

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

```
1 func sum(a, b int) (int, error) {
2     return a + b, nil
3 }
4
5 func Test_sum(t *testing.T) {
6     expected := 10
7     wantErr := false
8     actual, err := sum(3, 7)
9     if (err != nil) != wantErr {
10         t.Errorf("error = %v, wantErr %v", err, wantErr)
11         return
12     }
13     if actual != expected {
14         t.Errorf("got %d, want %d", actual, expected)
15         return
16     }
17 }
```

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.

```
1 func Test_sum(t *testing.T) {
2     type args struct {
3         a int
4         b int
5     }
6     tests := []struct {
7         name    string
8         args     args
9         want    int
10        wantErr bool
11    }{
12        {
13            name: "1: positive",
14            args: args{
15                a: 1,
```

```

16         b: 2,
17     },
18     want: 3,
19 },
20 {
21     name: "2: negative",
22     args: args{
23         a: 1,
24         b: 1,
25     },
26     wantErr: true,
27 },
28 }
29 for _, tt := range tests {
30     t.Run(tt.name, func(t *testing.T) {
31         got, err := sum(tt.args.a, tt.args.b)
32         if (err != nil) != tt.wantErr {
33             t.Errorf("sum() error = %v, wantErr %v", err, tt.wantErr)
34             return
35         }
36         if got != tt.want {
37             t.Errorf("sum() got = %v, want %v", got, tt.want)
38         }
39     })
40 }
41 }

```

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 from other object oriented languages such as java / python you are familiar with mocking frameworks such as **mockito** / **unittest** which can easily mock the entire object and methods but **that's not the case with golang**.

Mocking simple functions

```

1 func addUserToDb(user User) error{
2     return nil
3 }
4
5 func addUser(name string) error {
6     user := User{name}
7     return addUserToDb(user)
8 }

```

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

```
1 var addUserToDb = func(user User) error {
2     return nil
3 }
4
5 func Test_addUser(t *testing.T) {
6     type args struct {
7         name string
8     }
9     tests := []struct {
10         name    string
11         args    args
12         wantErr bool
13         mock    func()
14     }{
15         {
16             name: "1: error while saving user to db",
17             args: args{
18                 name: "shubham",
19             },
20             wantErr: true,
21             mock: func() {
22                 addUserToDb = func(user User) error {
23                     return errors.New("error while saving user to db")
24                 }
25             },
26         },
27         {
28             name: "2: positive",
29             args: args{
30                 name: "shubham",
31             },
32             wantErr: false,
33             mock: func() {
34                 addUserToDb = func(user User) error {
35                     return nil
36                 }
37             },
38         },
39     }
40
41     originalAddUserToDb := addUserToDb
42     for _, tt := range tests {
43         t.Run(tt.name, func(t *testing.T) {
44             tt.mock()
45             if err := addUser(tt.args.name); (err != nil) != tt.wantErr {
46                 t.Errorf("addUser() error = %v, wantErr %v", err, tt.wantErr)
47             }
48         })
49         addUserToDb = originalAddUserToDb
50     }
51 }
52
```

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

```
1 func addUser(name string, addUserToDb func(user User) error) error {
2     user := User{name}
3     return addUserToDb(user)
4 }
5
6 func Test_addUser(t *testing.T) {
7     type args struct {
8         name      string
9         addUserToDb func(user User) error
10    }
11    tests := []struct {
12        name     string
13        args     args
14        wantErr bool
15    }{
16        {
17            name: "1: error while saving user to db",
18            args: args{
19                name: "shubham",
20                addUserToDb: func(user User) error {
21                    return errors.New("error while saving user to db")
22                },
23            },
24            wantErr: true,
25        },
26        {
27            name: "2: positive",
28            args: args{
29                name: "shubham",
30                addUserToDb: func(user User) error {
31                    return nil
32                },
33            },
34            wantErr: false,
35        },
36    }
37
38    for _, tt := range tests {
39        t.Run(tt.name, func(t *testing.T) {
40            if err := addUser(tt.args.name, tt.args.addUserToDb); (err != nil) != tt.wantErr {
41                t.Errorf("addUser() error = %v, wantErr %v", err, tt.wantErr)
42            }
43        })
44    }
45 }
46
```

