

---

go report A+

go report A+

go report A+

codecov 94%

## Sql driver mock for Golang

**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.

- this library is now complete and stable. (you may not find new changes for this reason)
- supports concurrency and multiple connections.
- supports **go1.8** Context related feature mocking and Named sql parameters.
- does not require any modifications to your source code.
- the driver allows to mock any sql driver method behavior.
- has strict by default expectation order matching.
- has no third party dependencies.

**NOTE:** in **v1.2.0** **sqlmock.Rows** has changed to struct from interface, if you were using any type references to that interface, you will need to switch it to a pointer struct type. Also, **sqlmock.Rows** were used to implement **driver.Rows** interface, which was not required or useful for mocking and was removed. Hope it will not cause issues.

### Looking for maintainers

I do not have much spare time for this library and willing to transfer the repository ownership to person or an organization motivated to maintain it. Open up a conversation if you are interested. See #230.

### Install

```
1 go get github.com/DATA-DOG/go-sqlmock
```

### Documentation and Examples

Visit godoc for general examples and public api reference. See **.travis.yml** for supported **go** versions. Different use case, is to functionally test with a real database - go-txdb all database related actions are isolated within a single transaction so the database can remain in the same state.

See implementation examples:

- 
- blog API server
  - the same orders example

### Something you may want to test, assuming you use the go-mysql-driver

```
1 package main
2
3 import (
4     "database/sql"
5
6     _ "github.com/go-sql-driver/mysql"
7 )
8
9 func recordStats(db *sql.DB, userID, productID int64) (err error) {
10     tx, err := db.Begin()
11     if err != nil {
12         return
13     }
14
15     defer func() {
16         switch err {
17             case nil:
18                 err = tx.Commit()
19             default:
20                 tx.Rollback()
21         }
22     }()
23
24     if _, err = tx.Exec("UPDATE products SET views = views + 1"); err != nil {
25         return
26     }
27     if _, err = tx.Exec("INSERT INTO product_viewers (user_id, product_id) VALUES (?, ?)", userID, productID); err != nil {
28         return
29     }
30     return
31 }
32
33 func main() {
34     // @NOTE: the real connection is not required for tests
35     db, err := sql.Open("mysql", "root@/blog")
36     if err != nil {
37         panic(err)
38     }
39     defer db.Close()
40
41     if err = recordStats(db, 1 /*some user id*/, 5 /*some product id*/)
42         ; err != nil {
```

---

```
42         panic(err)
43     }
44 }
```

## Tests with sqlmock

```
1 package main
2
3 import (
4     "fmt"
5     "testing"
6
7     "github.com/DATA-DOG/go-sqlmock"
8 )
9
10 // a successful case
11 func TestShouldUpdateStats(t *testing.T) {
12     db, mock, err := sqlmock.New()
13     if err != nil {
14         t.Fatalf("an error '%s' was not expected when opening a stub
15             database connection", err)
16     }
17     defer db.Close()
18
19     mock.ExpectBegin()
20     mock.ExpectExec("UPDATE products").WillReturnResult(sqlmock.
21         NewResult(1, 1))
22     mock.ExpectExec("INSERT INTO product_viewers").WithArgs(2, 3).
23         WillReturnResult(sqlmock.NewResult(1, 1))
24     mock.ExpectCommit()
25
26     // now we execute our method
27     if err = recordStats(db, 2, 3); err != nil {
28         t.Errorf("error was not expected while updating stats: %s", err)
29     }
30
31     // we make sure that all expectations were met
32     if err := mock.ExpectationsWereMet(); err != nil {
33         t.Errorf("there were unfulfilled expectations: %s", err)
34     }
35 }
36
37 // a failing test case
38 func TestShouldRollbackStatUpdatesOnFailure(t *testing.T) {
39     db, mock, err := sqlmock.New()
40     if err != nil {
41         t.Fatalf("an error '%s' was not expected when opening a stub
42             database connection", err)
43     }
44     defer db.Close()
45
46     mock.ExpectBegin()
47     mock.ExpectExec("UPDATE products").WillReturnResult(sqlmock.
48         NewResult(1, 1))
49     mock.ExpectExec("INSERT INTO product_viewers").WithArgs(2, 3).
50         WillReturnResult(sqlmock.NewResult(1, 1))
51     mock.ExpectCommit()
52
53     // now we execute our method
54     if err = recordStats(db, 2, 3); err != nil {
55         t.Errorf("error was not expected while updating stats: %s", err)
56     }
57
58     // we make sure that all expectations were met
59     if err := mock.ExpectationsWereMet(); err != nil {
60         t.Errorf("there were unfulfilled expectations: %s", err)
61     }
62 }
```

---

```

39     }
40     defer db.Close()
41
42     mock.ExpectBegin()
43     mock.ExpectExec("UPDATE products").WillReturnResult(sqlmock.
        NewResult(1, 1))
44     mock.ExpectExec("INSERT INTO product_viewers").
45         WithArgs(2, 3).
46         WillReturnError(fmt.Errorf("some error"))
47     mock.ExpectRollback()
48
49     // now we execute our method
50     if err = recordStats(db, 2, 3); err == nil {
51         t.Errorf("was expecting an error, but there was none")
52     }
53
54     // we make sure that all expectations were met
55     if err := mock.ExpectationsWereMet(); err != nil {
56         t.Errorf("there were unfulfilled expectations: %s", err)
57     }
58 }

```

## Customize SQL query matching

There were plenty of requests from users regarding SQL query string validation or different matching option. We have now implemented the [QueryMatcher](#) interface, which can be passed through an option when calling `sqlmock.New` or `sqlmock.NewWithDSN`.

