Go: Working with Databases

A database first development guide

Tit Petric

Step by step guide for
working with databases in Go.

Go With Databases

Tit Petric

This book is for sale at http://leanpub.com/go-with-databases

This version was published on 2020-12-28

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Also By Tit Petric

API foundations in Go

12 Factor Applications with Docker and Go

Advent of Go Microservices

Go With Graphics

Contents

Introduction

About me

I'm passionate about API development, good practices, performance optimizations and educating people to strive for quality. I have about two decades of experience writing and optimizing software, and often solve real problems in various programming languages, Go being my favorite.

You might know some of my work from:

- Author of API Foundations in Go[1],
- Author of 12 Factor Applications with Docker and Go[2],
- Author of Advent of Go Microservices[3],
- Blog author on scene-si.org[4]

Professionally I specialize in writing APIs in the social/media industry and for various content management products. Due to the public exposure of such APIs, it's performance characteristics are of paramount importance. I'm a solid MySQL DBA with experience on other databases as well.

Who is this book for?

This book is for people who want to familiarize themselves with working with databases from Go. We will cover connecting to databases, issuing queries, transactions and other common usage patterns.

The aim of the book is to provide SQL database-specific examples, demonstrate best practices and common patterns when connecting and querying the database. We will also look at other NoSQL databases.

[1] https://leanpub.com/api-foundations
[2] https://leanpub.com/12fa-docker-golang
[3] https://leanpub.com/go-microservices
[4] https://scene-si.org

In the book, I will cover these subjects:

1. Introduction
2. Standard library
3. Connecting to our databases
4. go mod init example.com/go-with-databases
5. go run main.go
6. Context and database drivers
7. Querying the database (standard library)
8. Querying the database (jmoiron/sqlx)
9. go get github.com/jmoiron/sqlx@master
10. The difference between Select and Get
11. Special database column types/values
12. Query placeholders
13. Setting up integration tests
14. Connecting to the database
15. Querying the database
16. Transactions
17. Modelling database schema
18. SQL database compatibility
19. ElasticSearch
20. Redis
21. MongoDB
22. Dgraph

How should I study it?

Just go through the book from the start to finish. If possible, try to do the exercises yourself not by copy pasting but by actually writing the code and snippets in the book, tailored to how you would lay out your project.

The work in individual chapters builds on what was done in the previous chapter. The examples are part of a step by step, chapter by chapter process.

Be sure to follow the Requirements section as you're working with the book.

Requirements

This is a book which gives you hands on instruction on working with various databases. We will be using a modern stack of software that is requires to complete all the exercises.

Linux and Docker

The book relies extensively on docker to provide us both the ability to run integration tests, as well as to run the various databases which we will be working with.

- Go 1.14+ (latest stable)
- Docker (a recent version),
- Drone CI,
- Various shell utilities and programs (awk, bash, sed, find, ls, make,...)

Please refer to the official docker installation instructions[5] on how to install a recent docker version, or install it from your package manager.

Own hardware

The recommended configuration if you have your own hardware is:

- 2 CPU core, x86/amd64, 64bit,
- 2GB ram,
- 128GB disk (SSD)

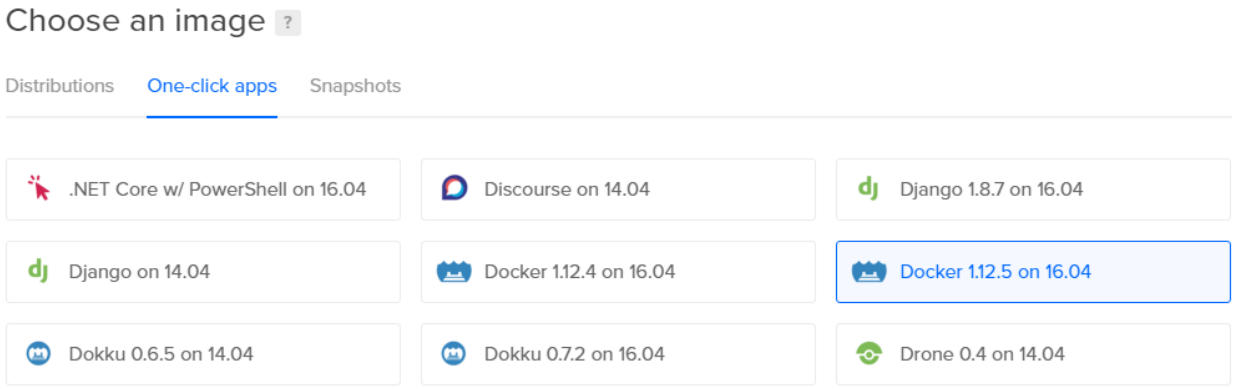
The minimal configuration known to mostly work is about half that, but you might find yourself in a tight place as soon as your usage goes up. If you're just tying out docker, a simple virtual machine might be good enough for you, if you're not running Linux on your laptop already.

