Unit testing in Golang

Unit testing is a software testing technique where individual components or units of a software application are tested in isolation from the rest of the application. The primary goal of unit testing is to validate that each unit of the software performs as expected.

By default go comes with all the tools to write, run and debug a test. Lets take a look at simple test case below

```go
func sum(a, b int) (int, error) {
    return a + b, nil
}

func Test_sum(t *testing.T) {
    expected := 10
    wantErr := false
    actual, err := sum(3, 7)
    if (err != nil) != wantErr {
        t.Errorf("error = %v, wantErr %v", err, wantErr)
        return
    }
    if actual != expected {
        t.Errorf("got %d, want %d", actual, expected)
        return
    }
}
```

Basic thumb rule for unit testing is that we set some expectations before running the function that needs to be tested and after execution we verified whether our expectations are met or not.

In above example we have set 2 expectations **expectedSum** and whether an **error** is expected or not

Note: It's not necessary that expectation always need to be positive / happy case, it can be negative as well.

Writing Table-Driven Tests

When writing tests, you may find yourself repeating a lot of code in order to cover all the cases required. You could write one test function per case, but this would lead to a lot of duplication. You also need to call the tested function several times and validate the output which is tiring and cumbersome process.

To solved all the issues and make your tests more readable we should follow table driven approach. Let's refactor the above test case into a table driven format.

```go
func Test_sum(t *testing.T) {
    type args struct {
        a int
        b int
    }
    tests := []struct {
        name    string
        args    args
        want    int
        wantErr bool
    }{
        {
            name: "1: positive",
            args: args{
                a: 1,
                b: 2,
            },
            want: 3,
        },
        {
            name: "2: negative",
            args: args{
                a: 1,
                b: 1,
            },
            wantErr: true,
        },
    }
    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            got, err := sum(tt.args.a, tt.args.b)
            if (err != nil) != tt.wantErr {
                t.Errorf("sum() error = %v, wantErr %v", err, tt.wantErr)
                return
            }
            if got != tt.want {
                t.Errorf("sum() got = %v, want %v", got, tt.want)
            }
        })
    }
}
```

Mocking

When writing unit tests, you often encounter situations where the function under test makes I/O calls, such as network requests or file system operations. These I/O calls can introduce several issues:

1. **Dependency**: The test becomes dependent on external systems.
2. **Non-Isolation**: The test is no longer isolated, as it relies on external factors.
3. **Slowness**: I/O operations can be slow, making your tests take longer to run.

Mocking is the answer to these problems. By using mocks, you can simulate the behaviour of these I/O calls.

How to apply mocks in Go?

If you come for other from other object oriented languages such as java / python you are familiar with mocking frameworks such as **mockito / unitest** which can easily mock the entire object and methods but **thats not the case with golang**.

Mocking simple functions

```go
func addUserToDb(user User) error{
    return nil
}

func addUser(name string)  error {
    user := User{name}
    return addUserToDb(user)
}
```

In this example we will start writing unit test for **addUser().** As you can see this function is making an **i / o** call `addUserToDb` We'll see what are different ways of mocking this function

1: Variable Substitution: In this technique we'll do assign function to a variable and change the implementation for each test case

```go
var addUserToDb = func(user User) error {
    return nil
}

func Test_addUser(t *testing.T) {
    type args struct {
        name string
    }
    tests := []struct {
        name    string
        args    args
        wantErr bool
        mock    func()
    }{
        {
            name: "1: error while saving user to db",
            args: args{
                name: "shubham",
            },
            wantErr: true,
            mock: func() {
                addUserToDb = func(user User) error {
                    return errors.New("error while saving user to db")
                }
            },
        },
        {
            name: "2: positive",
            args: args{
                name: "shubham",
            },
            wantErr: false,
            mock: func() {
                addUserToDb = func(user User) error {
                    return nil
                }
            },
        },
    }

    originalAddUserToDb := addUserToDb
    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            tt.mock()
            if err := addUser(tt.args.name); (err != nil) != tt.wantErr {
                t.Errorf("addUser() error = %v, wantErr %v", err, tt.wantErr)
            }
        })
        addUserToDb = originalAddUserToDb
    }
}

```

Pros:

- Minimum to no code-changes

Cons:

