Database-related Libraries in Go

Go Estonia Meetup – Summer 2022

Fabiano Spiga

July 24, 2022

Table of Contents

ORM **O**bject **R**elational **M**apping

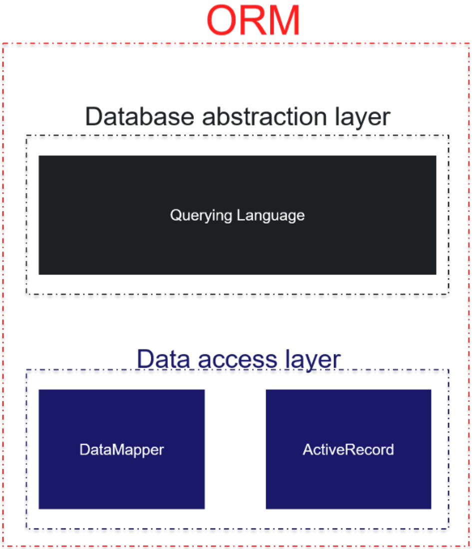
- *" (. . .) allows accessing relational databases in the form of abstract objects."*[1]
- is good for prototyping and/or CRUD operations, but it is *annoying* and *cumbersome* for long term maintenance due to the need for more complex queries
- Go offers SQL-related management packages - like sqlx or gorm - that still require pseudo-SQL *DSL*[2] *queries* and *manual mapping* via query text and struct tags that, if incorrect, will fail at runtime

[1] source: Wikipedia

[2] Domain Specific Language, i.e., to reinvent SQL in a set of Go function calls

Structure of an ORM

What's the abstract architecture of an ORM?



Gorm

The Go ORM Library: Main Features

Full-Featured ORM in Go:

- Gorm (website)
- Gorm V2 (repo)
- Gorm docs

```
go get -u gorm.io/gorm
go get -u gorm.io/driver/sqlite
```

Figure: Installing Go ORM + SQLite driver from CLI

- **Associations**: *HasOne*, *HasMany*, *BelongsTo*, *ManyToMany*, Polymorphism, Single-table inheritance
- **Hooks**: *Before/AfterCreate/Save/Update/Delete/Find*
- **Eager loading** with Preload, Joins
- **Transactions**: simple, Nested Transactions, Save Point, *RollbackTo* to Saved Point
- **Context**, Prepared Statement Mode, DryRun Mode
- **Batch** Insert, FindInBatches, Find To Map
- **SQL** Builder, Upsert, Locking, Optimizer/Index/Comment Hints, NamedArg, *Search/Update/Create* with SQL Expr
- **Composite Primary Key**
- **Auto Migrations**
- **Logger**
- **Extendable, flexible plugin API**: Database Resolver (Multiple Databases, Read/Write Splitting) / Prometheus
- **Tested**: Every feature comes with tests

GORM – Quickstart I

■ keyword ■ string ■ highlight ■ comment ■ code

```go
package main

import (
  "gorm.io/gorm"
  "gorm.io/driver/sqlite"
)

type Product struct {
  gorm.Model
  Code  string
  Price uint
}

func main() {
  db, err := gorm.Open(sqlite.Open("test.db"), &gorm.Config{})
  if err != nil {
    panic("failed to connect database")
  }

  // Migrate the schema
  db.AutoMigrate(&Product{})

  // Create
  db.Create(&Product{Code: "D42", Price: 100})
```

GORM – Quickstart II

keyword string highlight comment code

```go
// Read
var product Product
db.First(&product, 1) // find product with integer primary key
db.First(&product, "code = ?", "D42") // find product with code D42

// Update - update product's price to 200
db.Model(&product).Update("Price", 200)

// Update - update multiple fields
db.Model(&product).Updates(Product{Price: 200, Code: "F42"}) // non-zero
    fields
db.Model(&product).Updates(map[string]interface{}{"Price": 200, "Code": "F42"
    })

// Delete - delete product
db.Delete(&product, 1)
```

