Fork me on GitHub

Illustrated guide to SQLX

sqlx is a package for Go which provides a set of extensions on top of the excellent built-in **database/sql** package.

Examining *Go* idioms is the focus of this document, so there is no presumption being made that any *SQL* herein is actually a recommended way to use a database. It will not cover setting up a Go development environment, basic Go information about syntax or semantics, or SQL itself.

Finally, the standard **err** variable will be used to indicate that errors are being returned, but for brevity they will be ignored. You should make sure to check all errors in an actual program.

Resources

There are other resources of excellent information about using SQL in Go:

- database/sql documentation
- go-database-sql tutorial

If you need help getting started with Go itself, I recommend these resources:

- The Go tour
- How to write Go code
- Effective Go

Because the **database/sql** interface is a subset of sqlx, all of the advice in these documents about **database/sql** usage also apply to usage of sqlx.

Getting Started

You will want to install **sqlx** and a database driver. Since it's infrastructureless, I recommend mattn's sqlite3 driver to start out with:

```
$ go get github.com/jmoiron/sqlx
$ go get github.com/mattn/go-sqlite3
```

Handle Types

sqlx is intended to have the same *feel* as **database/sql**. There are 4 main *handle* types:

- **sqlx.DB** - analagous to **sql.DB**, a representation of a database
- **sqlx.Tx** - analagous to **sql.Tx**, a representation of a transaction
- **sqlx.Stmt** - analagous to **sql.Stmt**, a representation of a prepared statement
- **sqlx.NamedStmt** - a representation of a prepared statement with support for named parameters

Handle types all embed their **database/sql** equivalents, meaning that when you call **sqlx.DB.Query**, you are calling the *same* code as **sql.DB.Query**. This makes it easy to introduce into an existing codebase.

In addition to these, there are 2 *cursor* types:

- **`sqlx.Rows`** - analagous to **`sql.Rows`**, a cursor returned from **`Queryx`**
- **`sqlx.Row`** - analagous to **`sql.Row`**, a result returned from **`QueryRowx`**

As with the handle types, **`sqlx.Rows`** embeds **`sql.Rows`**. Because the underlying implementation was inaccessible, **`sqlx.Row`** is a partial re-implementation of **`sql.Row`** that retains the standard interface.

Connecting to Your Database

A **`DB`** instance is *not* a connection, but an abstraction representing a Database. This is why creating a DB does not return an error and will not panic. It maintains a connection pool internally, and will attempt to connect when a connection is first needed. You can create an sqlx.DB via **`Open`** or by creating a new sqlx DB handle from an existing sql.DB via **`NewDb`**:

```
var db *sqlx.DB

// exactly the same as the built-in
db = sqlx.Open("sqlite3", ":memory:")

// from a pre-existing sql.DB; note the required driverName
db = sqlx.NewDb(sql.Open("sqlite3", ":memory:"), "sqlite3")

// force a connection and test that it worked
err = db.Ping()
```

In some situations, you might want to open a DB and connect at the same time; for instance, in order to catch configuration issues during your initialization phase. You can do this in one go with **`Connect`**, which Opens a new DB and attempts a **`Ping`**. The **`MustConnect`** variant will panic when encountering an error, suitable for use at the module level of your package.

```
var err error
// open and connect at the same time:
db, err = sqlx.Connect("sqlite3", ":memory:")

// open and connect at the same time, panicing on error
db = sqlx.MustConnect("sqlite3", ":memory:")
```

Querying 101

The handle types in sqlx implement the same basic verbs for querying your database:

- **`Exec(...) (sql.Result, error)`** - unchanged from database/sql
- **`Query(...) (*sql.Rows, error)`** - unchanged from database/sql
- **`QueryRow(...) *sql.Row`** - unchanged from database/sql

These extensions to the built-in verbs:

- **`MustExec() sql.Result`** -- Exec, but panic on error
- **`Queryx(...) (*sqlx.Rows, error)`** - Query, but return an sqlx.Rows
- **`QueryRowx(...) *sqlx.Row`** -- QueryRow, but return an sqlx.Row

And these new semantics:

- **`Get(dest interface{}, ...) error`**
- **`Select(dest interface{}, ...) error`**

Let's go from the unchanged interface through the new semantics, explaining their use.