- As we have introduced a global variable, test cases are no longer thread safe

2: Parameter substitution: As go allows to pass functions as parameter, we can take advantage of that

```go
func addUser(name string, addUserToDb func(user User) error) error {
    user := User{name}
    return addUserToDb(user)
}

func Test_addUser(t *testing.T) {
    type args struct {
        name        string
        addUserToDb func(user User) error
    }
    tests := []struct {
        name    string
        args    args
        wantErr bool
    }{
        {
            name: "1: error while saving user to db",
            args: args{
                name: "shubham",
                addUserToDb: func(user User) error {
                    return errors.New("error while saving user to db")
                },
            },
            wantErr: true,
        },
        {
            name: "2: positive",
            args: args{
                name: "shubham",
                addUserToDb: func(user User) error {
                    return nil
                },
            },
            wantErr: false,
        },
    }

    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            if err := addUser(tt.args.name, tt.args.addUserToDb); (err != nil) != tt.wantErr {
                t.Errorf("addUser() error = %v, wantErr %v", err, tt.wantErr)
            }
        })
    }
}

```

Pros: Very minimal code change, Thread Safe

Cons: Suppose the function under test call around 5-6 functions that make i/o calls. Passing each function as parameter makes the code more verbose and very difficult to manage.

3: Xgo mocking: We can use a third party package known as xgo that allows us to define mocks for functions without modifying the source code.

```go

func addUserToDb(user User) error {
    return nil
}

func addUser(name string) error {
    user := User{name}
    return addUserToDb(user)
}

func Test_addUser(t *testing.T) {
    type args struct {
        name string
    }
    tests := []struct {
        name    string
        args    args
        wantErr bool
        mock    func()
    }{
        {
            name: "1: error while saving user",
            args: args{
                name: "shubham",
            },
            wantErr: true,
            mock: func() {
                mock.Patch(addUserToDb, func(user User) error {
                    return errors.New("error while saving to db")
                })
            },
        },
        {
            name: "2: positive",
            args: args{
                name: "shubham",
            },
            wantErr: false,
            mock: func() {
                mock.Patch(addUserToDb, func(user User) error {
                    return nil
                })
            },
        },
    }
    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            tt.mock()
            if err := addUser(tt.args.name); (err != nil) != tt.wantErr {
                t.Errorf("addUser() error = %v, wantErr %v", err, tt.wantErr)
            }
        })
    }
}

```

Pros: No code changes, thread safe, supports for all architectures.

4: Dependency Injection: Instead of calling functions directly we can create an interface that covers the all the methods and replace the implementation at runtime. We'll discuss this more when we cover mocking on type functions

Pros: Best way to structure your code

Cons:

- Makes code verbose and takes away the advantage of function oriented benefits of golang.
- Creating a new interface just for the sake of mocking is completely useless and does't provide any benefits

Out of all the discusses approached **Xgo** is the winner. But there are some use cases where dependency injection makes more sense such as repository pattern

Mocking Type Functions:

1: **Xgo**

```go
type User struct {
    name string
}

type UserRepository struct {}

func (UserRepository) addUserToDb(user User) error {
    return nil
}

func addUser(name string) error {
    user := User{name}
    repo := UserRepository{}
    return repo.addUserToDb(user)
}

func Test_addUser(t *testing.T) {
    type args struct {
        name string
    }
    tests := []struct {
        name    string
        args    args
        wantErr bool
        mock    func()
    }{
        {
            name: "1: error while adding user",
            args: args{
                name: "test",
            },
            wantErr: true,
            mock: func() {
                mock.Patch((*UserRepository).addUserToDb, func(_ *UserRepository, user User) error {
                    return errors.New("errored")
                })
            },
        },
        {
            name: "2: positive",
            args: args{
                name: "test",
            },
            wantErr: false,
            mock: func() {
                mock.Patch((*UserRepository).addUserToDb, func(_ *UserRepository, user User) error {
                    return nil
                })
            },
        },
    }
    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            tt.mock()
            if err := addUser(tt.args.name); (err != nil) != tt.wantErr {
                t.Errorf("addUser() error = %v, wantErr %v", err, tt.wantErr)
            }
        })
    }
}