Squirrel

a fluent SQL Generator for Go

- Squirrel (repo)
- avoids some SQL query writing issues
- does not explicitly support tuples, but you can get some turnaround solutions

Squirrel – Quickstart I

■ keyword ■ string ■ highlight ■ comment ■ code

```
// ex.1 - building SQL queries from composable parts:

import sq "github.com/Masterminds/squirrel"

users := sq.Select("*").From("users").Join("emails USING (email_id)")

active := users.Where(sq.Eq{"deleted_at": nil})

sql, args, err := active.ToSql()

sql == "SELECT * FROM users JOIN emails USING (email_id) WHERE deleted_at IS
     NULL"

// ex.2 - building SQL queries from composable parts:

sql, args, err := sq.
    Insert("users").Columns("name", "age").
    Values("moe", 13).Values("larry",sq.Expr("? + 5", 12)).
    ToSql()

sql == "INSERT INTO users (name,age) VALUES (?,?),(?,? + 5)"
```

Squirrel – Quickstart II

■ keyword ■ string ■ highlight ■ comment ■ code

```
// ex.3 execute queries directly:
stooges := users.Where(sq.Eq{"username": []string{"moe", "larry", "curly", "
      shemp"}})
three_stooges := stooges.Limit(3)
rows, err := three_stooges.RunWith(db).Query()

// Behaves like:
rows, err := db.Query("SELECT * FROM users WHERE username IN (?,?,?,?) LIMIT 3",
                      "moe", "larry", "curly", "shemp")


// ex.4 conditional query building:
if len(q) > 0 {
    users = users.Where("name LIKE ?", fmt.Sprint("\%", q, "\%"))
}

// ex.5 caching and statements

// StmtCache caches Prepared Stmts for you
dbCache := sq.NewStmtCache(db)

// StatementBuilder keeps your syntax neat
mydb := sq.StatementBuilder.RunWith(dbCache)
select_users := mydb.Select("*").From("users")
```

Squirrel – Quickstart III

■ keyword ■ string ■ highlight ■ comment ■ code

```
// ex.6 PostgreSQL interactions:

psql := sq.StatementBuilder.PlaceholderFormat(sq.Dollar)

// You use question marks for placeholders...
sql, _, _ := psql.Select("*").From("elephants").Where("name IN (?,?)", "Dumbo",
     "Verna").ToSql()

/// ...squirrel replaces them using PlaceholderFormat.
sql == "SELECT * FROM elephants WHERE name IN (\$1,\$2)"


/// You can retrieve an id ...
query := sq.Insert("nodes").
    Columns("uuid", "type", "data").
    Values(node.Uuid, node.Type, node.Data).
    Suffix("RETURNING \"id\"").
    RunWith(m.db).
    PlaceholderFormat(sq.Dollar)

query.QueryRow().Scan(&node.id)

// ex.7 Escape question marks: ?? (2) of them generate a $ Dollar placeholder:

SELECT * FROM nodes WHERE meta->'format' ??| array [?, ?]  ⟹
SELECT * FROM nodes WHERE meta->'format'  ?| array [$1,$2]
```

Structable

A Golang Struct-to-Table Database Mapper

- **Structable** (repo)
- maps a struct(record) to a database *table* via a structable.Recorder
- intended to be used as a back-end tool for building systems like Active Record mappers
- works by mapping a struct to columns in a database
- satisfies a CRUD-centered record management system, through a standard contract

```
$ glide get github.com/Masterminds/structable
$ # or...
$ go get github.com/Masterminds/structable
```

