

---

## Go Relational Persistence



### Update 2016-11-13: Future versions

As many of the maintainers have become busy with other projects, progress toward the ever-elusive v2 has slowed to the point that we're only occasionally making progress outside of merging pull requests. In the interest of continuing to release, I'd like to lean toward a more maintainable path forward.

For the moment, I am releasing a v2 tag with the current feature set from master, as some of those features have been actively used and relied on by more than one project. Our next goal is to continue cleaning up the code base with non-breaking changes as much as possible, but if/when a breaking change is needed, we'll just release new versions. This allows us to continue development at whatever pace we're capable of, without delaying the release of features or refusing PRs.

### Introduction

I hesitate to call gorp an ORM. Go doesn't really have objects, at least not in the classic Smalltalk/Java sense. There goes the "O". gorp doesn't know anything about the relationships between your structs (at least not yet). So the "R" is questionable too (but I use it in the name because, well, it seemed more clever).

The "M" is alive and well. Given some Go structs and a database, gorp should remove a fair amount of boilerplate busy-work from your code.

I hope that gorp saves you time, minimizes the drudgery of getting data in and out of your database, and helps your code focus on algorithms, not infrastructure.

- Bind struct fields to table columns via API or tag
- Support for embedded structs
- Support for transactions
- Forward engineer db schema from structs (great for unit tests)
- Pre/post insert/update/delete hooks
- Automatically generate insert/update/delete statements for a struct
- Automatic binding of auto increment PKs back to struct after insert
- Delete by primary key(s)
- Select by primary key(s)
- Optional trace sql logging
- Bind arbitrary SQL queries to a struct

- 
- Bind slice to SELECT query results without type assertions
  - Use positional or named bind parameters in custom SELECT queries
  - Optional optimistic locking using a version column (for update/deletes)

## Installation

Use `go get` or your favorite vendoring tool, using whichever import path you'd like.

## Versioning

We use semantic version tags. Feel free to import through `gopkg.in` (e.g. `gopkg.in/gorp.v2`) to get the latest tag for a major version, or check out the tag using your favorite vendoring tool.

Development is not very active right now, but we have plans to restructure `gorp` as we continue to move toward a more extensible system. Whenever a breaking change is needed, the major version will be bumped.

The `master` branch is where all development is done, and breaking changes may happen from time to time. That said, if you want to live on the bleeding edge and are comfortable updating your code when we make a breaking change, you may use `github.com/go-gorp/gorp` as your import path.

Check the version tags to see what's available. We'll make a good faith effort to add badges for new versions, but we make no guarantees.

## Supported Go versions

This package is guaranteed to be compatible with the latest 2 major versions of Go.

Any earlier versions are only supported on a best effort basis and can be dropped any time. Go has a great compatibility promise. Upgrading your program to a newer version of Go should never really be a problem.

## Migration guide

**Pre-v2 to v2** Automatic mapping of the version column used in optimistic locking has been removed as it could cause problems if the type was not int. The version column must now explicitly be set with `tablemap.SetVersionCol()`.

---

## Help/Support

Use our [gitter](#) channel. We used to use IRC, but with most of us being pulled in many directions, we often need the email notifications from [gitter](#) to yell at us to sign in.

## Quickstart

```
1 package main
2
3 import (
4     "database/sql"
5     "gopkg.in/gorp.v1"
6     _ "github.com/mattn/go-sqlite3"
7     "log"
8     "time"
9 )
10
11 func main() {
12     // initialize the DbMap
13     dbmap := initDb()
14     defer dbmap.Db.Close()
15
16     // delete any existing rows
17     err := dbmap.TruncateTables()
18     checkErr(err, "TruncateTables failed")
19
20     // create two posts
21     p1 := newPost("Go 1.1 released!", "Lorem ipsum lorem ipsum")
22     p2 := newPost("Go 1.2 released!", "Lorem ipsum lorem ipsum")
23
24     // insert rows - auto increment PKs will be set properly after the
25     // insert
26     err = dbmap.Insert(&p1, &p2)
27     checkErr(err, "Insert failed")
28
29     // use convenience SelectInt
30     count, err := dbmap.SelectInt("select count(*) from posts")
31     checkErr(err, "select count(*) failed")
32     log.Println("Rows after inserting:", count)
33
34     // update a row
35     p2.Title = "Go 1.2 is better than ever"
36     count, err = dbmap.Update(&p2)
37     checkErr(err, "Update failed")
38     log.Println("Rows updated:", count)
39
40     // fetch one row - note use of "post_id" instead of "Id" since
41     // column is aliased
```