```

2: Dependency Injection: Instead of having a concrete type we'll create an interface, pass it everywhere. As we are using interface we can change the implementation at runtime with a mocked one.

```go
type User struct {
    name string
}

type UserRepository interface {
    addUserToDb(user User) error
}

type UserService struct {
    UserRepo UserRepository
}

type UserRepositoryImpl struct{}

func (UserRepositoryImpl) addUserToDb(user User) error {
    return nil
}

func (s UserService) addUser(name string) error {
    user := User{name}
    return s.UserRepo.addUserToDb(user)
}


type MockUserRepoIml struct {
    mock.Mock
}
```

```go

func (m *MockUserRepoIml) addUserToDb(user User) error {
    args := m.Called(user)
    return args.Error(0)
}

func TestUserService_addUser(t *testing.T) {
    type fields struct {
        UserRepo *MockUserRepoIml
    }
    type args struct {
        name string
    }
    tests := []struct {
        name    string
        fields  fields
        args    args
        wantErr bool
        mock    func(f fields)
    }{
        {
            name: "1: error while adding user",
            fields: fields{
                UserRepo: &MockUserRepoIml{},
            },
            args: args{
                name: "shubham",
            },
            wantErr: true,
            mock: func(f fields) {
                f.UserRepo.On("addUserToDb", mock.AnythingOfType("User")).Return(errors.New("error while adding
            },
        },
        {
            name: "2: positive",
            fields: fields{
                UserRepo: &MockUserRepoIml{},
            },
            args: args{
                name: "shubham",
            },
            wantErr: false,
            mock: func(f fields) {
                f.UserRepo.On("addUserToDb", mock.AnythingOfType("User")).Return(nil)
            },
        },
    }
    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            tt.mock(tt.fields)
            s := UserService{
                UserRepo: tt.fields.UserRepo,
            }
            if err := s.addUser(tt.args.name); (err != nil) != tt.wantErr {
                t.Errorf("addUser() error = %v, wantErr %v", err, tt.wantErr)
            }

            tt.fields.UserRepo.AssertExpectations(t)
        })
    }
}

```

From above examples its clear that dependency injections is more verbose but provide better controls such as argument type and much more. So according to me a combination of both approaches is way to go forward. For simple types we should prefer using **xgo** while for methods interacting with the databases, refactoring your code into interfaces makes much more sense.

Mocking databases:

We can use both the methods mentioned above to mock the methods that interact with the database layer but for some database drivers we can skip both methods. Example are given below:

- Sql Dbs: GitHub - DATA-DOG/go-sqlmock: Sql mock driver for golang to test database interactions
- Mongodb: mongo-go-driver/mongo/integration/mtest at v1 · mongodb/mongo-go-driver
- Redis: GitHub - go-redis/redismock: Redis client Mock

These libraries provides the mocked database object so the user don't have to. But if there is **no implementation** provided by library owner **we need to use either xgo's or interface segregation approach with testify.**

Mocking Sql Dbs:

```go
type User struct {
    Id   int    `gorm:"column:id;primaryKey"`
    Name string `gorm:"column:name"`
}

type UserService struct {
    UserRepo UserRepository
}

type UserRepository struct {
    db *gorm.DB
}

func (repo UserRepository) addUserToDb(user User) error {
    result := repo.db.Create(&user)
    return result.Error
}

func (repo UserRepository) getUser(id int) error {
    var user User
    result := repo.db.First(&user, id)
    return result.Error
}

func (repo UserRepository) deleteUser(id int) error {
    result := repo.db.Delete(&User{}, id)
    return result.Error
}

func (repo UserRepository) updateUser(id int, name string) error {
    return repo.db.Model(&User{}).Where("id = ?", id).Update("name", name).Error
}
```

In this example I am writing test cases for basic crud operations that uses gorm for interacting with db.