Figure: Install Structable via CLI

Structable – Quickstart I

■ keyword ■ string ■ highlight ■ comment ■ code

```
// Structable standard contract to be filled:

  type Recorder interface {
    Bind(string, Record) Recorder // Link struct to table
    Interface() interface{}  // Get the struct that has been linked
    Insert() error // INSERT just one record
    Update() error // UPDATE just one record
    Delete() error // DELETE just one record
    Exists() (bool, error) // Check for just one record
    ExistsWhere(cond interface{}, args ...interface{}) (bool, error)
    Load() error  // SELECT just one record
    LoadWhere(cond interface{}, args ...interface{}) error // Alternate Load()
  }
```

- DBX (repo) (by Storj)
- generates database schemata and code for DB manipulation
- takes a description of models and operations to perform on them, and can generate code to interact with SQL databases
- generates code for all of the models and fields and uses the postgres SQL dialect, i.e., PL/pgSQL Procedural Language
- nevertheless, DBX is designed to be agnostic to many different database engines, i.e., PostgreSQL and SQLite3.
- grammar: a DBX file has two constructs: tuples and lists. A list contains comma separated tuples, and tuples contain white space separated strings or more lists

DBX - Quickstart I

keyword string highlight comment code

```
// declaring a simple model, place it in a .dbx file

model user (
    key    pk
    unique id
    unique name

    field pk         serial64
    field created_at timestamp ( autoinsert )
    field updated_at timestamp ( autoinsert, autoupdate )
    field id         text
    field name       text
)

// build source via Command Line
```

```
$ dbx golang example.dbx .
```

The former produces an example.go file within the same directory

DBX - Quickstart II


```
// generating a schema via CLI
$ dbx schema example.dbx .
```

produces a file example.dbx.postgres.sql with SQL statements to create the tables for the models, using PL/psSQL dialect

```
// creating schemata for both PostgreSQL and SQLite3:
$ dbx schema -d postgres -d sqlite3  example.dbx .
$ dbx golang -d postgres -d sqlite3  example.dbx .
```

The former produces the appropriate schema according to the specific SQL dialect specified by the CLI flag -d; - -dialect.

These commands are intended to normally be used with //go:generate directives

DBX - Quickstart III

The Methods interface is shared between the Tx and DB interfaces and will contain methods to interact with the database when they get generated:

```
// instantiating a struct definition:

type User struct {
    Pk          int64
    CreatedAt time.Time
    UpdatedAt time.Time
    Id          string
    Name        string
}

// instantiating concrete types DB and Tx, and their related interfaces:

type Methods interface {
}

type TxMethods interface {
    Methods

    Commit() error
    Rollback() error
}

type DBMethods interface {
    Schema() string
    Methods
}
```

DBX - Quickstart IV

■ keyword ■ string ■ highlight ■ comment ■ code

There are four kinds of operations: create, read, update and delete.

```
// add one of each operation for the user model based on the primary key:
create user ( )
update user ( where user.pk = ? )
delete user ( where user.pk = ? )
read one (
    select user
    where  user.pk = ?
)
// regenerate the Go code to expand the database interface:
type Methods interface {
    Create_User(ctx context.Context,
        user_id User_Id_Field,
        user_name User_Name_Field) (
        user *User, err error)

    Delete_User_By_Pk(ctx context.Context,
        user_pk User_Pk_Field) (
        deleted bool, err error)

    Get_User_By_Pk(ctx context.Context,
        user_pk User_Pk_Field) (
        user *User, err error)

    Update_User_By_Pk(ctx context.Context,
        user_pk User_Pk_Field,
        update User_Update_Fields) (
        user *User, err error)
}
```