---

```

40     //
41     // Postgres users should use $1 instead of ? placeholders
42     // See 'Known Issues' below
43     //
44     err = dbmap.SelectOne(&p2, "select * from posts where post_id=?",
45         p2.Id)
46     checkErr(err, "SelectOne failed")
47     log.Println("p2 row:", p2)
48
49     // fetch all rows
50     var posts []Post
51     _, err = dbmap.Select(&posts, "select * from posts order by post_id")
52     checkErr(err, "Select failed")
53     log.Println("All rows:")
54     for x, p := range posts {
55         log.Printf("    %d: %v\n", x, p)
56     }
57
58     // delete row by PK
59     count, err = dbmap.Delete(&p1)
60     checkErr(err, "Delete failed")
61     log.Println("Rows deleted:", count)
62
63     // delete row manually via Exec
64     _, err = dbmap.Exec("delete from posts where post_id=?", p2.Id)
65     checkErr(err, "Exec failed")
66
67     // confirm count is zero
68     count, err = dbmap.SelectInt("select count(*) from posts")
69     checkErr(err, "select count(*) failed")
70     log.Println("Row count - should be zero:", count)
71
72     log.Println("Done!")
73 }
74
75 type Post struct {
76     // db tag lets you specify the column name if it differs from the
77     // struct field
78     Id      int64 `db:"post_id"`
79     Created int64
80     Title   string `db:",size:50"` // Column size set to
81     // 50
82     Body    string `db:"article_body,size:1024"` // Set both column
83     // name and size
84 }
85
86 func newPost(title, body string) Post {
87     return Post{
88         Created: time.Now().UnixNano(),
89         Title:   title,

```

---

---

```

86         Body:    body,
87     }
88 }
89
90 func initDb() *gorp.DbMap {
91     // connect to db using standard Go database/sql API
92     // use whatever database/sql driver you wish
93     db, err := sql.Open("sqlite3", "/tmp/post_db.bin")
94     checkErr(err, "sql.Open failed")
95
96     // construct a gorp DbMap
97     dbmap := &gorp.DbMap{Db: db, Dialect: gorp.SqliteDialect{}}
98
99     // add a table, setting the table name to 'posts' and
100    // specifying that the Id property is an auto incrementing PK
101    dbmap.AddTableWithName(Post{}, "posts").SetKeys(true, "Id")
102
103    // create the table. in a production system you'd generally
104    // use a migration tool, or create the tables via scripts
105    err = dbmap.CreateTablesIfNotExists()
106    checkErr(err, "Create tables failed")
107
108    return dbmap
109 }
110
111 func checkErr(err error, msg string) {
112     if err != nil {
113         log.Fatalln(msg, err)
114     }
115 }
```

## Examples

### Mapping structs to tables

First define some types:

```

1  type Invoice struct {
2      Id      int64
3      Created int64
4      Updated int64
5      Memo    string
6      PersonId int64
7  }
8
9  type Person struct {
10     Id      int64
11     Created int64
12     Updated int64
```

---

```

13     FName    string
14     LName    string
15 }
16
17 // Example of using tags to alias fields to column names
18 // The 'db' value is the column name
19 //
20 // A hyphen will cause gorp to skip this field, similar to the
21 // Go json package.
22 //
23 // This is equivalent to using the ColMap methods:
24 //
25 //     table := dbmap.AddTableWithName(Product{}, "product")
26 //     table.ColMap("Id").Rename("product_id")
27 //     table.ColMap("Price").Rename("unit_price")
28 //     table.ColMap("IgnoreMe").SetTransient(true)
29 //
30 // You can optionally declare the field to be a primary key and/or
31 //     autoincrement
32 type Product struct {
33     Id          int64      `db:"product_id, primaryKey, autoincrement"`
34     Price       int64      `db:"unit_price"`
35     IgnoreMe    string     `db:"-"`
36 }

