
ozzo-dbx

go report A+ go report A+ coverage 89% go report A+

Summary

- Description
- Requirements
- Installation
- Supported Databases
- Getting Started
- Connecting to Database
- Executing Queries
- Binding Parameters
- Building Queries
 - Building SELECT Queries
 - Building Query Conditions
 - Building Data Manipulation Queries
 - Building Schema Manipulation Queries
- CRUD Operations
 - Create
 - Read
 - Update
 - Delete
 - Null Handling
- Quoting Table and Column Names
- Using Transactions
- Logging Executed SQL Statements
- Supporting New Databases

Other Languages

Русский

Description

ozzo-dbx is a Go package that enhances the standard `database/sql` package by providing powerful data retrieval methods as well as DB-agnostic query building capabilities. ozzo-dbx is not an ORM. It has the following features:

- Populating data into structs and `NullString` maps
- Named parameter binding
- DB-agnostic query building methods, including `SELECT` queries, data manipulation queries, and schema manipulation queries
- Inserting, updating, and deleting model structs
- Powerful query condition building
- Open architecture allowing addition of new database support or customization of existing support
- Logging executed SQL statements
- Supporting major relational databases

For an example on how this library is used in an application, please refer to `go-rest-api` which is a starter kit for building RESTful APIs in Go.

Requirements

Go 1.13 or above.

Installation

Run the following command to install the package:

```
1 go get github.com/go-ozzo/ozzo-dbx
```

In addition, install the specific DB driver package for the kind of database to be used. Please refer to SQL database drivers for a complete list. For example, if you are using MySQL, you may install the following package:

```
1 go get github.com/go-sql-driver/mysql
```

and import it in your main code like the following:

```
1 import _ "github.com/go-sql-driver/mysql"
```

Supported Databases

The following databases are fully supported out of box:

- SQLite
- MySQL
- PostgreSQL
- MS SQL Server (2012 or above)
- Oracle

For other databases, the query building feature may not work as expected. You can create a custom builder to solve the problem. Please see the last section for more details.

Getting Started

The following code snippet shows how you can use this package in order to access data from a MySQL database.

```
1 import (
2     "fmt"
3     "github.com/go-ozzo/ozzo-dbx"
4     _ "github.com/go-sql-driver/mysql"
5 )
6
7 func main() {
8     db, _ := dbx.Open("mysql", "user:pass@/example")
9
10    // create a new query
11    q := db.NewQuery("SELECT id, name FROM users LIMIT 10")
12
13    // fetch all rows into a struct array
14    var users []struct {
15        ID, Name string
16    }
17    err := q.All(&users)
18
19    // fetch a single row into a struct
20    var user struct {
21        ID, Name string
22    }
23    err = q.One(&user)
24
25    // fetch a single row into a string map
26    data := dbx.NullStringMap{}
27    err = q.One(data)
28
29    // fetch row by row
```

```

30     rows2, _ := q.Rows()
31     for rows2.Next() {
32         _ = rows2.ScanStruct(&user)
33         // rows.ScanMap(data)
34         // rows.Scan(&id, &name)
35     }
36 }

```

And the following example shows how to use the query building capability of this package.

```

1  import (
2      "fmt"
3      "github.com/go-ozzo/ozzo-dbx"
4      _ "github.com/go-sql-driver/mysql"
5  )
6
7  func main() {
8      db, _ := dbx.Open("mysql", "user:pass@/example")
9
10     // build a SELECT query
11     // SELECT `id`, `name` FROM `users` WHERE `name` LIKE '%Charles%'
12     // ORDER BY `id`
13     q := db.Select("id", "name").
14         From("users").
15         Where(dbx.Like("name", "Charles")).
16         OrderBy("id")
17
18     // fetch all rows into a struct array
19     var users []struct {
20         ID, Name string
21     }
22     err := q.All(&users)
23
24     // build an INSERT query
25     // INSERT INTO `users` (`name`) VALUES ('James')
26     err = db.Insert("users", dbx.Params{
27         "name": "James",
28     }).Execute()
29 }

```

Connecting to Database

To connect to a database, call `dbx.Open()` in the same way as you would do with the `Open()` method in `database/sql`.

```

1  db, err := dbx.Open("mysql", "user:pass@hostname/db_name")

```

The method returns a `dbx.DB` instance which can be used to create and execute DB queries. Note that

the method does not really establish a connection until a query is made using the returned `dbx.DB` instance. It also does not check the correctness of the data source name either. Call `dbx.MustOpen()` to make sure the data source name is correct.

