Rosaniline Follow
Mar 31, 2019 · 3 min read · Listen

Save

Unit testing GORM with go-sqlmock in Go

I started miss the time with `MagicMock` in python when I started writing Go. However, it is not really that difficult to write tests in Go.

Let's talk about how to test the database interaction in Go with GORM.

Prerequisite

Let's take simple model `Person` for example.

Model

```
type Person struct {
   ID   uuid.UUID `gorm:"column:id;primary_key" json:"id"`
   Name string    `gorm:"column:name" json:"name"`
}
```

`Repository`

The repository serves as the wrapped data access layer for the given model with two functions `GET` and `CREATE` .

```
type Repository interface {
   Get(id uuid.UUID) (*model.Person, error)
   Create(id uuid.UUID, name string) error
}
```

```go
func (p *repo) Create(id uuid.UUID, name string) error {
   person := &model.Person{
      ID:   id,
      Name: name,
   }

   return p.DB.Create(person).Error
}

func (p *repo) Get(id uuid.UUID) (*model.Person, error) {
   person := new(model.Person)

   err := p.DB.Where("id = ?", id).Find(person).Error

   return person, err
}
```

Our goal is to test the functions implemented in the `Repository` to ensure that the what happened under the GORM aligns with our expectation.

Testing Setup

Before dive into how the tests will be implemented. There are few components we have to go through first.

- `suite` from `testify`
- `sql-mock` from `DATA-DOG`

Suite

We use `suite` of testify to ease testing setup. If you are not yet familiar with `suite` , checkout the quote from the testify below.

> *The `suite` package provides functionality that you might be used to from more common object oriented languages. With it, you can build a testing suite as a struct, build setup/teardown methods and testing methods on your struct, and run them with 'go test' as per normal.*

Below is how the `suite` is written.

```
type Suite struct {
   suite.Suite
   DB   *gorm.DB
   mock sqlmock.Sqlmock

   repository Repository
   person     *model.Person
}
```

sql-mock

This is probably the main theme today. Again, we had quoted from `DATA-DOG` for what sql-mock is.

> ***sqlmock** is a mock library implementing sql/driver. Which has one and only purpose — to simulate any sql driver behavior in tests, without needing a real database connection. It helps to maintain correct TDD workflow.*

Testing

Finally we are here for today topic. Let's talk about how the `tests` should be written to test our GORM operations step by step.

- Setup `suite`
- Setup a series of `Expects` of sql statements with `sql-mock`
- Invoke functions to be tested
- Assert the return of the functions are correct
- Check whether `Expectations` of `sql-mock` were met

Setup suite

We will have our mocked database and repository ready at this stage. It quite similar for the orinary setup process but with `sql-mock` as the sql driver.

```
func (s *Suite) SetupSuite() {
   var (
      db  *sql.DB
```

```
        err error
    )

    db, s.mock, err = sqlmock.New()
    require.NoError(s.T(), err)

    s.DB, err = gorm.Open("postgres", db)
    require.NoError(s.T(), err)

    s.DB.LogMode(true)

    s.repository = CreateRepository(s.DB)
}
```

Test SELECT statement

Remember we have a `GET` function in our `Repository` right? To retrieve row in `person` with given id. Let's check how to test it.

```
func (s *Suite) Test_repository_Get() {
    var (
        id   = uuid.NewV4()
        name = "test-name"
    )

    s.mock.ExpectQuery(regexp.QuoteMeta(
        `SELECT * FROM "person" WHERE (id = $1)`)).
        WithArgs(id.String()).
        WillReturnRows(sqlmock.NewRows([]string{"id", "name"}).
            AddRow(id.String(), name))

    res, err := s.repository.Get(id)

    require.NoError(s.T(), err)
    require.Nil(s.T(), deep.Equal(&model.Person{ID: id, Name: name},
res))
}
```

Here we leverage `sql-mock` to do these for us

- Expect `SELECT * FROM "person" WHERE (id = $1)` to be executed
- With arg `id`
- Return the `id` and `name` as the stub of expected person record

Test INSERT statement

Besides `GET` there is another `CREATE` function in the `Repository` .

```
func (s *Suite) Test_repository_Create() {
   var (
      id   = uuid.NewV4()
      name = "test-name"
   )

   s.mock.ExpectQuery(regexp.QuoteMeta(
      `INSERT INTO "person" ("id","name")
       VALUES ($1,$2) RETURNING "person"."id"`)).
      WithArgs(id, name).
      WillReturnRows(
         sqlmock.NewRows([]string{"id"}).AddRow(id.String()))

   err := s.repository.Create(id, name)
```

Here we leverage `sql` `sql-mock` to do these for us

- Expect `INSERT` statement to be exectued
- With arg `id` and `name`
- Return the `id` for created row

Check whether `Expectations` of `sql-mock` were met

The check is put in the `AfterTest` section to ensure it is performed after each test case.

```
func (s *Suite) AfterTest(_, _ string) {
   require.NoError(s.T(), s.mock.ExpectationsWereMet())
}
```

Here is repository for abovementioned code if you find it hard to read with separated peices.

Testing with GORM in Go is not that difficult, right? happy coding 🐤

Golang Unittest Testing Gorm Go

👏 373 | 💬 4 | •••