

How to write Backend unit tests

I. Target

II. Setup


1. Eliminate dependencies (optional)
2. Mock
3. Test suite
4. Library

III. Test cases

1. Table driven test
2. Verify mock function
3. Repository layer test cases

IV. Test Report

I. Target

- Help the engineer identify the bug in the early stages when making code changes
- Test coverage: we will gradually increase the coverage over time. Some code coverage metrics  [Whats my Coverage? \(C0 C1 C2 C3 + Path\)](#)

II. Setup

We will take an example to test the AdminNote functions in license-service

1. Eliminate dependencies (optional)

Code Example - GetAdminNote function

```
1 func AddAdminNotes(ctx context.Context, salesOrderId string, adminId int, adminUserName string, note string) (bool, error) {
2     if stringUtils.IsBlank(salesOrderId) {
3         return false, errors.New("sales order id cannot be blank")
4     }
5
6     if adminId <= 0 {
7         return false, errors.New("invalid admin user id")
8     }
9
10    if stringUtils.IsBlank(adminUserName) {
11        return false, errors.New("invalid admin user name")
12    }
13
14    if stringUtils.IsBlank(note) {
15        return false, errors.New("note cannot be blank")
16    }
17
18    return postgres.AddAdminNotes(ctx, postgres.GetDb(), salesOrderId, adminId, adminUserName, note)
19 }
20
21 func GetAdminNotes(ctx context.Context, salesOrderId string) ([]*models.GetAdminNotes, error) {
22     if stringUtils.IsBlank(salesOrderId) {
23         return nil, errors.New("sales order id cannot be blank")
24     }
25
26     return postgres.GetAdminNotes(ctx, postgres.GetDb(), salesOrderId)
27 }
```

1

Some engines are tightly coupled with external systems. We must abstract the interaction with the external system (database, message queue, HTTP call, ...) through the interface to mock these calls when writing unit tests. For example, this function uses the Postgres package directly, which makes it impossible to mock the AddAdminNotes call function.

We can refactor this by setting up an IPostgresRepository interface inside the Postgres package.

Code example - IPostgresRepository interface

```
1 package postgres
2
3 import (
4     "context"
5     "database/sql"
6
7     "azure.com/ezrx/license-engine/models"
8 )
9
10 type PostgresRepository struct {
11     db *sql.DB
12 }
13
```

```

14 func NewPostgresRepository(db *sql.DB) IPostgresRepository {
15     return &PostgresRepository{
16         db: db,
17     }
18 }
19
20 type IPostgresRepository interface {
21     GetAdminNotes(ctx context.Context, salesOrderId string) ([]*models.GetAdminNotes, error)
22     AddAdminNotes(ctx context.Context, salesOrderId string, adminId int, adminUserName string, note string) (bool, error)
23 }

```

Then, embed the repository interface as a property of the LogicUseCase struct. Each time we use the repository function, we will call through the IRepositoryInterface function.

Code Example - LogicUseCase struct

```

1 package logic
2
3 import (
4     "context"
5
6     "azure.com/ezrx/license-engine/models"
7     "azure.com/ezrx/license-engine/postgres"
8 )
9
10 type LogicUseCase struct {
11     repo postgres.IPostgresRepository
12 }
13
14 type ILogicUseCase interface {
15     AddAdminNotes(ctx context.Context, salesOrderId string, adminId int, adminUserName string, note string) (bool, error)
16     GetAdminNotes(ctx context.Context, salesOrderId string) ([]*models.GetAdminNotes, error)
17 }
18
19 func NewLogicUseCase(repo postgres.IPostgresRepository) ILogicUseCase {
20     return LogicUseCase{repo: repo}
21 }
22