```

Then create a mapper, typically you'd do this one time at app startup:

```

1 // connect to db using standard Go database/sql API
2 // use whatever database/sql driver you wish
3 db, err := sql.Open("mymysql", "tcp:localhost:3306*mydb/myuser/
4     mypassword")
5
6 // construct a gorp DbMap
7 dbmap := &gorp.DbMap{Db: db, Dialect: gorp.MySQLDialect{"InnoDB", "UTF8
8     "}}
9
10 // register the structs you wish to use with gorp
11 // you can also use the shorter dbmap.AddTable() if you
12 // don't want to override the table name
13 //
14 // SetKeys(true) means we have a auto increment primary key, which
15 // will get automatically bound to your struct post-insert
16 //
17 t1 := dbmap.AddTableWithName(Invoice{}, "invoice_test").SetKeys(true, "
18     Id")
19 t2 := dbmap.AddTableWithName(Person{}, "person_test").SetKeys(true, "Id
20     ")
21 t3 := dbmap.AddTableWithName(Product{}, "product_test").SetKeys(true, "
22     Id")

```

---

## Struct Embedding

gorp supports embedding structs. For example:

```
1 type Names struct {
2     FirstName string
3     LastName  string
4 }
5
6 type WithEmbeddedStruct struct {
7     Id int64
8     Names
9 }
10
11 es := &WithEmbeddedStruct{-1, Names{FirstName: "Alice", LastName: "
12     Smith"}}
13 err := dbmap.Insert(es)
```

See the `TestWithEmbeddedStruct` function in `gorp_test.go` for a full example.

## Create/Drop Tables

Automatically create / drop registered tables. This is useful for unit tests but is entirely optional. You can of course use gorp with tables created manually, or with a separate migration tool (like sql-migrate, goose or migrate).

```
1 // create all registered tables
2 dbmap.CreateTables()
3
4 // same as above, but uses "if not exists" clause to skip tables that
5 // are
6 // already defined
7 dbmap.CreateTablesIfNotExists()
8
9 // drop
10 dbmap.DropTable()
```

## SQL Logging

Optionally you can pass in a logger to trace all SQL statements. I recommend enabling this initially while you're getting the feel for what gorp is doing on your behalf.

Gorp defines a `GorpLogger` interface that Go's built in `log.Logger` satisfies. However, you can write your own `GorpLogger` implementation, or use a package such as `glog` if you want more control over how statements are logged.

---

```
1 // Will log all SQL statements + args as they are run
2 // The first arg is a string prefix to prepend to all log messages
3 dbmap.TraceOn("[gorp]", log.New(os.Stdout, "myapp:", log.Lmicroseconds)
4     )
5 // Turn off tracing
6 dbmap.TraceOff()
```

## Insert

```
1 // Must declare as pointers so optional callback hooks
2 // can operate on your data, not copies
3 inv1 := &Invoice{0, 100, 200, "first order", 0}
4 inv2 := &Invoice{0, 100, 200, "second order", 0}
5
6 // Insert your rows
7 err := dbmap.Insert(inv1, inv2)
8
9 // Because we called SetKeys(true) on Invoice, the Id field
10 // will be populated after the Insert() automatically
11 fmt.Printf("inv1.Id=%d inv2.Id=%d\n", inv1.Id, inv2.Id)
```

## Update

Continuing the above example, use the [Update](#) method to modify an Invoice:

```
1 // count is the # of rows updated, which should be 1 in this example
2 count, err := dbmap.Update(inv1)
```

## Delete

If you have primary key(s) defined for a struct, you can use the [Delete](#) method to remove rows:

```
1 count, err := dbmap.Delete(inv1)
```

## Select by Key

Use the [Get](#) method to fetch a single row by primary key. It returns nil if no row is found.

```
1 // fetch Invoice with Id=99
2 obj, err := dbmap.Get(Invoice{}, 99)
3 inv := obj.(*Invoice)
```