DBX - Quickstart V

■ keyword ■ string ■ highlight ■ comment ■ code

DBX exposes transaction handling, but it can be *verbose* in *Commits* and *Rollbacks*. If so, define a package as a *collection of multiple files*,e.g., a helper method in another file to the $*\mathrm{DB}$ type.

```
// createUser can be succintely written as:
func createUser(ctx context.Context, db *DB) (user *User, err error) {
    err = db.WithTx(func(ctx context.Context, tx *Tx) error) {
        user, err = tx.Create_User(ctx,
            User_Id("some unique id i just generated"),
            User_Name("Donny B. Xavier"))
        return err
    })
    return user, err
}
// function as an added 'helper method' in a separate file
func (db *DB) WithTx(ctx context.Context, fn func(context.Context, *Tx) error)
(err error) {
    tx, err := db.Open()
    if err != nil {
        return err
    }
    defer func() {
        if err == nil {
            err = tx.Commit()
        } else {
            tx.Rollback() // log this perhaps?
        }
    }()
    return fn(ctx, tx)
}
```

pgx

PostgreSQL and CockroachDB Driver and Toolkit

- pgx (repo)
- perform native, direct SQL queries – but it is easy to make typos
- enable PostgreSQL-specific features that the standard *database/sql* package does not allow for
- excellent flexibility and performance comparing to *database/sql*, i.e.:
    1. PostgreSQL specific types - types such as *arrays* can be parsed much quicker because pgx uses the *binary format*
    2. Automatic statement preparation and caching - pgx will prepare and cache statements by default. This can provide an significant free improvement to code that does not explicitly use prepared statements. Under certain workloads, it can perform nearly 3x the number of queries per second.
    3. Batched queries - multiple queries can be batched together to minimize network round trips
- toolkit component: underlying packages can be used to implement alternative drivers, proxies, load balancers, logical replication clients, etc.

pgx

List of pgx family libraries – pt.I

pgconn
: pgconn is a lower-level PostgreSQL database driver that operates at nearly the same level as the C library libpq.

pgxpool
: pgpool is a connection pool for pgx, which is entirely decoupled from its default pool implementation. This means that pgx can be used with a different pool or without any pool at all.

stdlib
: stdlib is a database/sql compatibility layer for pgx, which can be used as a normal database/sql driver. But - at any time - the native interface can be acquired for more performance or PostgreSQL specific functionality.

pgtype
: pgtype over 70 PostgreSQL types are supported including uuid, hstore, json, bytea, numeric, interval, inet, and arrays. These types support database/sql interfaces and are usable outside of pgx. They are fully tested in pgx and pq. They also support a higher performance interface when used with the pgx driver.

pgproto3
: pgproto3 provides standalone encoding and decoding of the PostgreSQL v3 wire protocol. This is useful for implementing very low level PostgreSQL tooling.

List of pgx family libraries – pt.II

pglogrepl: pglogrepl provides functionality to act as a client for PostgreSQL logical replication.

pgmock: pgmoc offers the ability to create a server that mocks the PostgreSQL wire protocol. This is used internally to test pgx by purposely inducing unusual errors. pgproto3 and pgmock together provide most of the foundational tooling required to implement a PostgreSQL proxy or MitM (such as for a custom connection pooler).

tern: tern is a stand-alone SQL migration system.

pgerrcode: pgerrcode contains constants for the PostgreSQL error codes.

- 3rd Party Libraries with PGX Support:

scany: scany Library for scanning data from a database into Go structs and more.

gopgkrb5: gopgkrb5 adds GSSAPI / Kerberos authentication support.

google-uuid: pgx-google-uuid adds support for github.com/google/uuid.

pgx - Quickstart

■ keyword ■ string ■ highlight ■ comment ■ code

```go
package main

import (
    "context"
    "fmt"
    "os"
    "github.com/jackc/pgx/v4"
)

func main() {
    // urlExample := "postgres://username:password@localhost:5432/database_name"
    conn, err := pgx.Connect(context.Background(), os.Getenv("DATABASE_URL"))
    if err != nil {
        fmt.Fprintf(os.Stderr, "Unable to connect to database: %v\n", err)
        os.Exit(1)
    }
    defer conn.Close(context.Background())

    var name string
    var weight int64
    err = conn.QueryRow(context.Background(), "select name, weight from widgets
     where id=$1", 42).Scan(&name, &weight)
    if err != nil {
        fmt.Fprintf(os.Stderr, "QueryRow failed: %v\n", err)
        os.Exit(1)
    }

    fmt.Println(name, weight)
}
```