Executing Queries

To execute a SQL statement, first create a `dbx.Query` instance by calling `DB.NewQuery()` with the SQL statement to be executed. And then call `Query.Execute()` to execute the query if the query is not meant to retrieving data. For example,

```
1 q := db.NewQuery("UPDATE users SET status=1 WHERE id=100")
2 result, err := q.Execute()
```

If the SQL statement does retrieve data (e.g. a `SELECT` statement), one of the following methods should be called, which will execute the query and populate the result into the specified variable(s).

- `Query.All()`: populate all rows of the result into a slice of structs or `NullString` maps.
- `Query.One()`: populate the first row of the result into a struct or a `NullString` map.
- `Query.Column()`: populate the first column of the result into a slice.
- `Query.Row()`: populate the first row of the result into a list of variables, one for each returning column.
- `Query.Rows()`: returns a `dbx.Rows` instance to allow retrieving data row by row.

For example,

```
1 type User struct {
2     ID    int
3     Name  string
4 }
5
6 var (
7     users []User
8     user  User
9
10    row dbx.NullStringMap
11
12    id    int
13    name  string
14
15    err error
16 )
17
18 q := db.NewQuery("SELECT id, name FROM users LIMIT 10")
19
20 // populate all rows into a User slice
21 err = q.All(&users)
```

```

22 fmt.Println(users[0].ID, users[0].Name)
23
24 // populate the first row into a User struct
25 err = q.One(&user)
26 fmt.Println(user.ID, user.Name)
27
28 // populate the first row into a NullString map
29 err = q.One(&row)
30 fmt.Println(row["id"], row["name"])
31
32 var ids []int
33 err = q.Column(&ids)
34 fmt.Println(ids)
35
36 // populate the first row into id and name
37 err = q.Row(&id, &name)
38
39 // populate data row by row
40 rows, _ := q.Rows()
41 for rows.Next() {
42     _ = rows.ScanMap(&row)
43 }

```

When populating a struct, the following rules are used to determine which columns should go into which struct fields:

- Only exported struct fields can be populated.
- A field receives data if its name is mapped to a column according to the field mapping function `Query.FieldMapper`. The default field mapping function separates words in a field name by underscores and turns them into lower case. For example, a field name `FirstName` will be mapped to the column name `first_name`, and `MyID` to `my_id`.
- If a field has a `db` tag, the tag value will be used as the corresponding column name. If the `db` tag is a dash `-`, it means the field should NOT be populated.
- For anonymous fields that are of struct type, they will be expanded and their component fields will be populated according to the rules described above.
- For named fields that are of struct type, they will also be expanded. But their component fields will be prefixed with the struct names when being populated.

An exception to the above struct expansion is that when a struct type implements `sql.Scanner` or when it is `time.Time`. In this case, the field will be populated as a whole by the DB driver. Also, if a field is a pointer to some type, the field will be allocated memory and populated with the query result if it is not null.

The following example shows how fields are populated according to the rules above:

```

1 type User struct {

```

```

2     id      int
3     Type    int `db:"-`
4     MyName  string `db:"name"`
5     Profile
6     Address Address `db:"addr"`
7 }
8
9 type Profile struct {
10     Age int
11 }
12
13 type Address struct {
14     City string
15 }

```

- `User.id`: not populated because the field is not exported;
- `User.Type`: not populated because the `db` tag is `-`;
- `User.MyName`: to be populated from the `name` column, according to the `db` tag;
- `Profile.Age`: to be populated from the `age` column, since `Profile` is an anonymous field;
- `Address.City`: to be populated from the `addr.city` column, since `Address` is a named field of struct type and its fields will be prefixed with `addr.` according to the `db` tag.

Note that if a column in the result does not have a corresponding struct field, it will be ignored. Similarly, if a struct field does not have a corresponding column in the result, it will not be populated.