---

## Ad Hoc SQL

**SELECT** `Select()` and `SelectOne()` provide a simple way to bind arbitrary queries to a slice or a single struct.

```
1 // Select a slice - first return value is not needed when a slice
  // pointer is passed to Select()
2 var posts []Post
3 _, err := dbmap.Select(&posts, "select * from post order by id")
4
5 // You can also use primitive types
6 var ids []string
7 _, err := dbmap.Select(&ids, "select id from post")
8
9 // Select a single row.
10 // Returns an error if no row found, or if more than one row is found
11 var post Post
12 err := dbmap.SelectOne(&post, "select * from post where id=?", id)
```

Want to do joins? Just write the SQL and the struct. gorp will bind them:

```
1 // Define a type for your join
2 // It *must* contain all the columns in your SELECT statement
3 //
4 // The names here should match the aliased column names you specify
5 // in your SQL - no additional binding work required. simple.
6 //
7 type InvoicePersonView struct {
8     InvoiceId    int64
9     PersonId     int64
10    Memo         string
11    FName        string
12 }
13
14 // Create some rows
15 p1 := &Person{0, 0, 0, "bob", "smith"}
16 err = dbmap.Insert(p1)
17 checkErr(err, "Insert failed")
18
19 // notice how we can wire up p1.Id to the invoice easily
20 inv1 := &Invoice{0, 0, 0, "xmas order", p1.Id}
21 err = dbmap.Insert(inv1)
22 checkErr(err, "Insert failed")
23
24 // Run your query
25 query := "select i.Id InvoiceId, p.Id PersonId, i.Memo, p.FName " +
26         "from invoice_test i, person_test p " +
27         "where i.PersonId = p.Id"
28
29 // pass a slice to Select()
```

---

```

30 var list []InvoicePersonView
31 _, err := dbmap.Select(&list, query)
32
33 // this should test true
34 expected := InvoicePersonView{inv1.Id, p1.Id, inv1.Memo, p1.FName}
35 if reflect.DeepEqual(list[0], expected) {
36     fmt.Println("Woot! My join worked!")
37 }

```

**SELECT string or int64** gorp provides a few convenience methods for selecting a single string or int64.

```

1 // select single int64 from db (use $1 instead of ? for postgresql)
2 i64, err := dbmap.SelectInt("select count(*) from foo where blah=?",
    blahVal)
3
4 // select single string from db:
5 s, err := dbmap.SelectStr("select name from foo where blah=?", blahVal)

```

**Named bind parameters** You may use a map or struct to bind parameters by name. This is currently only supported in SELECT queries.

```

1 _, err := dbm.Select(&dest, "select * from Foo where name = :name and
    age = :age", map[string]interface{}{
2     "name": "Rob",
3     "age": 31,
4 })

```

**UPDATE / DELETE** You can execute raw SQL if you wish. Particularly good for batch operations.

```

1 res, err := dbmap.Exec("delete from invoice_test where PersonId=?", 10)

```

## Transactions

You can batch operations into a transaction:

```

1 func InsertInv(dbmap *DbMap, inv *Invoice, per *Person) error {
2     // Start a new transaction
3     trans, err := dbmap.Begin()
4     if err != nil {
5         return err
6     }
7
8     err = trans.Insert(per)

```

---

```

 9     checkErr(err, "Insert failed")
10
11     inv.PersonId = per.Id
12     err = trans.Insert(inv)
13     checkErr(err, "Insert failed")
14
15     // if the commit is successful, a nil error is returned
16     return trans.Commit()
17 }

```

## Hooks

Use hooks to update data before/after saving to the db. Good for timestamps:

```

1 // implement the PreInsert and PreUpdate hooks
2 func (i *Invoice) PreInsert(s gorp.SqlExecutor) error {
3     i.Created = time.Now().UnixNano()
4     i.Updated = i.Created
5     return nil
6 }
7
8 func (i *Invoice) PreUpdate(s gorp.SqlExecutor) error {
9     i.Updated = time.Now().UnixNano()
10    return nil
11 }
12
13 // You can use the SqlExecutor to cascade additional SQL
14 // Take care to avoid cycles. gorp won't prevent them.
15 //
16 // Here's an example of a cascading delete
17 //
18 func (p *Person) PreDelete(s gorp.SqlExecutor) error {
19     query := "delete from invoice_test where PersonId=?"
20
21     _, err := s.Exec(query, p.Id)
22
23     if err != nil {
24         return err
25     }
26     return nil
27 }