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.

```
1
2 func addUserToDb(user User) error {
3     return nil
4 }
5
6 func addUser(name string) error {
7     user := User{name}
8     return addUserToDb(user)
9 }
10
11 func Test_addUser(t *testing.T) {
12     type args struct {
13         name string
14     }
15     tests := []struct {
16         name     string
17         args     args
18         wantErr  bool
19         mock     func()
20     }{
21         {
22             name: "1: error while saving user",
23             args: args{
24                 name: "shubham",
25             },
26             wantErr: true,
27             mock: func() {
28                 mock.Patch(addUserToDb, func(user User) error {
29                     return errors.New("error while saving to db")
30                 })
31             },
32         },
33         {
34             name: "2: positive",
35             args: args{
36                 name: "shubham",
37             },
38             wantErr: false,
39             mock: func() {
40                 mock.Patch(addUserToDb, func(user User) error {
41                     return nil
42                 })
43             },
44         },
45     }
46     for _, tt := range tests {
47         t.Run(tt.name, func(t *testing.T) {
48             tt.mock()
49             if err := addUser(tt.args.name); (err != nil) != tt.wantErr {
50                 t.Errorf("addUser() error = %v, wantErr %v", err, tt.wantErr)
51             }
52         })
53     }
```

```
54 }
55
```

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 doesn'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

```
1  type User struct {
2      name string
3  }
4
5  type UserRepository struct {}
6
7  func (UserRepository) addUserToDb(user User) error {
8      return nil
9  }
10
11 func addUser(name string) error {
12     user := User{name}
13     repo := UserRepository{}
14     return repo.addUserToDb(user)
15 }
16
17 func Test_addUser(t *testing.T) {
18     type args struct {
19         name string
20     }
21     tests := []struct {
22         name     string
23         args     args
24         wantErr  bool
25         mock     func()
26     }{
27         {
28             name: "1: error while adding user",
29             args: args{
30                 name: "test",
31             },
32             wantErr: true,
33             mock: func() {
34                 mock.Patch((*UserRepository).addUserToDb, func(_ *UserRepository, user User) error {
35                     return errors.New("errored")
36                 })
37             }
38         }
39     }
40     for _, test := range tests {
41         t.Run(test.name, func(t *testing.T) {
42             args := test.args
43             err := addUser(args.name)
44             if (err != nil) != test.wantErr {
45                 t.Errorf("addUser(%s) error = %v, wantErr %v", args.name, err, test.wantErr)
46             }
47         })
48     }
49 }
```

```

36         })
37     },
38 },
39 {
40     name: "2: positive",
41     args: args{
42         name: "test",
43     },
44     wantErr: false,
45     mock: func() {
46         mock.Patch((*UserRepository).addUserToDb, func(_ *UserRepository, user User) error {
47             return nil
48         })
49     },
50 },
51 }
52 for _, tt := range tests {
53     t.Run(tt.name, func(t *testing.T) {
54         tt.mock()
55         if err := addUser(tt.args.name); (err != nil) != tt.wantErr {
56             t.Errorf("addUser() error = %v, wantErr %v", err, tt.wantErr)
57         }
58     })
59 }
60 }
61

```

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.

```

1  type User struct {
2      name string
3  }
4
5  type UserRepository interface {
6      addUserToDb(user User) error
7  }
8
9  type UserService struct {
10     UserRepo UserRepository
11 }
12
13 type UserRepositoryImpl struct{}
14
15 func (UserRepositoryImpl) addUserToDb(user User) error {
16     return nil
17 }
18
19 func (s UserService) addUser(name string) error {
20     user := User{name}
21     return s.UserRepo.addUserToDb(user)
22 }
23
24
25 type MockUserRepoImpl struct {
26     mock.Mock
27 }

