Testing

In Go, unit testing is a crucial part of software development, and it involves testing a specific piece of code in a program or package. The purpose of unit tests is to check the correctness of an application. On the other hand, mocking is a way of creating a stub or fake piece of code that substitutes some functions of the whole code. Its purpose is to make testing easier because you can control those stub functions’ inputs and outputs freely, which allows you to focus only the business logic.

In Go, the difference between mock test and unit test is that a unit test is testing the correctness of a specific piece of code, while a mock test is testing the interaction between different components of the code. In other words, a unit test is testing the business logic of the code, while a mock test is testing the integration between different parts of the code.

To implement mocking in Go, there are different methods to follow, and two of the most common methods are as follows:

* Pass the dependency as a parameter of the function to be tested.
* Make the function to be tested a method of a type, and the type can hold the dependency.

For the first method, you define a function that takes the dependency as a parameter, and then you pass the function as a parameter to the function you want to test. In the test, you can then mock the function to return the values you want. For the second method, you create a type that holds the dependency and define a function as a method of the type. In the test, you can then create an instance of the type and pass the mock dependency to it.

Here is an example of how to implement the first method:

type PageGetter func(url string) string

func downloader(pageGetterFunc PageGetter) {

    // ...

    content := pageGetterFunc(BASE\_URL)

    // ...

}

func get\_page(url string) string { /\* ... \*/ }

func mock\_get\_page(url string) string {

    // mock your 'get\_page()' function here

}

func TestDownloader(t \*testing.T) {

    downloader(mock\_get\_page)

}

Here is an example of how to implement the second method:

type PageGetter func(url string) string

type Downloader struct {

    get\_page PageGetter

}

func NewDownloader(pg PageGetter) \*Downloader {

    return &Downloader{get\_page: pg}

}

func (d \*Downloader) download() {

    //...

    content := d.get\_page(BASE\_URL)

    //...

}

func mock\_get\_page(url string) string {

    // mock your 'get\_page()' function here

}

func TestDownloader() {

    d := NewDownloader(mock\_get\_page)

    d.download()

}

1. Unit Testing

a. What is unit testing?

Testing is part of the software development process and the purpose is to produce better software, more robust, with fewer bugs, and more stable. There are different kinds of testing that can be done to improve software quality. Starting with unit testing, integration testing, end to end testing, and we will focus on unit testin.



In the software testing pyramid image above, we can see that unit testing is cheaper and faster to do. That is why we should cover unit test as much as possible to reduce the testing need to be done on integration and end-to-end testing.

b. How unit testing can improve software quality

Unit testing can help us verify code behavior at the function/method level. For example, if we have an add function that takes 2 params we expect the return to be the addition of both params. For this, we can unit test the method by providing some test cases that verify the function behavior. The expectation is that if every unit is working properly proven by its test cases then the overall software behavior should also work as intended. Another advantage of unit testing is we will have more confidence in refactoring our code since as long as all UTs pass then we have confidence that the behavior of the code is still the same and should work fine. This is important since we need to keep improving our code and we should have confidence that we are not breaking anything in doing so. But of course, we should define our UT test cases properly that cover all conditions and edge cases.

c. Unit test coverage

There is a metric to measure how much percent of our code base is covered with unit testing. The metric name is Unit Test Coverage. In Halodoc we set a really high number for our unit test coverage, it's more than 95% for overall code base Unit Test Coverage and more than 95% for new code to be merged to the main branch (deployment). If the coverage condition is not met new code could not be deployed to production. Halodoc enforces this condition to force the implementation of Unit Testing and get the benefit from it. In GoLang, Unit Test Coverage is calculated from how many lines of code are executed when running testing. For certain packages that do not require unit testing, we can ignore them so that the package does not drop the coverage. We could not blindly believe that higher Unit Test Coverage means better software quality since being said earlier, Unit Test Coverage only calculates lines of code that are executed on running tests.