```

Code Example - AdminNote functions (after refactor)

```

1 func (u LogicUseCase) AddAdminNotes(ctx context.Context, salesOrderId string, adminId int, adminUserName string, note string) (bool, error) {
2     if stringUtils.IsBlank(salesOrderId) {
3         return false, errors.New("sales order id cannot be blank")
4     }
5
6     if adminId <= 0 {
7         return false, errors.New("invalid admin user id")
8     }
9
10    if stringUtils.IsBlank(adminUserName) {
11        return false, errors.New("invalid admin user name")
12    }
13
14    if stringUtils.IsBlank(note) {
15        return false, errors.New("note cannot be blank")
16    }
17
18    return u.repo.AddAdminNotes(ctx, salesOrderId, adminId, adminUserName, note)
19 }
20
21 func (u LogicUseCase) GetAdminNotes(ctx context.Context, salesOrderId string) ([]*models.GetAdminNotes, error) {
22     if stringUtils.IsBlank(salesOrderId) {
23         return nil, errors.New("sales order id cannot be blank")
24     }
25
26     return u.repo.GetAdminNotes(ctx, salesOrderId)
27 }

```

2. Mock

- To generate mock for the IRepositoryInterface, we will use the [gomock](#) library to auto-generate the mock. We also can replace it with the newer [uber-mock](#) library to do this as well. We can utilize the **go:generate** comment to quickly generate mock without running mockgen for each package and interface individually. Adding this comment to the IRepositoryInterface file

```

1 //go:generate mockgen -source=postgres.go -destination=./mocks/postgres.go

```

```

1 go install github.com/golang/mock/mockgen@latest
2 mockgen --version
3 go generate -v ./...

```

- We also can [sqlmock](#) library to mock the database instance so that unit test will **NOT** require call to an actual database.

3. Test suite

- Test suites are the logical grouping or collection of test cases to run a single job with different test scenarios.

For instance, a test suite for AdminNoteSuite has multiple test cases, like:

- Test Case 1: Add admin note
 - Test Case 2: Get admin note
 - Test Case 3: Update admin note
- The package [suite](#) contains logic for creating testing suite structs and running the methods on those structs as tests. The most useful piece of this package is that you can create setup/teardown methods on your testing suites, which will run before/after the whole suite or individual tests.

Code example - Setup Test suite

```
1 type AdminNoteSuite struct {
2     suite.Suite
3     mockRepository *mock_postgres.MockIPostgresRepository
4     logicUseCase    ILogicUseCase
5 }
6
7 func TestAdminNoteSuite(t *testing.T) {
8     suite.Run(t, new(AdminNoteSuite))
9 }
10
11 func (suite *AdminNoteSuite) SetupSuite() {
12     ctrl := gomock.NewController(suite.T())
13     mockRepo := mock_postgres.NewMockIPostgresRepository(ctrl)
14     usecase := NewLogicUseCase(mockRepo)
15     suite.mockRepository = mockRepo
16     suite.logicUseCase = usecase
17 }
```

4. Library

- [GitHub - stretchr/testify: A toolkit with common assertions and mocks that plays nicely with the standard library](#)
- [GitHub - DATA-DOG/go-sqlmock: Sql mock driver for golang to test database interactions](#)
- [GitHub - golang/mock: GoMock is a mocking framework for the Go programming language.](#)
- [GitHub - uber-go/mock: GoMock is a mocking framework for the Go programming language.](#)

III. Test cases

1. Table driven test

In table-driven testing, a table (often implemented as a spreadsheet or a simple data structure) defines input data, expected outputs, and sometimes other parameters such as preconditions or postconditions. Each row in the table represents a unique test case, with columns representing different aspects of the test case, such as input values, expected results, and any other relevant information.

A typical test case will have a structure like this:

```
1 Name      string      `json:"name"`
2 InputArgs ArgsStruct  `json:"args"`
3 Want      WantStruct  `json:"want"`
4 WantErr   *string          `json:"wantErr"`
```

Field	Type	Value
Name	string	Test case name
InputArgs	struct	The input arguments struct. Will be defined for each function. Example: <div><pre>1 type Args struct { 2 SaleOrderId string `json:"saleOrderId"` 3 AdminId string `json:"adminId"` 4 AdminUserName string `json:"adminUserName"` 5 Note string `json:"note"` 6 }</pre></div>
Want	custom	The data we expected from the function with corresponding input args
WantErr	string	The error we expected from the function with corresponding input args

Then we will create a slice to store all the possible cases we want to test, then loop through the test cases slice to check the result