```

```

28
29 func (m *MockUserRepoIml) addUserToDb(user User) error {
30     args := m.Called(user)
31     return args.Error(0)
32 }
33
34 func TestUserService_addUser(t *testing.T) {
35     type fields struct {
36         UserRepo *MockUserRepoIml
37     }
38     type args struct {
39         name string
40     }
41     tests := []struct {
42         name    string
43         fields  fields
44         args    args
45         wantErr bool
46         mock    func(f fields)
47     }{
48         {
49             name: "1: error while adding user",
50             fields: fields{
51                 UserRepo: &MockUserRepoIml{},
52             },
53             args: args{
54                 name: "shubham",
55             },
56             wantErr: true,
57             mock: func(f fields) {
58                 f.UserRepo.On("addUserToDb", mock.AnythingOfType("User")).Return(errors.New("error while adding
59             },
60         },
61         {
62             name: "2: positive",
63             fields: fields{
64                 UserRepo: &MockUserRepoIml{},
65             },
66             args: args{
67                 name: "shubham",
68             },
69             wantErr: false,
70             mock: func(f fields) {
71                 f.UserRepo.On("addUserToDb", mock.AnythingOfType("User")).Return(nil)
72             },
73         },
74     }
75     for _, tt := range tests {
76         t.Run(tt.name, func(t *testing.T) {
77             tt.mock(tt.fields)
78             s := UserService{
79                 UserRepo: tt.fields.UserRepo,
80             }
81             if err := s.addUser(tt.args.name); (err != nil) != tt.wantErr {
82                 t.Errorf("addUser() error = %v, wantErr %v", err, tt.wantErr)
83             }
84
85             tt.fields.UserRepo.AssertExpectations(t)

```



```

86     })
87 }
88 }
89

```

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](#)
- Mongoddb: [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:

```

1  type User struct {
2      Id    int    `gorm:"column:id;primaryKey"`
3      Name string `gorm:"column:name"`
4  }
5
6  type UserService struct {
7      UserRepo UserRepository
8  }
9
10 type UserRepository struct {
11     db *gorm.DB
12 }
13
14 func (repo UserRepository) addUserToDb(user User) error {
15     result := repo.db.Create(&user)
16     return result.Error
17 }
18
19 func (repo UserRepository) getUser(id int) error {
20     var user User
21     result := repo.db.First(&user, id)
22     return result.Error
23 }
24
25 func (repo UserRepository) deleteUser(id int) error {
26     result := repo.db.Delete(&User{}, id)
27     return result.Error
28 }
29
30 func (repo UserRepository) updateUser(id int, name string) error {
31     return repo.db.Model(&User{}).Where("id = ?", id).Update("name", name).Error
32 }

```

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

```
1 func TestUserRepository_addUserToDb(t *testing.T) {
2     type fields struct {
3         db *gorm.DB
4     }
5     type args struct {
6         user User
7     }
8     tests := []struct {
9         name      string
10        fields   fields
11        args     args
12        wantErr  bool
13        mock     func(a args) (*gorm.DB, sqlmock.Sqlmock)
14    }{
15        {
16            name: "1: error while adding user",
17            args: args{
18                user: User{
19                    Id: 1,
20                    Name: "shubham",
21                },
22            },
23            wantErr: true,
24            mock: func(a args) (*gorm.DB, sqlmock.Sqlmock) {
25                db, mock, err := sqlmock.New()
26                if err != nil {
27                    log.Fatalf("An error '%s' was not expected when opening a stub database connection", err)
28                }
29
30                gormDB, err := gorm.Open(mysql.New(mysql.Config{
31                    Conn: db,
32                    SkipInitializeWithVersion: true,
33                })), &gorm.Config{})
34
35                mock.ExpectBegin()
36                mock.ExpectExec(regexp.QuoteMeta("INSERT INTO `users` (`name`,`id`) VALUES (?,?)")).
37                    WithArgs("shubham", 1).
38                    WillReturnError(errors.New("error"))
39                return gormDB, mock
40            },
41        },
42        {
43            name: "2: success",
44            args: args{
45                user: User{
46                    Id: 1,
47                    Name: "shubham",
48                },
49            },
50            wantErr: false,
51            mock: func(a args) (*gorm.DB, sqlmock.Sqlmock) {
52                db, mock, err := sqlmock.New()
53                if err != nil {
```

```

54         log.Fatalf("An error '%s' was not expected when opening a stub database connection", err)
55     }
56
57     gormDB, err := gorm.Open(mysql.New(mysql.Config{
58         Conn:          db,
59         SkipInitializeWithVersion: true,
60     })), &gorm.Config{})
61
62     mock.ExpectBegin()
63     mock.ExpectExec(regexp.QuoteMeta("INSERT INTO `users` (`name`,`id`) VALUES (?,?)")).
64         WithArgs("shubham", 1).
65         WillReturnResult(sqlmock.NewResult(1, 1))
66
67     mock.ExpectCommit()
68     return gormDB, mock
69 },
70 },
71 }
72 for _, tt := range tests {
73     t.Run(tt.name, func(t *testing.T) {
74         db, mock := tt.mock(tt.args)
75         repo := UserRepository{
76             db: db,
77         }
78         if err := repo.addUserToDb(tt.args.user); (err != nil) != tt.wantErr {
79             t.Errorf("addUserToDb() error = %v, wantErr %v", err, tt.wantErr)
80         }
81
82         if mock != nil {
83             if err := mock.ExpectationsWereMet(); err != nil {
84                 t.Errorf("there were unfulfilled expectations: %s", err)
85             }
86         }
87     })
88 }
89 }