[5] https://docs.docker.com/engine/installation/linux/

Cloud quick-start

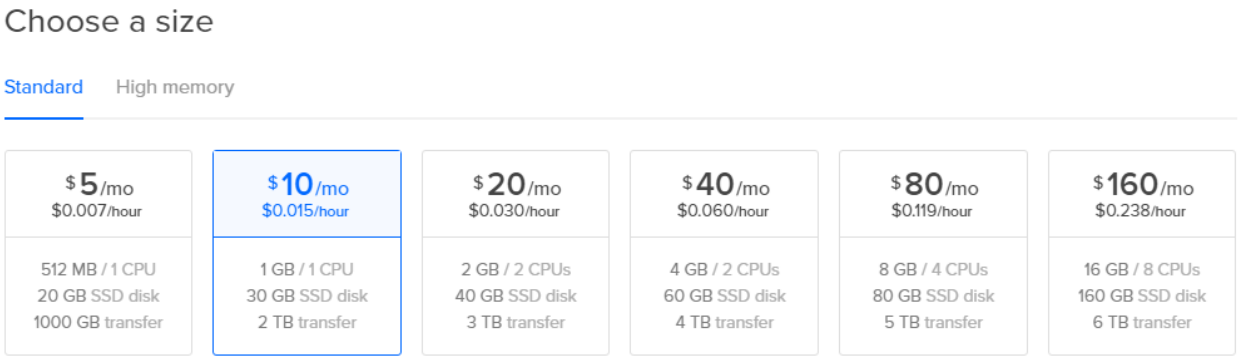
If having your own hardware is a bit of a buzzkill, welcome to the world of the cloud. You can literally set up your own virtual server on Digital Ocean within minutes. You can use this DigitalOcean referral link[6] to get a $10 credit, while also helping me take some zeros of my hosting bills.

After signing up, creating a Linux instance with a running Docker engine is simple, and only takes a few clicks. There's this nice green button on the top header of the page, where it says "Create Droplet". Click it, and on the page it opens, navigate to "One-click apps" where you choose a "Docker" from the list.



Choose Docker from "One-click apps"

Running docker can be disk-usage intensive. Some docker images may "weigh" up to or more than 1 GB. I would definitely advise choosing an instance with *at least* 30GB of disk space, which is a bargain for $10 a month, but you will have to keep an eye out for disk usage. It's been known to fill up.



Choose a reasonable disk size

Aside for some additional options on the page, like chosing a region where your droplet will be running in, there's only a big green "Create" button on the bottom of the page, which will set up everything you need.

[6] https://m.do.co/c/021b61109d56

Introduction

In this book, we'll take a look at how to work with SQL databases. We will start with the standard library extend this knowledge with sqlx. The book aims to give you the knowledge in a database-agnostic way, but the main examples will be tailored towards MySQL and Postgres.

Standard library

When dealing with SQL databases from go, the database/sql[7] will be our starting point. The package provides a rudimentary API for connecting to and working with SQL databases.