Code Example - AdminNote logic

```
1 func (suite *AdminNoteSuite) Test_AddAdminNote() {
2     type inputArgs struct {
3         salesOrderId string
4         adminId      int
5         adminUserName string
6         note         string
7     }
```

```

7   }
8
9   tests := []struct {
10      name      string
11      args       inputArgs
12      expectedErr error
13      expectedResult bool
14   }{
15      {
16         name: "Invalid Order ID",
17         args: inputArgs{
18             salesOrderId: "",
19             adminId:      1,
20             adminUserName: "zlpadmin",
21             note:         "Note test value",
22         },
23         expectedErr: errors.New("sales order id cannot be blank"),
24         expectedResult: false,
25     },
26     {
27         name: "Invalid Admin ID",
28         args: inputArgs{
29             salesOrderId: "2601",
30             adminId:      -1,
31             adminUserName: "zlpadmin",
32             note:         "Note test value",
33         },
34         expectedErr: errors.New("invalid admin user id"),
35         expectedResult: false,
36     },
37     {
38         name: "Invalid admin user name",
39         args: inputArgs{
40             salesOrderId: "2601",
41             adminId:      1,
42             adminUserName: "",
43             note:         "Note test value",
44         },
45         expectedErr: errors.New("invalid admin user name"),
46         expectedResult: false,
47     },
48     {
49         name: "Invalid note",
50         args: inputArgs{
51             salesOrderId: "2601",
52             adminId:      1,
53             adminUserName: "testzpadmin",
54             note:         "",
55         },
56         expectedErr: errors.New("note cannot be blank"),
57         expectedResult: false,
58     },
59     {
60         name: "Success",
61         args: inputArgs{
62             salesOrderId: "2601",
63             adminId:      1,
64             adminUserName: "testzpadmin",
65             note:         "Note test value",
66         },
67         expectedErr: nil,
68         expectedResult: true,
69     },
70 }
71
72 ctx := context.Background()
73 for _, c := range tests {
74     suite.Run(c.name, func() {
75         suite.mockRepository.EXPECT().
76             AddAdminNotes(gomock.Any(), c.args.salesOrderId, c.args.adminId, c.args.adminUserName, c.args.note).
77             Return(c.expectedResult, c.expectedErr).
78             AnyTimes()
79         result, err := suite.logicUseCase.AddAdminNotes(ctx, c.args.salesOrderId, c.args.adminId, c.args.adminUserName, c.args.note)
80         // Check for the expected error
81         suite.Equal(c.expectedErr, err)
82         suite.Equal(c.expectedResult, result)
83     })
84 }
85 }

```

2. Verify mock function

- Input values: We can validate if the mock function has been called with the correct values by using the EXPECT() function. We must specify the value and not use the gomock.Any() value, since it makes the test meaningless.
- Number of calls: We can check whether the mock function has been called and how many times it has been called.

3. Repository layer test cases

- Database mock: We can mock the call to the database using the [go-sqlmock](#) library.

Code Example - Setup database mock before suite start

```
1 func (suite *AdminNoteSuite) SetupSuite() {
2     db, mock, err := sqlmock.New(sqlmock.QueryMatcherOption(sqlmock.QueryMatcherEqual))
3     if err != nil {
4         suite.Errorf(err, "an error '%s' was not expected when opening a stub database connection")
5     }
6     suite.NoError(err)
7     suite.db = db
8     suite.repo = NewPostgresRepository(db)
9     suite.mock = mock
10    suite.schema = utility.GetStrapiSchemaPostgres()
11 }
```