```

Full list of hooks that you can implement:

```

1 PostGet
2 PreInsert
3 PostInsert
4 PreUpdate
5 PostUpdate

```

---

```
6 PreDelete
7 PostDelete
8
9 All have the same signature. for example:
10
11 func (p *MyStruct) PostUpdate(s gorp.SqlExecutor) error
```

## Optimistic Locking

**Note that this behaviour has changed in v2. See Migration Guide.** gorp provides a simple optimistic locking feature, similar to Java's JPA, that will raise an error if you try to update/delete a row whose `version` column has a value different than the one in memory. This provides a safe way to do "select then update" style operations without explicit read and write locks.

```
1 // Version is an auto-incremented number, managed by gorp
2 // If this property is present on your struct, update
3 // operations will be constrained
4 //
5 // For example, say we defined Person as:
6
7 type Person struct {
8     Id      int64
9     Created int64
10    Updated int64
11    FName   string
12    LName   string
13
14    // automatically used as the Version col
15    // use table.SetVersionCol("columnName") to map a different
16    // struct field as the version field
17    Version int64
18 }
19
20 p1 := &Person{0, 0, 0, "Bob", "Smith", 0}
21 err = dbmap.Insert(p1) // Version is now 1
22 checkErr(err, "Insert failed")
23
24 obj, err := dbmap.Get(Person{}, p1.Id)
25 p2 := obj.(*Person)
26 p2.LName = "Edwards"
27 _,err = dbmap.Update(p2) // Version is now 2
28 checkErr(err, "Update failed")
29
30 p1.LName = "Howard"
31
32 // Raises error because p1.Version == 1, which is out of date
33 count, err := dbmap.Update(p1)
34 _, ok := err.(gorp.OptimisticLockError)
```

---

```

35 if ok {
36     // should reach this statement
37
38     // in a real app you might reload the row and retry, or
39     // you might propagate this to the user, depending on the desired
40     // semantics
41     fmt.Printf("Tried to update row with stale data: %v\n", err)
42 } else {
43     // some other db error occurred - log or return up the stack
44     fmt.Printf("Unknown db err: %v\n", err)
45 }

```

### Adding INDEX(es) on column(s) beyond the primary key

Indexes are frequently critical for performance. Here is how to add them to your tables.

NB: SqlServer and Oracle need testing and possible adjustment to the CreateIndexSuffix() and DropIndexSuffix() methods to make AddIndex() work for them.

In the example below we put an index both on the Id field, and on the AcctId field.

```

1 type Account struct {
2     Id      int64
3     AcctId  string // e.g. this might be a long uuid for portability
4 }
5
6 // indexType (the 2nd param to AddIndex call) is "Btree" or "Hash" for
7 // MySQL.
8 // demonstrate adding a second index on AcctId, and constrain that
9 // field to have unique values.
10 dbm.AddTable(iptab.Account{}).SetKeys(true, "Id").AddIndex("AcctIdIndex",
11     "Btree", []string{"AcctId"}).SetUnique(true)
12
13 err = dbm.CreateTablesIfNotExists()
14 checkErr(err, "CreateTablesIfNotExists failed")
15
16 err = dbm.CreateIndex()
17 checkErr(err, "CreateIndex failed")

```

Check the effect of the CreateIndex() call in mysql:

```

1 $ mysql
2
3 MariaDB [test]> show create table Account;
4 +-----+
5 | Account | CREATE TABLE `Account` (
6 | `Id` bigint(20) NOT NULL AUTO_INCREMENT,
7 | `AcctId` varchar(255) DEFAULT NULL,
8 | PRIMARY KEY (`Id`),

```

---

```
9    UNIQUE KEY `AcctIdIndex` (`AcctId`) USING BTREE    <<<---- yes! index
      added.
10 ) ENGINE=InnoDB DEFAULT CHARSET=utf8
11 +-----+-----+-----+-----+-----+-----+
```

## Database Drivers

gorp uses the Go 1 [database/sql](#) package. A full list of compliant drivers is available here:

<http://code.google.com/p/go-wiki/wiki/SQLDrivers>

Sadly, SQL databases differ on various issues. gorp provides a Dialect interface that should be implemented per database vendor. Dialects are provided for:

- MySQL
- PostgreSQL
- sqlite3

Each of these three databases pass the test suite. See [gorp\\_test.go](#) for example DSNs for these three databases.