```

2: Test Get

```

1 func TestUserRepository_getUser(t *testing.T) {
2     type fields struct {
3         db *gorm.DB
4     }
5     type args struct {
6         id int
7     }
8     tests := []struct {
9         name     string
10        fields  fields
11        args    args
12        wantErr bool
13        mock    func(a args) (*gorm.DB, sqlmock.Sqlmock)
14    }{
15        {
16            name: "1: not found",
17            args: args{
18                id: 1,
19            },
20            wantErr: true,

```

```

21     mock: func(a args) (*gorm.DB, sqlmock.Sqlmock) {
22         db, mock, err := sqlmock.New()
23         if err != nil {
24             log.Fatalf("An error '%s' was not expected when opening a stub database connection", err)
25         }
26
27         gormDB, err := gorm.Open(mysql.New(mysql.Config{
28             Conn:            db,
29             SkipInitializeWithVersion: true,
30         })), &gorm.Config{})
31
32         mock.ExpectQuery(regexp.QuoteMeta("SELECT * FROM `users` WHERE `users`.`id` = ? ORDER BY `users`
33             WithArgs(1, 1).
34             WillReturnError(errors.New("error"))
35         return gormDB, mock
36     },
37 },
38 {
39     name: "2: success",
40     args: args{
41         id: 1,
42     },
43     wantErr: false,
44     mock: func(a args) (*gorm.DB, sqlmock.Sqlmock) {
45         db, mock, err := sqlmock.New()
46         if err != nil {
47             log.Fatalf("An error '%s' was not expected when opening a stub database connection", err)
48         }
49
50         gormDB, err := gorm.Open(mysql.New(mysql.Config{
51             Conn:            db,
52             SkipInitializeWithVersion: true,
53         })), &gorm.Config{})
54
55         mockedRows := sqlmock.NewRows([]string{"id", "name"}).
56             AddRow("1", "shubham")
57
58         mock.ExpectQuery(regexp.QuoteMeta("SELECT * FROM `users` WHERE `users`.`id` = ? ORDER BY `users`
59             WithArgs(1, 1).
60             WillReturnRows(mockedRows)
61         return gormDB, mock
62     },
63 },
64 }
65 for _, tt := range tests {
66     t.Run(tt.name, func(t *testing.T) {
67         db, mock := tt.mock(tt.args)
68
69         repo := UserRepository{
70             db: db,
71         }
72         if err := repo.getUser(tt.args.id); (err != nil) != tt.wantErr {
73             t.Errorf("getUser() error = %v, wantErr %v", err, tt.wantErr)
74         }
75
76         if mock != nil {
77             if err := mock.ExpectationsWereMet(); err != nil {
78                 t.Errorf("there were unfulfilled expectations: %s", err)