Working with **sqlx** will be almost similar instead of *gorm.DB we return *sqlx.DB)

1: Test Create

```go
func TestUserRepository_addUserToDb(t *testing.T) {
    type fields struct {
        db *gorm.DB
    }
    type args struct {
        user User
    }
    tests := []struct {
        name    string
        fields  fields
        args    args
        wantErr bool
        mock    func(a args) (*gorm.DB, sqlmock.Sqlmock)
    }{
        {
            name: "1: error while adding user",
            args: args{
                user: User{
                    Id:   1,
                    Name: "shubham",
                },
            },
            wantErr: true,
            mock: func(a args) (*gorm.DB, sqlmock.Sqlmock) {
                db, mock, err := sqlmock.New()
                if err != nil {
                    log.Fatalf("An error '%s' was not expected when opening a stub database connection", err)
                }

                gormDB, err := gorm.Open(mysql.New(mysql.Config{
                    Conn:                      db,
                    SkipInitializeWithVersion: true,
                }), &gorm.Config{})

                mock.ExpectBegin()
                mock.ExpectExec(regexp.QuoteMeta("INSERT INTO `users` (`name`,`id`) VALUES (?,?)")).
                    WithArgs("shubham", 1).
                    WillReturnError(errors.New("error"))
                return gormDB, mock
            },
        },
        {
            name: "2: success",
            args: args{
                user: User{
                    Id:   1,
                    Name: "shubham",
                },
            },
            wantErr: false,
            mock: func(a args) (*gorm.DB, sqlmock.Sqlmock) {
                db, mock, err := sqlmock.New()
                if err != nil {
```

```go
				log.Fatalf("An error '%s' was not expected when opening a stub database connection", err)
			}

			gormDB, err := gorm.Open(mysql.New(mysql.Config{
				Conn:                      db,
				SkipInitializeWithVersion: true,
			}), &gorm.Config{})

			mock.ExpectBegin()
			mock.ExpectExec(regexp.QuoteMeta("INSERT INTO `users` (`name`,`id`) VALUES (?,?)")).
				WithArgs("shubham", 1).
				WillReturnResult(sqlmock.NewResult(1, 1))

			mock.ExpectCommit()
			return gormDB, mock
		},
	},
}
for _, tt := range tests {
	t.Run(tt.name, func(t *testing.T) {
		db, mock := tt.mock(tt.args)
		repo := UserRepository{
			db: db,
		}
		if err := repo.addUserToDb(tt.args.user); (err != nil) != tt.wantErr {
			t.Errorf("addUserToDb() error = %v, wantErr %v", err, tt.wantErr)
		}

		if mock != nil {
			if err := mock.ExpectationsWereMet(); err != nil {
				t.Errorf("there were unfulfilled expectations: %s", err)
			}
		}
	})
}
}
```

2: Test Get

```go
func TestUserRepository_getUser(t *testing.T) {
	type fields struct {
		db *gorm.DB
	}
	type args struct {
		id int
	}
	tests := []struct {
		name    string
		fields  fields
		args    args
		wantErr bool
		mock    func(a args) (*gorm.DB, sqlmock.Sqlmock)
	}{
		{
			name: "1: not found",
			args: args{
				id: 1,
			},
			wantErr: true,
            mock: func(a args) (*gorm.DB, sqlmock.Sqlmock) {
                db, mock, err := sqlmock.New()
                if err != nil {
                    log.Fatalf("An error '%s' was not expected when opening a stub database connection", err)
                }

                gormDB, err := gorm.Open(mysql.New(mysql.Config{
                    Conn:                      db,
                    SkipInitializeWithVersion: true,
                }), &gorm.Config{})

                mock.ExpectQuery(regexp.QuoteMeta("SELECT * FROM `users` WHERE `users`.`id` = ? ORDER BY `users`
                    WithArgs(1, 1).
                    WillReturnError(errors.New("error"))
                return gormDB, mock
            },
        },
        {
            name: "2: success",
            args: args{
                id: 1,
            },
            wantErr: false,
            mock: func(a args) (*gorm.DB, sqlmock.Sqlmock) {
                db, mock, err := sqlmock.New()
                if err != nil {
                    log.Fatalf("An error '%s' was not expected when opening a stub database connection", err)
                }

                gormDB, err := gorm.Open(mysql.New(mysql.Config{
                    Conn:                      db,
                    SkipInitializeWithVersion: true,
                }), &gorm.Config{})

                mockedRows := sqlmock.NewRows([]string{"id", "name"}).
                    AddRow("1", "shubham")

                mock.ExpectQuery(regexp.QuoteMeta("SELECT * FROM `users` WHERE `users`.`id` = ? ORDER BY `users`
                    WithArgs(1, 1).
                    WillReturnRows(mockedRows)
                return gormDB, mock
            },
        },
    }
    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            db, mock := tt.mock(tt.args)

            repo := UserRepository{
                db: db,
            }
            if err := repo.getUser(tt.args.id); (err != nil) != tt.wantErr {
                t.Errorf("getUser() error = %v, wantErr %v", err, tt.wantErr)
            }

            if mock != nil {
                if err := mock.ExpectationsWereMet(); err != nil {
                    t.Errorf("there were unfulfilled expectations: %s", err)
                }
            }
        })
    }
}
```