Exec

Exec and MustExec get a connection from the connection pool and executes the provided query on the server. For drivers that do not support ad-hoc query execution, a prepared statement *may* be created behind the scenes to be executed. The connection is returned to the pool before the result is returned.

```
schema := `CREATE TABLE place (
    country text,
    city text NULL,
    telcode integer);`

// execute a query on the server
result, err := db.Exec(schema)

// or, you can use MustExec, which panics on error
cityState := `INSERT INTO place (country, telcode) VALUES (?, ?)`
countryCity := `INSERT INTO place (country, city, telcode) VALUES (?, ?, ?)`
db.MustExec(cityState, "Hong Kong", 852)
db.MustExec(cityState, "Singapore", 65)
db.MustExec(countryCity, "South Africa", "Johannesburg", 27)
```

The result has two possible pieces of data: **`LastInsertId()`** or **`RowsAffected()`**, the availability of which is driver dependent. In MySQL, for instance, **`LastInsertId()`** will be available on inserts with an auto-increment key, but in PostgreSQL, this information can only be retrieved from a normal row cursor by using the **`RETURNING`** clause.

bindvars

The **`?`** query placeholders, called **`bindvars`** internally, are important; you should *always* use these to send values to the database, as they will prevent SQL injection attacks. database/sql does not attempt *any* validation on the query text; it is sent to the server as is, along with the encoded parameters. Unless drivers implement a special interface, the query is prepared on the server first before execution. Bindvars are therefore database specific:

- MySQL uses the **`?`** variant shown above
- PostgreSQL uses an enumerated **`$1`**, **`$2`**, etc bindvar syntax
- SQLite accepts both **`?`** and **`$1`** syntax
- Oracle uses a **`:name`** syntax

Other databases may vary. You can use the **`sqlx.DB.Rebind(string) string`** function with the **`?`** bindvar syntax to get a query which is suitable for execution on your current database type.

A common misconception with bindvars is that they are used for interpolation. They are only for *parameterization*, and are not allowed to change the structure of an SQL statement. For instance, using bindvars to try and parameterize column or table names will not work:

```
// doesn't work
db.Query("SELECT * FROM ?", "mytable")
```

```
// also doesn't work
db.Query("SELECT ?, ? FROM people", "name", "location")
```

Query

Query is the primary way to run queries with database/sql that return row results. Query returns an **sql.Rows** object and an error:

```
// fetch all places from the db
rows, err := db.Query("SELECT country, city, telcode FROM place")

// iterate over each row
for rows.Next() {
    var country string
    // note that city can be NULL, so we use the NullString type
    var city    sql.NullString
    var telcode int
    err = rows.Scan(&country, &city, &telcode)
}
// check the error from rows
err = rows.Err()
```

You should treat the Rows like a database cursor rather than a materialized list of results. Although driver buffering behavior can vary, iterating via **Next()** is a good way to bound the memory usage of large result sets, as you're only scanning a single row at a time. **Scan()** uses reflect to map sql column return types to Go types like **string**, **[]byte**, et al. If you do not iterate over a whole rows result, be sure to call **rows.Close()** to return the connection back to the pool!

The error returned by Query is any error that might have happened while preparing or executing on the server. This can include grabbing a bad connection from the pool, although database/sql will retry 10 times to attempt to find or create a working connection. Generally, the error will be due to bad SQL syntax, type mismatches, or incorrect field and table names.

In most cases, Rows.Scan will copy the data it gets from the driver, as it is not aware of how the driver may reuse its buffers. The special type **sql.RawBytes** can be used to get a *zero-copy* slice of bytes from the actual data returned by the driver. After the next call to Next(), such a value is no longer valid, as that memory might have been overwritten by the driver.

The connection used by the Query remains active until *either* all rows are exhausted by the iteration via Next, or **rows.Close()** is called, at which point it is released. For more information, see the section on the connection pool.

