
    }  
  
    result := HelloWorld("Eko")  
    require.Equal(t, "Hello Eko", result)  
}
```

Before and After Tests



Before and After Tests

- Usually in unit tests, sometimes we want to do something before and after a unit test is executed
- If the code we do before and after is always the same between unit test functions, then manually creating the unit test function is tedious and results in too much duplicate code.
- Luckily in Go-Lang there is a feature called `testing.M`
- This feature is called `Main`, which is used to manage the execution of unit tests, but we can also use this to do Before and After in unit tests



testing.M

- To manage unit test execution, we simply create a function called TestMain with the parameter testing.M
- If there is a TestMain function, then Go-Lang will automatically execute this function every time it runs a unit test on a package
- With this we can set the Before and After unit tests according to what we want
- Remember, the TestMain function is executed only once per Go-Lang package, not per each unit test function



Code: TestMain

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

Sub Test



Sub Test

- Go-Lang supports the function unit test creation feature within the unit test function
- This feature is a little strange and is rarely found in unit tests in other programming languages
- To create a sub test, we can use the Run() function

Code: Create a Sub Test

```
func TestSubTest(t *testing.T) {  
    t.Run("Eko", func(t *testing.T) {  
        result := HelloWorld("Eko")  
        require.Equal(t, "Hello Eko", result)  
    })  
    t.Run("Kurniawan", func(t *testing.T) {  
        result := HelloWorld("Kurniawan")  
        require.Equal(t, "Hello Kurniawan", result)  
    })  
}
```




Running Only Sub Tests

- We already know that if we want to run a unit test function, we can use the command: `go test -run TestNamaFunction`
- If we want to run only one of the sub tests, we can use the command: `go test -run TestNamaFunction/NamaSubTest`
- Or for all tests, all sub tests in all functions, we can use the command: `go test -run /NamaSubTest`

Table Test



Table Test

- Previously we learned about sub tests
- If you pay attention, actually with sub tests, we can create tests dynamically
- And this sub test feature is usually used by Go-Lang programmers to create tests with the table test concept
- Table test is where we provide data in the form of slices containing parameters and expected results from the unit test
- Then we iterate over the slice using sub tests

Code: Table Test

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

```
    {
        name:      "HelloWorld(Kurniawan)",
        request:    "Kurniawan",
        expected:   "Hello Kurniawan",
    },
}

for _, test := range tests {
    t.Run(test.name, func(t *testing.T) {
        result := HelloWorld(test.request)
        assert.Equal(t, test.expected, result)
    })
}
```

Mock



Mock

- Mock is an object that we have programmed with certain expectations so that when called, it will produce the data that we programmed at the beginning
- Mock is a technique in unit testing, where we can create a mock object from an object that is difficult to test
- For example, we want to make a unit test, but it turns out that our program code has to make an API call to a third party service. This is very difficult to test, because our unit testing must always call third party service, and the response may not necessarily match what we want.
- In cases like this, it is very suitable to use mock objects



Testify Mock


- To create mock objects, there are no built-in Go-Lang features, but we can use the assertion library that we previously used for assertions
- Testify supports creating mock objects, so it is suitable for us to use when we want to create mock objects
- However, please note, if our program code design is bad, it will be difficult to do mocking, so make sure we design our program code well.
- Let's make an example case



Query Application to Database

- We will try an example case by creating an example Golang application that performs queries to the database
- Where we will create the Service layer as business logic, and the Repository layer as a bridge to the database
- So that our code is easy to test, it is recommended to create a contract in the form of an interface

Code: Category Entity

 category.go ×

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

Code: Category Repository

```
category_repository.go x
1 package repository
2
3 import "belajar-golang-unit-test/entity"
4
5 type CategoryRepository interface {
6     FindById(id string) *entity.Category
7 }
8
```



Code: Category Service

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



Code: Category Repository Mock