3: Test Delete

```go
func TestUserRepository_deleteUser(t *testing.T) {
    type fields struct {
        db *gorm.DB
    }
    type args struct {
        id int
    }
    tests := []struct {
        name    string
        fields  fields
        args    args
        wantErr bool
        mock    func(a args) (*gorm.DB, sqlmock.Sqlmock)
    }{
        {
            name: "1: error while deleting user",
            args: args{
                id: 1,
            },
            wantErr: true,
            mock: func(a args) (*gorm.DB, sqlmock.Sqlmock) {
                db, mock, err := sqlmock.New()
                if err != nil {
                    log.Fatalf("An error '%s' was not expected when opening a stub database connection", err)
                }

                gormDB, err := gorm.Open(mysql.New(mysql.Config{
                    Conn:                      db,
                    SkipInitializeWithVersion: true,
                }), &gorm.Config{})

                mock.ExpectBegin()
                mock.ExpectExec(regexp.QuoteMeta("DELETE FROM `users` WHERE `users`.`id` = ?")).
                    WithArgs(1).
                    WillReturnError(errors.New("error"))
                return gormDB, mock
            },
        },
        {
            name: "2: success",
            args: args{
                id: 1,
            },
            wantErr: false,
            mock: func(a args) (*gorm.DB, sqlmock.Sqlmock) {
                db, mock, err := sqlmock.New()
                if err != nil {
                    log.Fatalf("An error '%s' was not expected when opening a stub database connection", err)
                }

                gormDB, err := gorm.Open(mysql.New(mysql.Config{
                    Conn:                      db,
                    SkipInitializeWithVersion: true,
                }), &gorm.Config{})

                mock.ExpectBegin()
                mock.ExpectExec(regexp.QuoteMeta("DELETE FROM `users` WHERE `users`.`id` = ?")).
                    WithArgs(1).
                    WillReturnResult(sqlmock.NewResult(1, 1))
                mock.ExpectCommit()
                return gormDB, mock
            },
        },
    }
    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            db, mock := tt.mock(tt.args)

            repo := UserRepository{
                db: db,
            }
            if err := repo.deleteUser(tt.args.id); (err != nil) != tt.wantErr {
                t.Errorf("deleteUser() error = %v, wantErr %v", err, tt.wantErr)
            }

            if mock != nil {
                if err := mock.ExpectationsWereMet(); err != nil {
                    t.Errorf("there were unfulfilled expectations: %s", err)
                }
            }
        })
    }
}
```

4: Test Update

```go
func TestUserRepository_updateUser(t *testing.T) {
    type fields struct {
        db *gorm.DB
    }
    type args struct {
        id   int
        name string
    }
    tests := []struct {
        name    string
        fields  fields
        args    args
        wantErr bool
        mock    func(a args) (*gorm.DB, sqlmock.Sqlmock)
    }{
        {
            name: "1: error while updating user",
            args: args{
                name: "go",
                id:   1,
            },
            wantErr: true,
            mock: func(a args) (*gorm.DB, sqlmock.Sqlmock) {
                db, mock, err := sqlmock.New()
				if err != nil {
					log.Fatalf("An error '%s' was not expected when opening a stub database connection", err)
				}

				gormDB, err := gorm.Open(mysql.New(mysql.Config{
					Conn:                      db,
					SkipInitializeWithVersion: true,
				}), &gorm.Config{})

				mock.ExpectBegin()
				mock.ExpectExec(regexp.QuoteMeta("UPDATE `users` SET `name`=? WHERE id = ?")).
					WithArgs("go", 1).
					WillReturnError(errors.New("error"))
				return gormDB, mock
			},
		},
		{
			name: "2: success",
			args: args{
				name: "go",
				id:   1,
			},
			wantErr: false,
			mock: func(a args) (*gorm.DB, sqlmock.Sqlmock) {
				db, mock, err := sqlmock.New()
				if err != nil {
					log.Fatalf("An error '%s' was not expected when opening a stub database connection", err)
				}

				gormDB, err := gorm.Open(mysql.New(mysql.Config{
					Conn:                      db,
					SkipInitializeWithVersion: true,
				}), &gorm.Config{})

				mock.ExpectBegin()
				mock.ExpectExec(regexp.QuoteMeta("UPDATE `users` SET `name`=? WHERE id = ?")).
					WithArgs("go", 1).
					WillReturnResult(sqlmock.NewResult(1, 1))
				mock.ExpectCommit()
				return gormDB, mock
			},
		},
	}
	for _, tt := range tests {
		t.Run(tt.name, func(t *testing.T) {
			db, mock := tt.mock(tt.args)

			repo := UserRepository{
				db: db,
			}
			if err := repo.updateUser(tt.args.id, tt.args.name); (err != nil) != tt.wantErr {
				t.Errorf("updateUser() error = %v, wantErr %v", err, tt.wantErr)
			}

			if mock != nil {
				if err := mock.ExpectationsWereMet(); err != nil {
					t.Errorf("there were unfulfilled expectations: %s", err)
				}
```

```go
            }
        })
    }
}