The sqlx extension **Queryx** behaves exactly as Query does, but returns an **sqlx.Rows**, which has extended scanning behaviors:

```
type Place struct {
    Country       string
    City          sql.NullString
    TelephoneCode int `db:"telcode"`
}

rows, err := db.Queryx("SELECT * FROM place")
```

```
for rows.Next() {
    var p Place
    err = rows.StructScan(&p)
}
```

The primary extension on sqlx.Rows is **`StructScan()`**, which automatically scans results into struct fields. Note that the fields must be exported (capitalized) in order for sqlx to be able to write into them, something true of *all* marshallers in Go. You can use the **`db`** struct tag to specify which column name maps to each struct field, or set a new default mapping with db.MapperFunc(). The default behavior is to use **`strings.Lower`** on the field name to match against the column names. For more information about **`StructScan`**, **`SliceScan`**, and **`MapScan`**, see the section on advanced scanning.

QueryRow

QueryRow fetches one row from the server. It takes a connection from the connection pool and executes the query using Query, returning a **`Row`** object which has its own internal Rows object:

```
row := db.QueryRow("SELECT * FROM place WHERE telcode=?", 852)
var telcode int
err = row.Scan(&telcode)
```

Unlike Query, QueryRow returns a Row type result with no error, making it safe to chain the Scan off of the return. If there was an error executing the query, that error is returned by Scan. If there are no rows, Scan returns **`sql.ErrNoRows`**. If the scan itself fails (eg. due to type mismatch), that error is also returned.

The Rows struct internal to the Row result is Closed upon Scan, meaning that the connection used by QueryRow is kept open until the result is scanned. It also means that **`sql.RawBytes`** is not usable here, since the referenced memory belongs to the driver and may already be invalid by the time control is returned to the caller.

The sqlx extension **`QueryRowx`** will return an sqlx.Row instead of an sql.Row, and it implements the same scanning extensions as Rows, outlined above and in the advanced scanning section:

```
var p Place
err := db.QueryRowx("SELECT city, telcode FROM place LIMIT 1").StructScan(&p)
```

Get and Select

`Get` and **`Select`** are time saving extensions to the handle types. They combine the execution of a query with flexible scanning semantics. To explain them clearly, we have to talk about what it means to be **`scannable`**:

- a value is scannable if it is not a struct, eg **`string`**, **`int`**
- a value is scannable if it implements **`sql.Scanner`**
- a value is scannable if it is a struct with no exported fields (eg. **`time.Time`**)

`Get` and **`Select`** use **`rows.Scan`** on scannable types and **`rows.StructScan`** on non-scannable types. They are roughly analagous to **`QueryRow`** and **`Query`**, where Get is useful for fetching a single result and scanning it, and Select is useful for fetching a slice of results:

```
p := Place{}
```

```
pp := []Place{}

// this will pull the first place directly into p
err = db.Get(&p, "SELECT * FROM place LIMIT 1")

// this will pull places with telcode > 50 into the slice pp
err = db.Select(&pp, "SELECT * FROM place WHERE telcode > ?", 50)

// they work with regular types as well
var id int
err = db.Get(&id, "SELECT count(*) FROM place")

// fetch at most 10 place names
var names []string
err = db.Select(&names, "SELECT name FROM place LIMIT 10")
```

Get and Select both will close the Rows they create during query execution, and will return any error encountered at any step of the process. Since they use StructScan internally, the details in the advanced scanning section also apply to Get and Select.

Select can save you a lot of typing, but beware! It's semantically different from **`Queryx`**, since it will load the entire result set into memory at once. If that set is not bounded by your query to some reasonable size, it might be best to use the classic Queryx/StructScan iteration instead.

Transactions

To use transactions, you must create a transaction handle with **`DB.Begin()`**. Code like this **will not work**:

```
// this will not work if connection pool > 1
db.MustExec("BEGIN;")
db.MustExec(...)
db.MustExec("COMMIT;")
```

Remember, Exec and all other query verbs will ask the DB for a connection and then return it to the pool each time. There's no guarantee that you will receive the same connection that the BEGIN statement was executed on. To use transactions, you must therefore use **`DB.Begin()`**

```
tx, err := db.Begin()
err = tx.Exec(...)
err = tx.Commit()
```

The DB handle also has the extensions **`Beginx()`** and **`MustBegin()`**, which return an **`sqlx.Tx`** instead of an **`sql.Tx`**:

```
tx := db.MustBegin()
tx.MustExec(...)
err = tx.Commit()
```

`sqlx.Tx` has all of the handle extensions that **`sqlx.DB`** has.