```
type CategoryRepositoryMock struct {  
    Mock mock.Mock  
}  
  
func (repository *CategoryRepositoryMock) FindById(id string) *entity.Category {  
    arguments := repository.Mock.Called(id)  
    if arguments.Get(0) == nil {  
        return nil  
    }  
    category := arguments.Get(0).(entity.Category)  
    return &category  
}
```



Code: Category Service Unit Test (1)

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

func TestCategoryService_GetNotFound(t *testing.T) {
    categoryRepository.Mock.On("FindById", "1").Return(nil)
    category, err := categoryService.Get("1")
    assert.NotNil(t, err)
    assert.Nil(t, category)
}
```



Code: Category Service Unit Test (2)

```
func TestCategoryService_GetFound(t *testing.T) {  
    category := entity.Category{  
        Id:    "2",  
        Name:  "Handphone",  
    }  
    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)  
}
```

Benchmarks



Benchmarks

- Apart from unit tests, the Go-Lang testing package also supports benchmarking
- Benchmarks are a mechanism for calculating the performance speed of our application code
- Benchmarking in Go-Lang is done by automatically iterating the code that we call many times until a certain time
- We don't need to determine the number of iterations and their duration, because that is already set by the default testing.B from the testing package



testing.B

- testing.B is the struct used to benchmark.
- testing.B is similar to testing.T, there are functions Fail(), FailNow(), Error(), Fatal() and others
- What is different is that there are several additional attributes and functions used to benchmark
- One of them is the N attribute, this is used to carry out the total iterations of a benchmark



How Benchmarks Work

- The way benchmarks work in Go-Lang is very simple
- How about we only need to repeat a number of N attributes
- Later Go-Lang will automatically execute a specified number of iterations automatically, then detect how long the process has been running, and conclude the benchmark performance in time.

Creating Benchmarks



Benchmark Function

- Similar to unit tests, even for benchmarks, in Go-Lang the function name has been determined, it must begin with the word Benchmark, for example BenchmarkHelloWorld, BenchmarkXxx
- Additionally, it must have a parameter (b *testing.B)
- And it cannot return a return value
- For benchmark file names, the same as unit tests, end with _test, for example hello_world_test.go



Code: Create a Benchmark Function

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



Running Benchmarks

- To run all the benchmarks in the module, we can use the same command as test, but add the bench parameter:
`go test -v -bench=.`
- If we just want to run the benchmark without unit tests, we can use the command: `go test -v -run=NotMathUnitTest -bench=.`
- The code above, apart from running benchmarks, will also run unit tests. If we only want to run certain benchmarks, we can use the command:
`go test -v -run=NotMathUnitTest -bench=BenchmarkTest`
- If we run a benchmark on the root module and want all modules to run, we can use command:
`go test -v -bench=. ./...`

Sub Benchmarks



Sub Benchmarks

- Just like `testing.T`, in `testing.B` we can also create sub benchmarks using the `Run()` function



Code: Creating Sub Benchmarks

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



Running Only Sub Benchmarks

- When we run the benchmark function, all sub benchmarks will run
- However, if we want to run just one of the sub benchmarks, we can use the command: `go test -v -bench=BenchmarkName/SubName`

Table Benchmarks



Table Benchmarks

- Just like in unit tests, Go-Lang programmers are used to creating benchmark tables too
- This is used so that we can easily carry out performance tests with different data combinations without having to create many benchmark functions

Code: Create Benchmark Table

```
func BenchmarkHelloWorldTable(b *testing.B) {  
    benchmarks := []struct {  
        name    string  
        request string  
    }{  
        {  
            name:    "HelloWorld(Eko)",  
            request: "Eko",  
        },  
        {  
            name:    "HelloWorld(Khannedy)",  
            request: "Khannedy",  
        },  
    }  
}
```

```
for _, benchmark := range benchmarks {  
    b.Run(benchmark.name, func(b *testing.B) {  
        for i := 0; i < b.N; i++ {  
            HelloWorld(benchmark.name)  
        }  
    })  
}
```

Next Material



Next Material

- Go-Lang Goroutines
- Go-Lang Database
- Go-Lang Web