Binding Parameters

A SQL statement is usually parameterized with dynamic values. For example, you may want to select the user record according to the user ID received from the client. Parameter binding should be used in this case, and it is almost always preferred to prevent from SQL injection attacks. Unlike `database/sql` which does anonymous parameter binding, `ozzo-dbx` uses named parameter binding. *Anonymous parameter binding is not supported*, as it will mess up with named parameters. For example,

```

1 q := db.NewQuery("SELECT id, name FROM users WHERE id={:id}")
2 q.Bind(dbx.Params{"id": 100})
3 err := q.One(&user)

```

The above example will select the user record whose `id` is 100. The method `Query.Bind()` binds a set of named parameters to a SQL statement which contains parameter placeholders in the format of `{:ParamName}`.

If a SQL statement needs to be executed multiple times with different parameter values, it may be prepared to improve the performance. For example,

```
1 q := db.NewQuery("SELECT id, name FROM users WHERE id=:id")
2 q.Prepare()
3 defer q.Close()
4
5 q.Bind(dbx.Params{"id": 100})
6 err := q.One(&user)
7
8 q.Bind(dbx.Params{"id": 200})
9 err = q.One(&user)
10
11 // ...
```

Cancelable Queries

Queries are cancelable when they are used with `context.Context`. In particular, by calling `Query.WithContext()` you can associate a context with a query and use the context to cancel the query while it is running. For example,

```
1 q := db.NewQuery("SELECT id, name FROM users")
2 err := q.WithContext(ctx).All(&users)
```

Building Queries

Instead of writing plain SQLs, `ozzo-dbx` allows you to build SQLs programmatically, which often leads to cleaner, more secure, and DB-agnostic code. You can build three types of queries: the SELECT queries, the data manipulation queries, and the schema manipulation queries.

Building SELECT Queries

Building a SELECT query starts by calling `DB.Select()`. You can build different clauses of a SELECT query using the corresponding query building methods. For example,

```
1 db, _ := dbx.Open("mysql", "user:pass@example")
2 err := db.Select("id", "name").
3     From("users").
4     Where(dbx.HashExp{"id": 100}).
5     One(&user)
```

The above code will generate and execute the following SQL statement:

```
1 SELECT `id`, `name` FROM `users` WHERE `id`=:p0}
```

Notice how the table and column names are properly quoted according to the currently using database type. And parameter binding is used to populate the value of `p0` in the `WHERE` clause.

Every SQL keyword has a corresponding query building method. For example, `SELECT` corresponds to `Select()`, `FROM` corresponds to `From()`, `WHERE` corresponds to `Where()`, and so on. You can chain these method calls together, just like you would do when writing a plain SQL. Each of these methods returns the query instance (of type `dbx.SelectQuery`) that is being built. Once you finish building a query, you may call methods such as `One()`, `All()` to execute the query and populate data into variables. You may also explicitly call `Build()` to build the query and turn it into a `dbx.Query` instance which may allow you to get the SQL statement and do other interesting work.

Building Query Conditions

`ozzo-dbx` supports very flexible and powerful query condition building which can be used to build SQL clauses such as `WHERE`, `HAVING`, etc. For example,

```
1 // id=100
2 dbx.NewExp("id={:id}", dbx.Params{"id": 100})
3
4 // id=100 AND status=1
5 dbx.HashExp{"id": 100, "status": 1}
6
7 // status=1 OR age>30
8 dbx.Or(dbx.HashExp{"status": 1}, dbx.NewExp("age>30"))
9
10 // name LIKE '%admin%' AND name LIKE '%example%'
11 dbx.Like("name", "admin", "example")
```

When building a query condition expression, its parameter values will be populated using parameter binding, which prevents SQL injection from happening. Also if an expression involves column names, they will be properly quoted. The following condition building functions are available:

- `dbx.NewExp()`: creating a condition using the given expression string and binding parameters. For example, `dbx.NewExp("id={:id}", dbx.Params{"id":100})` would create the expression `id=100`.
- `dbx.HashExp`: a map type that represents name-value pairs concatenated by `AND` operators. For example, `dbx.HashExp{"id":100, "status":1}` would create `id=100 AND status=1`.
- `dbx.Not()`: creating a `NOT` expression by prepending `NOT` to the given expression.
- `dbx.And()`: creating an `AND` expression by concatenating the given expressions with the `AND` operators.
- `dbx.Or()`: creating an `OR` expression by concatenating the given expressions with the `OR` operators.