```

```

79     }
80     }
81     })
82 }
83 }

```

3: Test Delete

```

1  func TestUserRepository_deleteUser(t *testing.T) {
2      type fields struct {
3          db *gorm.DB
4      }
5      type args struct {
6          id int
7      }
8      tests := []struct {
9          name      string
10         fields   fields
11         args     args
12         wantErr  bool
13         mock      func(a args) (*gorm.DB, sqlmock.Sqlmock)
14     }{
15         {
16             name: "1: error while deleting user",
17             args: args{
18                 id: 1,
19             },
20             wantErr: true,
21             mock: func(a args) (*gorm.DB, sqlmock.Sqlmock) {
22                 db, mock, err := sqlmock.New()
23                 if err != nil {
24                     log.Fatalf("An error '%s' was not expected when opening a stub database connection", err)
25                 }
26
27                 gormDB, err := gorm.Open(mysql.New(mysql.Config{
28                     Conn:      db,
29                     SkipInitializeWithVersion: true,
30                 })), &gorm.Config{})
31
32                 mock.ExpectBegin()
33                 mock.ExpectExec(regexp.QuoteMeta("DELETE FROM `users` WHERE `users`.`id` = ?")).
34                     WithArgs(1).
35                     WillReturnError(errors.New("error"))
36                 return gormDB, mock
37             },
38         },
39         {
40             name: "2: success",
41             args: args{
42                 id: 1,
43             },
44             wantErr: false,
45             mock: func(a args) (*gorm.DB, sqlmock.Sqlmock) {
46                 db, mock, err := sqlmock.New()
47                 if err != nil {
48                     log.Fatalf("An error '%s' was not expected when opening a stub database connection", err)
49                 }
50
51                 gormDB, err := gorm.Open(mysql.New(mysql.Config{

```

```

52         Conn:                db,
53         SkipInitializeWithVersion: true,
54     )), &gorm.Config{}}
55
56     mock.ExpectBegin()
57     mock.ExpectExec(regexp.QuoteMeta("DELETE FROM `users` WHERE `users`.`id` = ?")).
58         WithArgs(1).
59         WillReturnResult(sqlmock.NewResult(1, 1))
60     mock.ExpectCommit()
61     return gormDB, mock
62 },
63 },
64 }
65 for _, tt := range tests {
66     t.Run(tt.name, func(t *testing.T) {
67         db, mock := tt.mock(tt.args)
68
69         repo := UserRepository{
70             db: db,
71         }
72         if err := repo.deleteUser(tt.args.id); (err != nil) != tt.wantErr {
73             t.Errorf("deleteUser() error = %v, wantErr %v", err, tt.wantErr)
74         }
75
76         if mock != nil {
77             if err := mock.ExpectationsWereMet(); err != nil {
78                 t.Errorf("there were unfulfilled expectations: %s", err)
79             }
80         }
81     })
82 }
83 }

```

4: Test Update

```

1 func TestUserRepository_updateUser(t *testing.T) {
2     type fields struct {
3         db *gorm.DB
4     }
5     type args struct {
6         id int
7         name string
8     }
9     tests := []struct {
10         name string
11         fields fields
12         args args
13         wantErr bool
14         mock func(a args) (*gorm.DB, sqlmock.Sqlmock)
15     }{
16         {
17             name: "1: error while updating user",
18             args: args{
19                 name: "go",
20                 id: 1,
21             },
22             wantErr: true,
23             mock: func(a args) (*gorm.DB, sqlmock.Sqlmock) {
24                 db, mock, err := sqlmock.New()

```

```

25         if err != nil {
26             log.Fatalf("An error '%s' was not expected when opening a stub database connection", err)
27         }
28
29         gormDB, err := gorm.Open(mysql.New(mysql.Config{
30             Conn:            db,
31             SkipInitializeWithVersion: true,
32         })), &gorm.Config{})
33
34         mock.ExpectBegin()
35         mock.ExpectExec(regexp.QuoteMeta("UPDATE `users` SET `name`=? WHERE id = ?")).
36             WithArgs("go", 1).
37             WillReturnError(errors.New("error"))
38         return gormDB, mock
39     },
40 },
41 {
42     name: "2: success",
43     args: args{
44         name: "go",
45         id: 1,
46     },
47     wantErr: false,
48     mock: func(a args) (*gorm.DB, sqlmock.Sqlmock) {
49         db, mock, err := sqlmock.New()
50         if err != nil {
51             log.Fatalf("An error '%s' was not expected when opening a stub database connection", err)
52         }
53
54         gormDB, err := gorm.Open(mysql.New(mysql.Config{
55             Conn:            db,
56             SkipInitializeWithVersion: true,
57         })), &gorm.Config{})
58
59         mock.ExpectBegin()
60         mock.ExpectExec(regexp.QuoteMeta("UPDATE `users` SET `name`=? WHERE id = ?")).
61             WithArgs("go", 1).
62             WillReturnResult(sqlmock.NewResult(1, 1))
63         mock.ExpectCommit()
64         return gormDB, mock
65     },
66 },
67 }
68 for _, tt := range tests {
69     t.Run(tt.name, func(t *testing.T) {
70         db, mock := tt.mock(tt.args)
71
72         repo := UserRepository{
73             db: db,
74         }
75         if err := repo.updateUser(tt.args.id, tt.args.name); (err != nil) != tt.wantErr {
76             t.Errorf("updateUser() error = %v, wantErr %v", err, tt.wantErr)
77         }
78
79         if mock != nil {
80             if err := mock.ExpectationsWereMet(); err != nil {
81                 t.Errorf("there were unfulfilled expectations: %s", err)
82             }
83         }
84     })
85 }