To actually connect to a database, we will need to import a "driver", a package that implements the communication protocol for a given database. There are a range of packages listed on the SQL Driver Wiki[8], but we will use the following few:

- SQLite: `modernc.org/sqlite`
- Postgres: `github.com/lib/pq`
- MySQL: `github.com/go-sql-driver/mysql`

All the drivers I've chosen here are "Pure Go" drivers, meaning you can build your application with `CGO_ENABLED=0` in order to produce static, portable binaries.

You can use multiple drivers at the same time.

[7] https://godoc.org/database/sql
[8] https://github.com/golang/go/wiki/SQLDrivers

Connecting to our databases

First, create a go.mod package for your project:

```
# go mod init example.com/go-with-databases
```

This is a required step, so we can rely on `go run` and `go build` to pull any required packages we use. In our `main.go` file, let's import all the SQL drivers and the `database/sql` package.

```
package main

import (
        "context"
        "database/sql"

        _ "github.com/go-sql-driver/mysql"
        _ "github.com/lib/pq"
        _ "modernc.org/sqlite"

        "github.com/apex/log"
)
```

We are importing the drivers using _ before the package name, so the import itself just runs the `init()` function from the driver, which registers it to make it available for `sql.Open`.

We will test the SQLite database first, as it doesn't require a running database service. We can open up the SQLite `:memory:` database, which will not persist between runs.

```
func start(ctx context.Context) error {
        // create db client
        db, err := sql.Open("sqlite", ":memory:")
        if err != nil {
                return err
        }

        // open a db connection
        _, err = db.Conn(ctx)
        if err != nil {
                return err
```

```
12          }
13
14          return nil
15  }
```

And finally, our `main()` function where we create a global context, and invoke our `start` function that uses the database.

```
1   func main() {
2           ctx := context.Background()
3           if err := start(ctx); err != nil {
4                   log.WithError(err).Fatal("failed")
5           }
6           log.Info("success")
7   }
```

Finally, we can run our connection test:

```
1   # go run main.go
2   2020/12/25 13:01:23  info success
```

Context and database drivers

When dealing with databases, or with Go in general, it's extremely important that we use context.Context; The reason isn't only for cancellation, but also for application performance monitoring.

Application performance monitoring uses the context to create and propagate a transaction, for example, a REST API HTTP request. Each database query we will execute will be registered as a "span", so you can see which queries have been issued in your web service API request, and how long that query took, along with other diagnostics and performance metrics. The context value is used to bring this data together.

With Elastic APM[9] there are a number of database drivers provided. The database drivers are implemented by wrapping the original database driver and adding the APM instrumentation code around each query.

The updated list of drivers to use with APM tracing:

- go.elastic.co/apm/module/apmsql/mysql
- go.elastic.co/apm/module/apmsql/pq

Elastic APM doesn't provide the same SQLite driver, but we can wrap the driver ourselves. Create a `db/sqlite` folder with `sqlite.go`:

```
1  package sqlite
2
3  import (
4    "strings"
5
6    "go.elastic.co/apm/module/apmsql"
7    "modernc.org/sqlite"
8  )
9
10 func init() {
11   apmsql.Register("sqlite", &sqlite.Driver{}, apmsql.WithDSNParser(ParseDSN))
12 }
13
14 // ParseDSN parses the sqlite datasource name.
15 func ParseDSN(name string) apmsql.DSNInfo {
16   if pos := strings.IndexRune(name, '?'); pos >= 0 {
```

[9] https://www.elastic.co/apm

```
    name = name[:pos]
  }
  return apmsql.DSNInfo{
    Database: name,
  }
}
```

This is the same basic implementation as the wrappers for mysql and postgres drivers. Since we already started by implementing some of our project structures, let's create `db/mysql` containing `mysql.go`:

```
package mysql

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

And `db/pq` with `pq.go`:

```
package pq

import (
  _ "github.com/lib/pq"
)
```

We can now update the main imports to use our locally defined drivers:

```
  _ "example.com/go-with-databases/db/mysql"
  _ "example.com/go-with-databases/db/pq"
  _ "example.com/go-with-databases/db/sqlite"
```

Querying the database (standard library)

The functions for querying the database can be split into three distinct functions:

1. `ExecContext` - for running queries that insert or modify data or schema,
2. `QueryContext` - for SELECT queries returning multiple rows,
3. `QueryRowContext` for SELECT queries returning 1 row at most,

You can use them from either an `sql.DB` or an `sql.Conn` (DB.Conn() returns `sql.Conn`). The difference between Query and QueryRow is that the destination is either a slice of rows, or a single row struct.

Exec

Using `ExecContext` to create a `testing` table if it doesn't exist:

```
1  // create a table
2  if _, err := conn.ExecContext(ctx, "create table if not exists testing ( id int PRIM\
3  ARY KEY )"); err != nil {
4    return err
5  }
```

The `sql.Result` returned by Exec/ExecContext satisfies the following functions:

- LastInsertId() (int64, error)
- RowsAffected() (int64, error)

The result of these functions varies across databases, but it generally tells you how many rows have been affected by the SQL query, and what the last inserted ID was. The last inserted ID is a reference to typical database sequence or auto_increment columns, where the actual value is generated by the database at INSERT time.

In practice it's rare that these functions are used - the table primary keys might be an `uint64` type, or often some form of a string (UUID), and for what it's worth, the affected rows information also isn't very useful in day-to-day work with SQL databases.

Query

Let's produce a list of tables in our database. For SQLite, that query is as follows:

```
rows, err := conn.QueryContext(ctx, "select name from sqlite_master where type='tabl\
e' order by name")
if err != nil {
  return err
}
```

The QueryContext function returns an `sql.Rows` which we can iterate over with the function `Next`, scan individual rows with `Scan`, and finally check if any error was encountered from iterating over the rows:

```
  var dbNames []string
  for rows.Next() {
    var dbName string
    if err := rows.Scan(&dbName); err != nil {
      return err
    }
    dbNames = append(dbNames, dbName)
  }
  // check errors from iterating over rows
  if err := rows.Err(); err != nil {
    return err
  }
```

This is also where the difficulty of using the standard library database/sql package becomes really apparent. Even with this simple example of reading very primitive database structures, we:

- don't have buffered reads providing result counts, rows is a database cursor,
- scanning needs to be aware of column count (typical query for SQL is `select * from table ...`),
- we don't have scanning to complex types like map, slice or struct to scan the whole row

It's because of this reason, that people usually resort to jmoiron/sqlx[10] that adds on general purpose extensions over the database/sql API. From here on out, we will use this package to access and query our databases.

[10] https://github.com/jmoiron/sqlx

Querying the database (jmoiron/sqlx)

Using sqlx means we have a "drop-in" replacement for the `database/sql` import, with an extended API that is not available in the standard library. We can choose to alias the import to `sql`, or, preferrably, let's rename `sql.Open` to `sqlx.Open` and the rest of the code will continue to function without any required changes.

Of couse, sqlx adds new APIs that should be used as a replacement to `Query/QueryRow` functions:

- GetContext(ctx context.Context, dest interface{}, query string, args ...interface{}) error
- SelectContext(ctx context.Context, dest interface{}, query string, args ...interface{}) error

The original Query functions have actually been extended to `QueryxContext` and `QueryRowxContext`. These now return `sqlx.Rows/sqlx.Row`, which have three utility functions: `MapScan`, `StructScan` and `SliceScan`. These improve on the database/sql functionality where just a simple `Scan()` was provided. They allow reading rows from the database and filling out your own provided structures, maps or slices.

I'd suggest you forget that `Query*` functions exist. By default you should opt into using `SelectContext`, or `GetContext` where you require a returned row. Particularly using the cursor-backed Query functions should be encouraged only when a large dataset it read from the database, and processed row by row (e.g. map/reduce jobs). This is because it isn't practical to buffer the results in memory for such workloads.

Like Query -> Queryx "rename", we must also use `Connx(ctx)` instead of `Conn()` so we can use the Get and Select functions from the `sqlx.Conn` object. Rename `db.Conn(` to `db.Connx(` and you're good to continue.

Connx() requires updating jmoiron/sqlx to a newer version than is available at the time of writing. In your project (where go.mod lives), issue the following command to update it:

```
1  # go get github.com/jmoiron/sqlx@master
```

Our code for listing the databases becomes:

```
var dbNames []string
if err := conn.SelectContext(ctx, &dbNames, "select name from sqlite_master where ty\
pe='table' order by name"); err != nil {
  return err
}
```

Even this simple example is already significantly shorter than the standard library usage example. The database results are scanned into a slice of strings, but working with structs is similarly simple. Like `encoding/json`, we can rely on field tags to specify which column will be scanned.