```

Unit Tests for MongoDB

```go

type User struct {
    Id   int    `bson:"_.id"`
    Name string `bson:"name"`
}

type UserService struct {
    UserRepo UserRepository
}

type UserRepository struct {
    db *mongo.Database
}

func (repo UserRepository) addUserToDb(user User) error {
    collection := repo.db.Collection("users")
    _, err := collection.InsertOne(context.Background(), user)
    return err
}

func (repo UserRepository) getUser(id int) error {
    var user User
    collection := repo.db.Collection("users")
    return collection.FindOne(context.TODO(), bson.D{{"_id", id}}).Decode(&user)
}

func (repo UserRepository) deleteUser(id int) error {
    collection := repo.db.Collection("users")
    _, err := collection.DeleteOne(context.TODO(), bson.D{{"_id", id}})
    return err
}

func (repo UserRepository) updateUser(id int, name string) error {
    collection := repo.db.Collection("users")
    _, err := collection.UpdateOne(context.TODO(), bson.D{{"_id", id}}, bson.D{{"$set", bson.M{"name": name}}})
    return err
}
```

1: Test Create

```go
func TestUserRepository_addUserToDb(t *testing.T) {
    type fields struct {
        db *mongo.Database
    }
    type args struct {
        user User
    }
    tests := []struct {
```

```go
		name    string
		fields  fields
		args    args
		wantErr bool
		mock    func(db *mtest.T)
	}{
		{
			name: "1: error creating user",
			args: args{
				user: User{
					Id:   1,
					Name: "shubham",
				},
			},
			wantErr: true,
			mock: func(db *mtest.T) {
				// zero means errored
				db.AddMockResponses(bson.D{{"ok", 0}})
			},
		},
		{
			name: "2: success",
			args: args{
				user: User{
					Id:   1,
					Name: "shubham",
				},
			},
			wantErr: false,
			mock: func(db *mtest.T) {
				db.AddMockResponses(mtest.CreateSuccessResponse())
			},
		},
	}
	for _, tt := range tests {
		t.Run(tt.name, func(t *testing.T) {
			mt := mtest.New(t, mtest.NewOptions().ClientType(mtest.Mock))
			mt.Run(tt.name, func(m *mtest.T) {
				repo := UserRepository{
					db: m.DB,
				}
				tt.mock(m)
				if err := repo.addUserToDb(tt.args.user); (err != nil) != tt.wantErr {
					t.Errorf("addUserToDb() error = %v, wantErr %v", err, tt.wantErr)
				}
			})
		})
	}
}
```