- sqlc (home) and sqlc (repo)
  1. write vanilla SQL code
  2. run sqlc to generate code with type-safe interfaces to those queries
  3. write application code that calls the generated code
- produces type-safe code from the SQL source of truth itself, maintains runtime performance, but introduces code generation and extra tooling
- input/output SQL mappings from the standard database/sql package are straightforward, but they do not scale, and it's trivial to make mistakes on mappings that are not caught until runtime
- avoid using struct tags, hand-written mapper functions, unnecessary reflection, unsafe empty interfaces or add any new dependencies to your code

sqlc - Quickstart I

■ keyword ■ string ■ highlight ■ comment ■ code

```
// example of sqlc config file:

{
    "version": "1",
    "packages": [{
        "schema": "schema.sql",
        "queries": "query.sql",
        "name": "main",
        "path": "."
   }]
}


// compiling SQL into fully type-safe, idiomatic Go code:

CREATE TABLE authors (
    id   BIGSERIAL PRIMARY KEY,
    name text      NOT NULL,
    bio  text
);

-- name: GetAuthor :one
SELECT * FROM authors
WHERE id = $1 LIMIT 1;
```

sqlc - Quickstart II

keyword string highlight comment code

```go
// automatically generates the following code:
package db

import (
	"context"
	"database/sql"
)

type Author struct {
	ID   int64
	Name string
	Bio  sql.NullString
}


const getAuthor = `-- name: GetAuthor :one
SELECT id, name, bio FROM authors
WHERE id = $1 LIMIT 1
`
type Queries struct {
	db *sql.DB
}

func (q *Queries) GetAuthor(ctx context.Context, id int64) (Author, error) {
	row := q.db.QueryRowContext(ctx, getAuthor, id)
	var i Author
	err := row.Scan(&i.ID, &i.Name, &i.Bio)
	return i, err
}
```

dbx

Enable Full Table Data Key/Value Caching

- dbx (repo)
- supports read KV-cached full table data for very efficient caching, e.g., Sqlite3 queried directly at 3w+/s, or at 350 w+/s after opening the cache (it has no network I/O), i.e., much faster than Redis
- supports nesting of structures, e.g., multi-layer nested JSON
- avoids the verbosity of code and data inconsistencies of running DB with operating cache
- supports reflections to implement more complex functions, with unlimited layers, retaining good performance
- supports MySQL/Sqlite3 and Memcached databases

```
go get "github.com/go-sql-driver/mysql"
go get "github.com/mattn/go-sqlite3"
```

Figure: Install dbx via CLI

dbx – Quickstart I

■ keyword ■ string ■ highlight ■ comment ■ code

```
// ex.1 enabling the Key/Value cache

db.BindStruct("user", &User{}, true)
db.BindStruct("group", &Group{}, true)
db.EnableCache(true)

// ex.2 support nesting, avoid inefficient reflection

type Human struct {
Age int64      `db:"age"`
    }

type User struct {
    Human
    Uid        int64     `db:"uid"`
    Gid        int64     `db:"gid"`
    Name       string    `db:"name"`
    CreateDate time.Time `db:"createDate"`
}
```

dbx – Quickstart II

keyword string highlight comment code

```
// dry syntax, close to that of any scripting language:

// open database
db, err = dbx.Open("mysql", "root@tcp(localhost)/test?parseTime=true\&
 charset=utf8")

// insert one
db.Table("user").Insert(u1)

// find one
db.Table("user").WherePK(1).One(u2)

// update one
db.Table("user").Update(u2)

// delete one
db.Table("user").WherePK(1).Delete()

// find multi
db.Table("user").Where("uid>?", 1).All(&userList)
```