This now allows to include some library, which would allow for example to parse and validate `mysql` SQL AST. And create a custom `QueryMatcher` in order to validate SQL in sophisticated ways.

By default, **sqlmock** is preserving backward compatibility and default query matcher is `sqlmock.QueryMatcherRegexp` which uses expected SQL string as a regular expression to match incoming query string. There is an equality matcher: `QueryMatcherEqual` which will do a full case sensitive match.

In order to customize the `QueryMatcher`, use the following:

```

1     db, mock, err := sqlmock.New(sqlmock.QueryMatcherOption(sqlmock.
        QueryMatcherEqual))

```

The query matcher can be fully customized based on user needs. **sqlmock** will not provide a standard sql parsing matchers, since various drivers may not follow the same SQL standard.

---

## Matching arguments like `time.Time`

There may be arguments which are of `struct` type and cannot be compared easily by value like `time.Time`. In this case **sqlmock** provides an `Argument` interface which can be used in more sophisticated matching. Here is a simple example of time argument matching:

```
1 type AnyTime struct{}
2
3 // Match satisfies sqlmock.Argument interface
4 func (a AnyTime) Match(v driver.Value) bool {
5     _, ok := v.(time.Time)
6     return ok
7 }
8
9 func TestAnyTimeArgument(t *testing.T) {
10     t.Parallel()
11     db, mock, err := sqlmock.New()
12     if err != nil {
13         t.Errorf("an error '%s' was not expected when opening a stub
14             database connection", err)
15     }
16     defer db.Close()
17
18     mock.ExpectExec("INSERT INTO users").
19         WithArgs("john", AnyTime{}).
20         WillReturnResult(sqlmock.NewResult(1, 1))
21
22     _, err = db.Exec("INSERT INTO users(name, created_at) VALUES (?, ?)
23         ", "john", time.Now())
24     if err != nil {
25         t.Errorf("error '%s' was not expected, while inserting a row",
26             err)
27     }
28
29     if err := mock.ExpectationsWereMet(); err != nil {
30         t.Errorf("there were unfulfilled expectations: %s", err)
31     }
32 }
```

It only asserts that argument is of `time.Time` type.

## Run tests

```
1 go test -race
```

---

## Change Log

- **2019-04-06** - added functionality to mock a sql MetaData request
- **2019-02-13** - added `go.mod` removed the references and suggestions using `gopkg.in`.
- **2018-12-11** - added expectation of Rows to be closed, while mocking expected query.
- **2018-12-11** - introduced an option to provide **QueryMatcher** in order to customize SQL query matching.
- **2017-09-01** - it is now possible to expect that prepared statement will be closed, using **ExpectedPrepare.WillBeClosed**.
- **2017-02-09** - implemented support for **go1.8** features. **Rows** interface was changed to struct but contains all methods as before and should maintain backwards compatibility. **ExpectedQuery.WillReturnRows** may now accept multiple row sets.
- **2016-11-02** - `db.Prepare()` was not validating expected prepare SQL query. It should still be validated even if Exec or Query is not executed on that prepared statement.
- **2016-02-23** - added **sqlmock.AnyArg()** function to provide any kind of argument matcher.
- **2016-02-23** - convert expected arguments to driver.Value as natural driver does, the change may affect time.Time comparison and will be stricter. See issue.
- **2015-08-27** - **v1** api change, concurrency support, all known issues fixed.
- **2014-08-16** instead of **panic** during reflect type mismatch when comparing query arguments - now return error
- **2014-08-14** added **sqlmock.NewErrorResult** which gives an option to return driver.Result with errors for interface methods, see issue
- **2014-05-29** allow to match arguments in more sophisticated ways, by providing an **sqlmock.Argument** interface
- **2014-04-21** introduce **sqlmock.New()** to open a mock database connection for tests. This method calls sql.DB.Ping to ensure that connection is open, see issue. This way on Close it will surely assert if all expectations are met, even if database was not triggered at all. The old way is still available, but it is advisable to call db.Ping manually before asserting with db.Close.
- **2014-02-14** RowsFromCSVString is now a part of Rows interface named as FromCSVString. It has changed to allow more ways to construct rows and to easily extend this API in future. See issue 1 **RowsFromCSVString** is deprecated and will be removed in future

## Contributions

Feel free to open a pull request. Note, if you wish to contribute an extension to public (exported methods or types) - please open an issue before, to discuss whether these changes can be accepted. All backward incompatible changes are and will be treated cautiously

---

## **License**

The three clause BSD license