Since transactions are connection state, the Tx object must bind and control a single connection from the pool. A Tx will maintain that single connection for its entire life cycle, releasing it only when **`Commit()`** or **`Rollback()`** is called. You should take care to call at least one of these, or else the connection will be held until garbage collection.

Because you only have one connection to use in a transaction, you can only execute one statement at a time; the cursor types Row and Rows must be Scanned or Closed, respectively, before executing another query. If you attempt to send the server data while it is sending you a result, it can potentially corrupt the connection.

Finally, Tx objects do not actually imply any behavior on the server; they merely execute a BEGIN statement and bind a single connection. The actual behavior of the transaction, including things like locking and isolation, is completely unspecified and database dependent.

Prepared Statements

On most databases, statements will actually be prepared behind the scenes whenever a query is executed. However, you may also explicitly prepare statements for reuse elsewhere with **`sqlx.DB.Prepare()`**:

```
stmt, err := db.Prepare(`SELECT * FROM place WHERE telcode=?`)
row = stmt.QueryRow(65)

tx, err := db.Begin()
txStmt, err := tx.Prepare(`SELECT * FROM place WHERE telcode=?`)
row = txStmt.QueryRow(852)
```

Prepare actually runs the preparation on the database, so it requires a connection and its connection state. database/sql abstracts this from you, allowing you to execute from a single Stmt object concurrently on many connections by creating the statements on new connections automatically. **`Preparex()`**, which returns an **`sqlx.Stmt`** which has all of the handle extensions that sqlx.DB and sqlx.Tx do:

```
stmt, err := db.Preparex(`SELECT * FROM place WHERE telcode=?`)
var p Place
err = stmt.Get(&p, 852)
```

The standard sql.Tx object also has a **`Stmt()`** method which returns a transaction-specific statement from a pre-existing one. sqlx.Tx has a **`Stmtx`** version which will create a new transaction specific **`sqlx.Stmt`** from an existing sql.Stmt *or* sqlx.Stmt.

Query Helpers

The **`database/sql`** package does not do anything with your actual query text. This makes it trivial to use backend-specific features in your code; you can write queries just as you would write them in your database prompt. While this is very flexible, it makes writing certain kinds of queries difficult.

"In" Queries

Because **`database/sql`** does not inspect your query and it passes your arguments directly to the driver, it makes dealing with queries with IN clauses difficult:

```
SELECT * FROM users WHERE level IN (?);
```

When this gets prepared as a statement on the backend, the bindvar **?** will only correspond to a *single* argument, but what is often desired is for that to be a variable number of arguments depending on the length of some slice, eg:

```
var levels = []int{4, 6, 7}
rows, err := db.Query("SELECT * FROM users WHERE level IN (?);", levels)
```

This pattern is possible by first processing the query with **sqlx.In**:

```
var levels = []int{4, 6, 7}
query, args, err := sqlx.In("SELECT * FROM users WHERE level IN (?);", levels)

// sqlx.In returns queries with the `?` bindvar, we can rebind it for our backend
query = db.Rebind(query)
rows, err := db.Query(query, args...)
```

What **sqlx.In** does is expand any bindvars in the query passed to it that correspond to a slice in the arguments to the length of that slice, and then append those slice elements to a new arglist. It does this with the **?** bindvar only; you can use **db.Rebind** to get a query suitable for your backend.

Named Queries

Named queries are common to many other database packages. They allow you to use a bindvar syntax which refers to the names of struct fields or map keys to bind variables a query, rather than having to refer to everything positionally. The struct field naming conventions follow that of **StructScan**, using the **NameMapper** and the **db** struct tag. There are two extra query verbs related to named queries:

- **NamedQuery(...) (*sqlx.Rows, error)** - like Queryx, but with named bindvars
- **NamedExec(...) (sql.Result, error)** - like Exec, but with named bindvars

And one extra handle type:

- **NamedStmt** - an sqlx.Stmt which can be prepared with named bindvars

```
// named query with a struct
p := Place{Country: "South Africa"}
rows, err := db.NamedQuery(`SELECT * FROM place WHERE country=:country`, p)

// named query with a map
m := map[string]interface{}{"city": "Johannesburg"}
result, err := db.NamedExec(`SELECT * FROM place WHERE city=:city`, m)
```

Named query execution and preparation works off both structs and maps. If you desire the full set of query verbs, prepare a named statement and use that instead:

```
p := Place{TelephoneCode: 50}
pp := []Place{}

// select all telcodes > 50
nstmt, err := db.PrepareNamed(`SELECT * FROM place WHERE telcode > :telcode`)
err = nstmt.Select(&pp, p)
```

Named query support is implemented by parsing the query for the **`:param`** syntax and replacing it with the bindvar supported by the underlying database, then performing the mapping at execution, so it is usable on any database that sqlx supports. You can also use **`sqlx.Named`**, which uses the **`?`** bindvar, and can be composed with **`sqlx.In`**:

```
arg := map[string]interface{}{
    "published": true,
    "authors": []{8, 19, 32, 44},
}
query, args, err := sqlx.Named("SELECT * FROM articles WHERE published=:published AN
query, args, err := sqlx.In(query, args...)
query = db.Rebind(query)
db.Query(query, args...)
```

Advanced Scanning

`StructScan` is deceptively sophisticated. It supports embedded structs, and assigns to fields using the same precedence rules that Go uses for embedded attribute and method access. A common use of this is sharing common parts of a table model among many tables, eg:

```
type AutoIncr struct {
    ID       uint64
    Created  time.Time
}

type Place struct {
    Address string
    AutoIncr
}

type Person struct {
    Name string
    AutoIncr
}
```

With the structs above, Person and Place will both be able to receive **`id`** and **`created`** columns from a StructScan, because they embed the **`AutoIncr`** struct which defines them. This feature can enable you to quickly create an ad-hoc table for joins. It works recursively as well; the following will have the Person's Name and its AutoIncr ID and Created fields accessible, both via the Go dot operator and via StructScan:

```
type Employee struct {
    BossID uint64
    EmployeeID uint64
    Person
}
```

Note that sqlx historically supported this feature for non-embedded structs, this ended up being confusing because users were using this feature to define relationships and embedding the same structs twice:

```
type Child struct {
    Father Person
    Mother Person
}
```

This causes some problems. In Go, it's legal to shadow descendent fields; if Employee from the embedded example defined a **Name**, it would take precedence over the Person's Name. But *ambiguous* selectors are illegal and cause a runtime error. If we wanted to create a quick JOIN type for Person and Place, where would we put the **id** column, which is defined in both via their embedded AutoIncr? Would there be an error?

Because of the way that sqlx builds the mapping of field name to field address, by the time you Scan into a struct, it no longer knows whether or not a name was encountered twice during its traversal of the struct tree. So unlike Go, StructScan will choose the "first" field encountered which has that name. Since Go struct fields are ordered from top to bottom, and sqlx does a breadth-first traversal in order to maintain precedence rules, it would happen in the shallowest, top-most definition. For example, in the type:

```
type PersonPlace struct {
    Person
    Place
}
```

A StructScan will set an **id** column result in **Person.AutoIncr.ID**, also accessible as **Person.ID**. To avoid confusion, it's suggested that you use **AS** to create column aliases in your SQL instead.

Scan Destination Safety

By default, StructScan will return an error if a column does not map to a field in the destination. This mimics the treatment for things like unused variables in Go, but does *not* match the way that standard library marshallers like **encoding/json** behave. Because SQL is generally executed in a more controlled fashion than parsing JSON, and these errors are generally coding errors, a decision was made to return errors by default.

Like unused variables, columns which you ignore are a waste of network and database resources, and detecting things like an incompatible mapping or a typo in a struct tag early can be difficult without the mapper letting you know something wasn't found.