-
- `dbx.In()`: creating an `IN` expression for the specified column and the range of values. For example, `dbx.In("age", 30, 40, 50)` would create the expression `age IN (30, 40, 50)`. Note that if the value range is empty, it will generate an expression representing a false value.
 - `dbx.NotIn()`: creating an `NOT IN` expression. This is very similar to `dbx.In()`.
 - `dbx.Like()`: creating a `LIKE` expression for the specified column and the range of values. For example, `dbx.Like("title", "golang", "framework")` would create the expression `title LIKE "%golang%" AND title LIKE "%framework%"`. You can further customize a `LIKE` expression by calling `Escape()` and/or `Match()` functions of the resulting expression. Note that if the value range is empty, it will generate an empty expression.
 - `dbx.NotLike()`: creating a `NOT LIKE` expression. This is very similar to `dbx.Like()`.
 - `dbx.OrElseLike()`: creating a `LIKE` expression but concatenating different `LIKE` sub-expressions using `OR` instead of `AND`.
 - `dbx.OrNotLike()`: creating a `NOT LIKE` expression and concatenating different `NOT LIKE` sub-expressions using `OR` instead of `AND`.
 - `dbx.Exists()`: creating an `EXISTS` expression by prepending `EXISTS` to the given expression.
 - `dbx.NotExists()`: creating a `NOT EXISTS` expression by prepending `NOT EXISTS` to the given expression.
 - `dbx.Between()`: creating a `BETWEEN` expression. For example, `dbx.Between("age", 30, 40)` would create the expression `age BETWEEN 30 AND 40`.
 - `dbx.NotBetween()`: creating a `NOT BETWEEN` expression. For example

You may also create other convenient functions to help building query conditions, as long as the functions return an object implementing the `dbx.Expression` interface.

Building Data Manipulation Queries

Data manipulation queries are those changing the data in the database, such as `INSERT`, `UPDATE`, `DELETE` statements. Such queries can be built by calling the corresponding methods of `DB`. For example,

```
1 db, _ := dbx.Open("mysql", "user:pass@example")
2
3 // INSERT INTO `users` (`name`, `email`) VALUES (:{:p0}, {:p1})
4 err := db.Insert("users", dbx.Params{
5     "name": "James",
6     "email": "james@example.com",
7 }).Execute()
8
9 // UPDATE `users` SET `status`={:p0} WHERE `id`={:p1}
```

```
10 err = db.Update("users", dbx.Params{"status": 1}, dbx.HashExp{"id":
    100}).Execute()
11
12 // DELETE FROM `users` WHERE `status`=:p0}
13 err = db.Delete("users", dbx.HashExp{"status": 2}).Execute()
```

When building data manipulation queries, remember to call `Execute()` at the end to execute the queries.

Building Schema Manipulation Queries

Schema manipulation queries are those changing the database schema, such as creating a new table, adding a new column. These queries can be built by calling the corresponding methods of `DB`. For example,

```
1 db, _ := dbx.Open("mysql", "user:pass@example")
2
3 // CREATE TABLE `users` (`id` int primary key, `name` varchar(255))
4 q := db.CreateTable("users", map[string]string{
5     "id": "int primary key",
6     "name": "varchar(255)",
7 })
8 err := q.Execute()
```

CRUD Operations

Although `ozzo-dbx` is not an ORM, it does provide a very convenient way to do typical CRUD (Create, Read, Update, Delete) operations without the need of writing plain SQL statements.

To use the CRUD feature, first define a struct type for a table. By default, a struct is associated with a table whose name is the snake case version of the struct type name. For example, a struct named `MyCustomer` corresponds to the table name `my_customer`. You may explicitly specify the table name for a struct by implementing the `dbx.TableModel` interface. For example,

```
1 type MyCustomer struct{}
2
3 func (c MyCustomer) TableName() string {
4     return "customer"
5 }
```

Note that the `TableName` method should be defined with a value receiver instead of a pointer receiver.

If the struct has a field named `ID` or `Id`, by default the field will be treated as the primary key field. If you want to use a different field as the primary key, tag it with `db: "pk"`. You may tag multiple fields

for composite primary keys. Note that if you also want to explicitly specify the column name for a primary key field, you should use the tag format `db:"pk,col_name"`.