2: Test Get

```go
func TestUserRepository_getUserToDb(t *testing.T) {
	type fields struct {
		db *mongo.Database
	}
	type args struct {
		id int
	}
```

```go
    tests := []struct {
        name    string
        fields  fields
        args    args
        wantErr bool
        mock    func(db *mtest.T)
    }{
        {
            name: "1: success",
            args: args{
                id: 1,
            },
            wantErr: false,
            mock: func(db *mtest.T) {
                db.AddMockResponses(mtest.CreateCursorResponse(1, "a.b", mtest.FirstBatch, bson.D{{"_id", 1}}))
            },
        },
    }
    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            mt := mtest.New(t, mtest.NewOptions().ClientType(mtest.Mock))
            mt.Run(tt.name, func(m *mtest.T) {
                repo := UserRepository{
                    db: m.DB,
                }
                tt.mock(m)
                if err := repo.getUser(tt.args.id); (err != nil) != tt.wantErr {
                    t.Errorf("addUserToDb() error = %v, wantErr %v", err, tt.wantErr)
                }
            })
        })
    }
}
```

3: Test Update

```go
func TestUserRepository_updateUserToDb(t *testing.T) {
    type fields struct {
        db *mongo.Database
    }
    type args struct {
        id   int
        name string
    }
    tests := []struct {
        name    string
        fields  fields
        args    args
        wantErr bool
        mock    func(db *mtest.T)
    }{
        {
            name: "1: success",
            args: args{
                id: 1,
            },
            wantErr: false,
            mock: func(db *mtest.T) {
                db.AddMockResponses(mtest.CreateSuccessResponse())
```

```go
			},
		},
	}
	for _, tt := range tests {
		t.Run(tt.name, func(t *testing.T) {
			mt := mtest.New(t, mtest.NewOptions().ClientType(mtest.Mock))
			mt.Run(tt.name, func(m *mtest.T) {
				repo := UserRepository{
					db: m.DB,
				}
				tt.mock(m)
				if err := repo.updateUser(tt.args.id, tt.args.name); (err != nil) != tt.wantErr {
					t.Errorf("addUserToDb() error = %v, wantErr %v", err, tt.wantErr)
				}
			})
		})
	}
}
```

4: Test Delete

```go
func TestUserRepository_deleteUserToDb(t *testing.T) {
	type fields struct {
		db *mongo.Database
	}
	type args struct {
		id int
	}
	tests := []struct {
		name    string
		fields  fields
		args    args
		wantErr bool
		mock    func(db *mtest.T)
	}{
		{
			name: "1: success",
			args: args{
				id: 1,
			},
			wantErr: false,
			mock: func(db *mtest.T) {
				db.AddMockResponses(bson.D{{"ok", 1}, {"acknowledged", true}, {"n", 1}})
			},
		},
	}
	for _, tt := range tests {
		t.Run(tt.name, func(t *testing.T) {
			mt := mtest.New(t, mtest.NewOptions().ClientType(mtest.Mock))
			mt.Run(tt.name, func(m *mtest.T) {
				repo := UserRepository{
					db: m.DB,
				}
				tt.mock(m)
				if err := repo.deleteUser(tt.args.id); (err != nil) != tt.wantErr {
					t.Errorf("addUserToDb() error = %v, wantErr %v", err, tt.wantErr)
				}
			})
		})
```

```
    }
}
```

Testing Gin Server

```go
func healthCheckHandler(c *gin.Context) {
    c.JSON(http.StatusOK, gin.H{
        "message": "pong",
    })
}

```

```go
func Test_healthCheckHandler(t *testing.T) {
    type args struct {
       c *gin.Context
    }
    tests := []struct {
       name string
       args args
       want int
       body string
    }{
       {
          name: "1: positive",
          want: 200,
          body: `{"message":"pong"}`,
       },
    }
    for _, tt := range tests {
       t.Run(tt.name, func(t *testing.T) {
          w := httptest.NewRecorder()
          req, _ := http.NewRequest(http.MethodGet, "/heathCheck", nil)
          ctx, _ := gin.CreateTestContext(w)
          ctx.Request = req
          tt.args.c = ctx
          healthCheckHandler(tt.args.c)
          assert.Equal(t, tt.want, w.Code)
          assert.Equal(t, tt.body, w.Body.String())
       })
    }
}
```

Run test for a file

```
go test your_file_test.go
```

Run Single test for a package

```
go test -run <function_name> path/to/your/package
```

Run test for a package

```
go test path/to/your/package
```

Running all the tests

```
go test ./... -coverprofile=coverage.out
```

Percentage Of Covered Statements

```
go tool cover -func=coverage.out
```

Coverage viewer:

```
go tool cover -html=coverage.out -o cover.html
go tool cover -html=coverage.out
```