```

```

83     }
84     })
85 }
86 }
87

```

Unit Tests for MongoDB

```

1
2 type User struct {
3     Id    int    `bson:"_id"`
4     Name string `bson:"name"`
5 }
6
7 type UserService struct {
8     UserRepo UserRepository
9 }
10
11 type UserRepository struct {
12     db *mongo.Database
13 }
14
15 func (repo UserRepository) addUserToDb(user User) error {
16     collection := repo.db.Collection("users")
17     _, err := collection.InsertOne(context.Background(), user)
18     return err
19 }
20
21 func (repo UserRepository) getUser(id int) error {
22     var user User
23     collection := repo.db.Collection("users")
24     return collection.FindOne(context.TODO(), bson.D{{"_id", id}}).Decode(&user)
25 }
26
27 func (repo UserRepository) deleteUser(id int) error {
28     collection := repo.db.Collection("users")
29     _, err := collection.DeleteOne(context.TODO(), bson.D{{"_id", id}})
30     return err
31 }
32
33 func (repo UserRepository) updateUser(id int, name string) error {
34     collection := repo.db.Collection("users")
35     _, err := collection.UpdateOne(context.TODO(), bson.D{{"_id", id}}, bson.D{{"$set", bson.M{"name": name}}})
36     return err
37 }

```

1: Test Create

```

1 func TestUserRepository_addUserToDb(t *testing.T) {
2     type fields struct {
3         db *mongo.Database
4     }
5     type args struct {
6         user User
7     }
8     tests := []struct {

```



```

9      name    string
10     fields  fields
11     args    args
12     wantErr bool
13     mock    func(db *mtest.T)
14 }{
15     {
16         name: "1: error creating user",
17         args: args{
18             user: User{
19                 Id: 1,
20                 Name: "shubham",
21             },
22         },
23         wantErr: true,
24         mock: func(db *mtest.T) {
25             // zero means errored
26             db.AddMockResponses(bson.D{{"ok", 0}})
27         },
28     },
29     {
30         name: "2: success",
31         args: args{
32             user: User{
33                 Id: 1,
34                 Name: "shubham",
35             },
36         },
37         wantErr: false,
38         mock: func(db *mtest.T) {
39             db.AddMockResponses(mtest.CreateSuccessResponse())
40         },
41     },
42 }
43 for _, tt := range tests {
44     t.Run(tt.name, func(t *testing.T) {
45         mt := mtest.New(t, mtest.NewOptions().ClientType(mtest.Mock))
46         mt.Run(tt.name, func(m *mtest.T) {
47             repo := UserRepository{
48                 db: m.DB,
49             }
50             tt.mock(m)
51             if err := repo.addUserToDb(tt.args.user); (err != nil) != tt.wantErr {
52                 t.Errorf("addUserToDb() error = %v, wantErr %v", err, tt.wantErr)
53             }
54         })
55     })
56 }
57 }

```

2: Test Get

```

1 func TestUserRepository_getUserToDb(t *testing.T) {
2     type fields struct {
3         db *mongo.Database
4     }
5     type args struct {
6         id int
7     }

```

```

8   tests := []struct {
9       name     string
10      fields   fields
11      args     args
12      wantErr  bool
13      mock     func(db *mtest.T)
14  }{
15      {
16          name: "1: success",
17          args: args{
18              id: 1,
19          },
20          wantErr: false,
21          mock: func(db *mtest.T) {
22              db.AddMockResponses(mtest.CreateCursorResponse(1, "a.b", mtest.FirstBatch, bson.D{{"_id", 1}}))
23          },
24      },
25  }
26  for _, tt := range tests {
27      t.Run(tt.name, func(t *testing.T) {
28          mt := mtest.New(t, mtest.NewOptions().ClientType(mtest.Mock))
29          mt.Run(tt.name, func(m *mtest.T) {
30              repo := UserRepository{
31                  db: m.DB,
32              }
33              tt.mock(m)
34              if err := repo.getUser(tt.args.id); (err != nil) != tt.wantErr {
35                  t.Errorf("addUserToDb() error = %v, wantErr %v", err, tt.wantErr)
36              }
37          })
38      })
39  }
40 }