- Test cases: some common test cases for the repository layer


Code example - GetAdminNotes function

```
1 func (suite *AdminNoteSuite) Test_GetAdminNotesRefactor() {
2     type inputArgs struct {
3         SaleOrderID string
4     }
5
6     testCases := []struct {
7         name      string
8         args      inputArgs
9         rows      *sqlmock.Rows
10        sqlError  error
11        expectedValue []*models.GetAdminNotes
12        expectedErr  error
13    }{
14        {
15            name: "Success",
16            args: inputArgs{
17                SaleOrderID: "2601",
18            },
19            rows: sqlmock.NewRows([]string{"id", "sales_order_id", "admin_id", "admin_user_name", "note", "creation_time"}).
20                AddRow(1, "2601", 1, "zlpadmin1", "User note 1", "").
21                AddRow(2, "2601", 2, "zlpadmin2", "User note 2", ""),
22            sqlError: nil,
23            expectedValue: []*models.GetAdminNotes{
24                {
25                    ID: 1,
26                    SalesOrderID: "2601",
27                    AdminID: 1,
28                    AdminUserName: "zlpadmin1",
29                    Note: "User note 1",
30                },
31                {
32                    ID: 2,
33                    SalesOrderID: "2601",
34                    AdminID: 2,
35                    AdminUserName: "zlpadmin2",
36                    Note: "User note 2",
37                },
38            },
39            expectedErr: nil,
40        },
41        {
42            name: "Scan to struct error",
43            args: inputArgs{
44                SaleOrderID: "2601",
45            },
46            sqlError: nil,
47            rows: sqlmock.NewRows([]string{"id", "sales_order_id", "admin_id", "admin_user_name", "note", "creation_time"}).
48                AddRow(3, "2601", "fake", "zlpadmin1", "User note 1", ""),
49            expectedErr: errors.New("Failed to scan data"),
50            expectedValue: nil,
51        },
52        {
53            name: "Empty Data",
54            args: inputArgs{
55                SaleOrderID: "2602",
```

```

57     },
58     sqlError:      nil,
59     rows:          sqlmock.NewRows([]string{"id", "sales_order_id", "admin_id", "admin_user_name", "note", "creation_time"}),
60     expectedErr:   nil,
61     expectedValue: nil,
62 },
63 {
64     name: "Handle database management",
65     args: inputArgs{
66         SaleOrderID: "2602",
67     },
68     sqlError:      sql.ErrTxDone,
69     rows:          sqlmock.NewRows([]string{"id", "sales_order_id", "admin_id", "admin_user_name", "note", "creation_time"}),
70     expectedErr:   errors.New("Failed to fetch data"),
71     expectedValue: nil,
72 },
73 }
74
75 ctx := context.Background()
76
77 timezone := utility.GetMarketQueryTimezoneAbbrev()
78 for _, c := range testCases {
79     suite.Run(c.name, func() {
80         suite.mock.ExpectQuery(
81             `select
82             id,
83             sales_order_id,
84             admin_id,
85             admin_user_name,
86             note,
87             to_char(creation_time at time zone $1, 'MM/DD/YYYY HH24:MI:SS') creation_time
88             from `+suite.schema+`.admin_order_notes
89             where sales_order_id = $2
90             order by creation_time desc`).
91             WithArgs(timezone, c.args.SaleOrderID).
92             WillReturnRows(c.rows).
93             WillReturnError(c.sqlError)
94
95             // Call the method being tested
96             result, err := suite.repo.GetAdminNotes(ctx, c.args.SaleOrderID)
97
98             // Check the result
99             suite.Equal(err, c.expectedErr)
100             suite.Equal(result, c.expectedValue)
101         })
102     }
103 }

```

Test case	Descriptions	Example
Query matching	Assert the SQL query string	The query has been changed and does not match the expected query.
SQL input values	Check if the SQL input value is corresponding with the function input	A mismatch in order of the input values when preparing a query statement.
Scan to struct	Check the compatibility between row data and the struct	If the variable type has been changed (from string to int), there will be an error when scanning a string to an integer variable.
Handling database error	Check whether the database has been handled correctly. We should only log the error and return a general error instead. We must not return the database error to the client, which can cause security issues.	Set the database error and assert if the expected response from the function. 
Function return values	Assert the value from the database and the data from the function.	Sometimes, we want to process data returns from the database to map with our data model. Then we can check if the mapping is correct or not.

IV. Test Report

- Test coverage: We can use this command to run the test and then generate a profile of the test results. We can also specify the format using the **-json** flag, then pipe output to the `tparse` command to easily see the summarizing.

- Test coverage HTML: When the cover profile is generated, we can use the **go tool cover** to transform the profile to an HTML file, which makes it easy to visualize the coverage. We can see which lines have been covered, and which lines are not. We can also attach a screenshot with the coverage in the PR to make sure the code changes have been covered by unit test.

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

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