You can give a common prefix or suffix to your table names by defining your own table name mapping via `DB.TableMapFunc`. For example, the following code prefixes `tbl_` to all table names.

```
1 db.TableMapper = func(a interface{}) string {
2     return "tbl_" + GetTableName(a)
3 }
```

Create

To create (insert) a new row using a model, call the `ModelQuery.Insert()` method. For example,

```
1 type Customer struct {
2     ID      int
3     Name    string
4     Email   string
5     Status  int
6 }
7
8 db, _ := dbx.Open("mysql", "user:pass@example")
9
10 customer := Customer{
11     Name: "example",
12     Email: "test@example.com",
13 }
14 // INSERT INTO customer (name, email, status) VALUES ('example', '
15     test@example.com', 0)
16 err := db.Model(&customer).Insert()
```

This will insert a row using the values from *all* public fields (except the primary key field if it is empty) in the struct. If a primary key field is zero (a integer zero or a nil pointer), it is assumed to be auto-incremental and will be automatically filled with the last insertion ID after a successful insertion.

You can explicitly specify the fields that should be inserted by passing the list of the field names to the `Insert()` method. You can also exclude certain fields from being inserted by calling `Exclude()` before calling `Insert()`. For example,

```
1 db, _ := dbx.Open("mysql", "user:pass@example")
2
3 // insert only Name and Email fields
4 err := db.Model(&customer).Insert("Name", "Email")
5 // insert all public fields except Status
6 err = db.Model(&customer).Exclude("Status").Insert()
7 // insert only Name
```

```
8 err = db.Model(&customer).Exclude("Status").Insert("Name", "Status")
```

Read

To read a model by a given primary key value, call `SelectQuery.Model()`.

```
1 db, _ := dbx.Open("mysql", "user:pass@example")
2
3 var customer Customer
4 // SELECT * FROM customer WHERE id=100
5 err := db.Select().Model(100, &customer)
6
7 // SELECT name, email FROM customer WHERE status=1 AND id=100
8 err = db.Select("name", "email").Where(dbx.HashExp{"status": 1}).Model
    (100, &customer)
```

Note that `SelectQuery.Model()` does not support composite primary keys. You should use `SelectQuery.One()` in this case. For example,

```
1 db, _ := dbx.Open("mysql", "user:pass@example")
2
3 var orderItem OrderItem
4
5 // SELECT * FROM order_item WHERE order_id=100 AND item_id=20
6 err := db.Select().Where(dbx.HashExp{"order_id": 100, "item_id": 20}).
    One(&orderItem)
```

In the above queries, we do not call `From()` to specify which table to select data from. This is because the select query automatically sets the table according to the model struct being populated. If the struct implements `TableModel`, the value returned by its `TableName()` method will be used as the table name. Otherwise, the snake case version of the struct type name will be the table name.

You may also call `SelectQuery.All()` to read a list of model structs. Similarly, you do not need to call `From()` if the table name can be inferred from the model structs.

Update

To update a model, call the `ModelQuery.Update()` method. Like `Insert()`, by default, the `Update()` method will update *all* public fields except primary key fields of the model. You can explicitly specify which fields can be updated and which cannot in the same way as described for the `Insert()` method. For example,

```
1 db, _ := dbx.Open("mysql", "user:pass@example")
2
```

```
3 // update all public fields of customer
4 err := db.Model(&customer).Update()
5 // update only Status
6 err = db.Model(&customer).Update("Status")
7 // update all public fields except Status
8 err = db.Model(&customer).Exclude("Status").Update()
```

Note that the `Update()` method assumes that the primary keys are immutable. It uses the primary key value of the model to look for the row that should be updated. An error will be returned if a model does not have a primary key.

Delete

To delete a model, call the `ModelQuery.Delete()` method. The method deletes the row using the primary key value specified by the model. If the model does not have a primary key, an error will be returned. For example,

```
1 db, _ := dbx.Open("mysql", "user:pass@example")
2
3 err := db.Model(&customer).Delete()
```

Null Handling

To represent a nullable database value, you can use a pointer type. If the pointer is nil, it means the corresponding database value is null.