```

3: Test Update

```

1  func TestUserRepository_updateUserToDb(t *testing.T) {
2      type fields struct {
3          db *mongo.Database
4      }
5      type args struct {
6          id  int
7          name string
8      }
9      tests := []struct {
10         name     string
11         fields   fields
12         args     args
13         wantErr  bool
14         mock     func(db *mtest.T)
15     }{
16         {
17             name: "1: success",
18             args: args{
19                 id: 1,
20             },
21             wantErr: false,
22             mock: func(db *mtest.T) {
23                 db.AddMockResponses(mtest.CreateSuccessResponse())

```

```

24     },
25     },
26 }
27 for _, tt := range tests {
28     t.Run(tt.name, func(t *testing.T) {
29         mt := mtest.New(t, mtest.NewOptions().ClientType(mtest.Mock))
30         mt.Run(tt.name, func(m *mtest.T) {
31             repo := UserRepository{
32                 db: m.DB,
33             }
34             tt.mock(m)
35             if err := repo.updateUser(tt.args.id, tt.args.name); (err != nil) != tt.wantErr {
36                 t.Errorf("addUserToDb() error = %v, wantErr %v", err, tt.wantErr)
37             }
38         })
39     })
40 }
41 }

```

4: Test Delete

```

1 func TestUserRepository_deleteUserToDb(t *testing.T) {
2     type fields struct {
3         db *mongo.Database
4     }
5     type args struct {
6         id int
7     }
8     tests := []struct {
9         name     string
10        fields   fields
11        args     args
12        wantErr  bool
13        mock     func(db *mtest.T)
14    }{
15        {
16            name: "1: success",
17            args: args{
18                id: 1,
19            },
20            wantErr: false,
21            mock: func(db *mtest.T) {
22                db.AddMockResponses(bson.D{{"ok", 1}, {"acknowledged", true}, {"n", 1}})
23            },
24        },
25    }
26    for _, tt := range tests {
27        t.Run(tt.name, func(t *testing.T) {
28            mt := mtest.New(t, mtest.NewOptions().ClientType(mtest.Mock))
29            mt.Run(tt.name, func(m *mtest.T) {
30                repo := UserRepository{
31                    db: m.DB,
32                }
33                tt.mock(m)
34                if err := repo.deleteUser(tt.args.id); (err != nil) != tt.wantErr {
35                    t.Errorf("addUserToDb() error = %v, wantErr %v", err, tt.wantErr)
36                }
37            })
38        })
39    }

```

```
39     }
40 }
```

Testing Gin Server

```
1 func healthCheckHandler(c *gin.Context) {
2     c.JSON(http.StatusOK, gin.H{
3         "message": "pong",
4     })
5 }
6
```

```
1 func Test_healthCheckHandler(t *testing.T) {
2     type args struct {
3         c *gin.Context
4     }
5     tests := []struct {
6         name string
7         args args
8         want int
9         body string
10    }{
11        {
12            name: "1: positive",
13            want: 200,
14            body: `{"message":"pong"}`,
15        },
16    }
17    for _, tt := range tests {
18        t.Run(tt.name, func(t *testing.T) {
19            w := httptest.NewRecorder()
20            req, _ := http.NewRequest(http.MethodGet, "/healthCheck", nil)
21            ctx, _ := gin.CreateTestContext(w)
22            ctx.Request = req
23            tt.args.c = ctx
24            healthCheckHandler(tt.args.c)
25            assert.Equal(t, tt.want, w.Code)
26            assert.Equal(t, tt.body, w.Body.String())
27        })
28    }
29 }
```

Run test for a file

```
1 go test your_file_test.go
```

Run Single test for a package

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

Run test for a package

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

Running all the tests

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

Percentage Of Covered Statements

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

Coverage viewer:

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

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