Support is also provided for:

- Oracle (contributed by @klaidliadon)
- SQL Server (contributed by @qrawl) - use driver: [github.com/denisenkong/go-mssqldb](https://github.com/denisenkong/go-mssqldb)

Note that these databases are not covered by CI and I (@coopernurse) have no good way to test them locally. So please try them and send patches as needed, but expect a bit more unpredictability.

## Sqlite3 Extensions

In order to use sqlite3 extensions you need to first register a custom driver:

```
1 import (
2     "database/sql"
3
4     // use whatever database/sql driver you wish
5     sqlite "github.com/mattn/go-sqlite3"
6 )
7
8 func customDriver() (*sql.DB, error) {
9
10    // create custom driver with extensions defined
11    sql.Register("sqlite3-custom", &sqlite.SQLiteDriver{
12        Extensions: []string{
```

---

```

13         "mod_spatialite",
14     },
15 })
16
17     // now you can then connect using the 'sqlite3-custom' driver
18     // instead of 'sqlite3'
19     return sql.Open("sqlite3-custom", "/tmp/post_db.bin")

```

## Known Issues

### SQL placeholder portability

Different databases use different strings to indicate variable placeholders in prepared SQL statements. Unlike some database abstraction layers (such as JDBC), Go's `database/sql` does not standardize this.

SQL generated by gorp in the `Insert`, `Update`, `Delete`, and `Get` methods delegates to a Dialect implementation for each database, and will generate portable SQL.

Raw SQL strings passed to `Exec`, `Select`, `SelectOne`, `SelectInt`, etc will not be parsed. Consequently you may have portability issues if you write a query like this:

```

1 // works on MySQL and Sqlite3, but not with PostgreSQL err :=
2 dbmap.SelectOne(&val, "select * from foo where id = ?", 30)

```

In `Select` and `SelectOne` you can use named parameters to work around this. The following is portable:

```

1 err := dbmap.SelectOne(&val, "select * from foo where id = :id",
2 map[string]interface{} { "id": 30})

```

Additionally, when using Postgres as your database, you should utilize `$1` instead of `?` placeholders as utilizing `?` placeholders when querying Postgres will result in `pq: operator does not exist` errors. Alternatively, use `dbMap.Dialect.BindVar(varIdx)` to get the proper variable binding for your dialect.

### time.Time and time zones

gorp will pass `time.Time` fields through to the `database/sql` driver, but note that the behavior of this type varies across database drivers.

MySQL users should be especially cautious. See: <https://github.com/ziutek/mymysql/pull/77>

---

To avoid any potential issues with timezone/DST, consider:

- Using an integer field for time data and storing UNIX time.
- Using a custom time type that implements some SQL types:
  - `"database/sql".Scanner`
  - `"database/sql/driver".Valuer`

## Running the tests

The included tests may be run against MySQL, Postgresql, or sqlite3. You must set two environment variables so the test code knows which driver to use, and how to connect to your database.

```
1 # MySQL example:
2 export GORP_TEST_DSN=gomysql_test/gomysql_test/abc123
3 export GORP_TEST_DIALECT=mysql
4
5 # run the tests
6 go test
7
8 # run the tests and benchmarks
9 go test -bench="Bench" -benchtime 10
```

Valid `GORP_TEST_DIALECT` values are: “mysql”(for mymysql), “gomysql”(for go-sql-driver), “postgres”, “sqlite” See the `test_all.sh` script for examples of all 3 databases. This is the script I run locally to test the library.

## Performance

gorp uses reflection to construct SQL queries and bind parameters. See the `BenchmarkNativeCrud` vs `BenchmarkGorpCrud` in `gorp_test.go` for a simple perf test. On my MacBook Pro gorp is about 2-3% slower than hand written SQL.

## Contributors

- matthias-margush - column aliasing via tags
- Rob Figueiredo - @robfig
- Quinn Slack - @sq