Another option to represent a database null is to use `sql.NullXYZ` types. For example, if a string column is nullable, you may use `sql.NullString`. The `NullString.Valid` field indicates whether the value is a null or not, and `NullString.String` returns the string value when it is not null. Because `sql.NullXYZ` types do not handle JSON marshalling, you may use the null package, instead.

Below is an example of handling nulls:

```
1 type Customer struct {
2     ID      int
3     Email   string
4     FirstName *string // use pointer to represent null
5     LastName sql.NullString // use sql.NullString to represent null
6 }
```

Quoting Table and Column Names

Databases vary in quoting table and column names. To allow writing DB-agnostic SQLs, ozzo-dbx introduces a special syntax in quoting table and column names. A word enclosed within `{{` and `}}` is treated as a table name and will be quoted according to the particular DB driver. Similarly, a word enclosed within `[[` and `]]` is treated as a column name and will be quoted accordingly as well. For example, when working with a MySQL database, the following query will be properly quoted:

```
1 // SELECT * FROM `users` WHERE `status`=1
2 q := db.NewQuery("SELECT * FROM {{users}} WHERE [[status]]=1")
```

Note that if a table or column name contains a prefix, it will still be properly quoted. For example, `{{public.users}}` will be quoted as `"public"."users"` for PostgreSQL.

Using Transactions

You can use all aforementioned query execution and building methods with transaction. For example,

```
1 db, _ := dbx.Open("mysql", "user:pass@example")
2
3 tx, _ := db.Begin()
4
5 _, err1 := tx.Insert("users", dbx.Params{
6     "name": "user1",
7 }).Execute()
8 _, err2 := tx.Insert("users", dbx.Params{
9     "name": "user2",
10 }).Execute()
11
12 if err1 == nil && err2 == nil {
13     tx.Commit()
14 } else {
15     tx.Rollback()
16 }
```

You may use `DB.Transactional()` to simplify your transactional code without explicitly committing or rolling back transactions. The method will start a transaction and automatically roll back the transaction if the callback returns an error. Otherwise it will automatically commit the transaction.

```
1 db, _ := dbx.Open("mysql", "user:pass@example")
2
3 err := db.Transactional(func(tx *dbx.Tx) error {
4     var err error
5     _, err = tx.Insert("users", dbx.Params{
6         "name": "user1",
```

```

7     }).Execute()
8     if err != nil {
9         return err
10    }
11    _, err = tx.Insert("users", dbx.Params{
12        "name": "user2",
13    }).Execute()
14    return err
15 })
16
17 fmt.Println(err)

```

Logging Executed SQL Statements

You can log and instrument DB queries by installing loggers with a DB connection. There are three kinds of loggers you can install: * `DB.LogFunc`: this is called each time when a SQL statement is queried or executed. The function signature is the same as that of `fmt.Printf`, which makes it very easy to use. * `DB.QueryLogFunc`: this is called each time when querying with a SQL statement. * `DB.ExecLogFunc`: this is called when executing a SQL statement.

The following example shows how you can make use of these loggers.

```

1  import (
2      "fmt"
3      "log"
4      "github.com/go-ozzo/ozzo-dbx"
5  )
6
7  func main() {
8      db, _ := dbx.Open("mysql", "user:pass@example")
9
10     // simple logging
11     db.LogFunc = log.Printf
12
13     // or you can use the following more flexible logging
14     db.QueryLogFunc = func(ctx context.Context, t time.Duration, sql
15         string, rows *sql.Rows, err error) {
16         log.Printf("[%2fms] Query SQL: %v", float64(t.Milliseconds()),
17             sql)
18     }
19     db.ExecLogFunc = func(ctx context.Context, t time.Duration, sql
20         string, result sql.Result, err error) {
21         log.Printf("[%2fms] Execute SQL: %v", float64(t.Milliseconds())
22             , sql)
23     }
24     // ...
25 }

```

Supporting New Databases

While `ozzo-dbx` provides out-of-box query building support for most major relational databases, its open architecture allows you to add support for new databases. The effort of adding support for a new database involves:

- Create a struct that implements the `QueryBuilder` interface. You may use `BaseQueryBuilder` directly or extend it via composition.
- Create a struct that implements the `Builder` interface. You may extend `BaseBuilder` via composition.
- Write an `init()` function to register the new builder in `dbx.BuilderFuncMap`.