```
type table struct {
        Name string `db:"name"`
}
var dbNamesStruct []table
if err := conn.SelectContext(ctx, &dbNamesStruct, "select name from sqlite_master wh\
ere type='table' order by name"); err != nil {
        return err
}
```

The difference between Select and Get

The Select APIs are there to fetch multiple rows, while the Get API is there to fetch a single row. Similarly, Query is there to fetch multiple rows, and QueryRow is there to fetch a single row. The Get() and QueryRow() APIs return a particular error, sql.ErrNoRows[11] if no rows are returned.

```
var ErrNoRows = errors.New("sql: no rows in result set")
```

If you really think about it - in every case where you would use Get(), you'd need to remap this error into one of your own, which you can document and handle in your application, especially if you're writing a HTTP API service which basically just returns whatever error occured. Or if you handle empty values, you need to swallow it with a condition similar to this one:

```
if err := svc.db.GetContext(ctx, &group, query, groupID); err != nil && !errors.Is(e\
rr, sql.ErrNoRows) {
```

Perhaps you should have been using Select in the first place if all you're doing is just throwing away ErrNoRows. This is one of such snippets from a piece of production code:

```
if err := svc.db.SelectContext(ctx, &result, query, groupID); err != nil {
  return nil, err
}
if len(result) == 0 {
  return nil, errors.New("no such group")
}
return result[0], nil
```

The example is readable, nil-pointer safe, and has the ability to return your own error type to ease debugging. When fetching multiple rows, you can have one liner utility functions like this one:

[11] https://godoc.org/database/sql#pkg-variables

```
queryMessageIDs := func(query string, args []interface{}) ([]int64, error) {
  result := []int64{}
  return result, svc.db.SelectContext(ctx, &result, query, args...)
}
```

With GetContext we would leak the ErrNoRows error value. If a single API call is composed using multiple similar queries, you'll have a hard time knowing where the error came from.

There are additional ways to approach this problem:

- you could resort to github.com/pkg/errors[12] Wrap() function to add more context for such errors,
- you could use github.com/hashicorp/go-multierror[13] to append your own error value and have the ability to use errors.Is[14] on both sql.ErrNoRows and your own sentinel error.

Of course, just being aware that this particular error case must be handled, maybe you'll listen to my advice, and just avoid the scenario where the error is expected to occur, instead of buying into it and then masking it out. Or to put it differently - if you tolerate empty rows, don't fix this issue by clearing the expected error, fix this issue by creating the error on an unexpected value (zero rows).

[12] https://github.com/pkg/errors
[13] https://github.com/hashicorp/go-multierror
[14] https://golang.org/pkg/errors/#Is

Special database column types/values

Depending on the go field definition, or the SQL database column type, there can be some complex differences between one or the other.

The standard library already provides several such types, to handle a database column definition where a field may contain a NULL value:

- NullBool
- NullFloat64
- NullInt32
- NullInt64
- NullString
- NullTime

If you need to differentiate between a NULL and empty value, using these types is a more practical and less error-prone way how to approach the problem. The `jmoiron/sqlx` package provides a few additional types:

- BitBool
- GzippedText
- JSONText
- NullJSONText

These custom types are examples of how a complex encoding can be used in order to provide a value for a database column. You should resort to these whenever needed.

Both MySQL and Postgres have support for complex JSON types. Personally I think that they are taking things a bit too far, because storing JSON as column values is usually an indicator of bad practices, but I understand that modifying database schema is prohibitive in many cases, and a lot of older, still functioning databases, don't have support for JSON column types. Using the JSONText types is a valid approach to JSON storage in a relational database.