Despite this, there are some cases where ignoring columns with no destination might be desired. For this, there is the **Unsafe** method on each Handle type which returns a new copy of that handle with this safety turned off:

```
var p Person
// err here is not nil because there are no field destinations for columns in `place
err = db.Get(&p, "SELECT * FROM person, place LIMIT 1;")

// this will NOT return an error, even though place columns have no destination
udb := db.Unsafe()
err = udb.Get(&p, "SELECT * FROM person, place LIMIT 1;")
```

Controlling Name Mapping

Struct fields used as targets for StructScans *must* be capitalized in order to be accessible by sqlx. Because of this, sqlx uses a *NameMapper* which applies **strings.ToLower** to field names to map them to columns in your rows result. This isn't always desirable, depending on your schema, so sqlx allows the mapping to be customized a number of ways.

The simplest of these ways is to set it for a db handle by using **`sqlx.DB.MapperFunc`**, which receives an argument of type **`func(string) string`**. If your library requires a particular mapper, and you don't want to poison the **`sqlx.DB`** you receive, you can create a copy for use in the library to ensure a particular default mapping:

```
// if our db schema uses ALLCAPS columns, we can use normal fields
db.MapperFunc(strings.ToUpper)

// suppose a library uses lowercase columns, we can create a copy
copy := sqlx.NewDb(db.DB, db.DriverName())
copy.MapperFunc(strings.ToLower)
```

Each **`sqlx.DB`** uses the **`sqlx/reflectx`** package's Mapper to achieve this mapping underneath, and exposes the active mapper as **`sqlx.DB.Mapper`**. You can further customize the mapping on a DB by setting it directly:

```
import "github.com/jmoiron/sqlx/reflectx"

// Create a new mapper which will use the struct field tag "json" instead of "db"
db.Mapper = reflectx.NewMapperFunc("json", strings.ToLower)
```

Alternate Scan Types

In addition to using Scan and StructScan, an sqlx Row or Rows can be used to automatically return a slice or a map of results:

```
rows, err := db.Queryx("SELECT * FROM place")
for rows.Next() {
    // cols is an []interface{} of all of the column results
    cols, err := rows.SliceScan()
}

rows, err := db.Queryx("SELECT * FROM place")
for rows.Next() {
    results := make(map[string]interface{})
    err = rows.MapScan(results)
}
```

SliceScan returns an **`[]interface{}`** of all columns, which can be useful in situations where you are executing queries on behalf of a third party and have no way of knowing what columns may be returned. MapScan behaves the same way, but maps the column names to interface{} values. An important caveat here is that the results returned by **`rows.Columns()`** does not include fully qualified names, such that **`SELECT a.id, b.id FROM a NATURAL JOIN b`** will result in a Columns result of **`[]string{"id", "id"}`**, clobbering one of the results in your map.

Custom Types

The examples above all used the built-in types to both scan and query with, but database/sql provides interfaces to allow you to use any custom types:

- **`sql.Scanner`** allows you to use custom types in a Scan()
- **`driver.Valuer`** allows you to use custom types in a Query/QueryRow/Exec

These are the standard interfaces, and using them will ensure portability to any library that might be providing services on top of database/sql. For a detailed look at how to use them, read this blog post or check out the sqlx/types package, which implements a few standard useful types.

The Connection Pool

Statement preparation and query execution require a connection, and the DB object will manage a pool of them so that it can be safely used for concurrent querying. There are two ways to control the size of the connection pool as of Go 1.2:

- **`DB.SetMaxIdleConns(n int)`**
- **`DB.SetMaxOpenConns(n int)`**

By default, the pool grows unbounded, and connections will be created whenever there isn't a free connection available in the pool. You can use **`DB.SetMaxOpenConns`** to set the maximum size of the pool. Connections that are not being used are marked idle and then closed if they aren't required. To avoid making and closing lots of connections, set the maximum idle size with **`DB.SetMaxIdleConns`** to a size that is sensible for your query loads.

It is easy to get into trouble by accidentally holding on to connections. To prevent this:

- Ensure you **`Scan()`** every Row object
- Ensure you either **`Close()`** or fully-iterate via **`Next()`** every Rows object
- Ensure every transaction returns its connection via **`Commit()`** or **`Rollback()`**

If you neglect to do one of these things, the connections they use may be held until garbage collection, and your db will end up creating far more connections at once in order to compensate for the ones its using. Note that **`Rows.Close()`** can be called multiple times safely, so do not fear calling it where it might not be necessary.