INTRODUCCIÓN A LA CLASE

STORAGE IMPLEMENTATION

Objetivos de esta clase

- Conocer el paquete database/sql de go.
- Implementar el paquete database/sql para generar un repository
- Aprender a implementar la capa repository
- Implementar los métodos tradicionales de un repository

PACKAGE DATABASE/SQL

STORAGE IMPLEMENTATION

// Package Database/Sql

“Es una implementación nativa de Go, que expone una interfaz que permite gestionar la conexión y datos de una base de datos.”

Características

- Forma parte de las librerías standard de Go.
- Compatible con Bases de Datos SQL.
- Debe ser usado conjuntamente con un driver de base de datos.
- Hace uso del type **sql.DB** para gestionar la conexión y la ejecución.
- Puede generar una conexión o administrar un pool de conexiones.

Instalación

Para usar sql/database necesitamos instalar el driver correspondiente a la DB con la que vayamos a conectar. En este caso vamos a hacer uso de MySql

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

Ahora podemos crear un file llamado db.go en el que debemos importar el package database/sql y nuestro driver

```
{}
import (
    "database/sql"

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

Implementación

Ahora se puede ejecutar la conexión a nuestra DB. Se invoca la función Open del package sql la cual recibe por parámetro el nombre del driver (por eso debemos importar el driver), y los datos

{}

```
dataSource := "user:pass@tcp(server:Port)/dbName"
// Open inicia un pool de conexiones. Sólo abrir una vez
storageDB, err = sql.Open("mysql", dataSource)
if err != nil {
    panic(err)
}
```

Implementación

Es una buena práctica dejar en un archivo la creación de la conexión. Podría quedar de la siguiente manera. En la función de inicialización se establece la conexión y se exporta la variable StorageDB como puntero del type *sql.DB creado con los datos de nuestra conexión. De esta manera cuando necesitemos operar la DB, solo tenemos que llamar a la variable StorageDB.

{}

```go
package db
import (
    "database/sql"
    "log"

    _ "github.com/go-sql-driver/mysql"
)
var (
    StorageDB *sql.DB
)
func init() {
    dataSource := "root:abcd1234@tcp(localhost:3306)/storage"
    // Open inicia un pool de conexiones. Sólo abrir una vez
    var err error
    StorageDB, err = sql.Open("mysql", dataSource)
    if err != nil {
        panic(err)
    }
    if err = StorageDB.Ping(); err != nil {
        panic(err)
    }
    log.Println("database Configured")
}
```

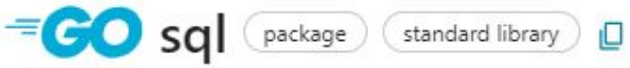
Consideraciones Generales

- El package se encarga de gestionar el pool de conexiones. No reutilizar el método Open, esto generará leaks de memoria y saturación de conexiones a la base.

- El dataSource es la configuración de la conexión. Se define de la siguiente forma:

- La documentación oficial ofrece detalles de la forma correcta de implementación y todas sus funciones. Puede ser consultada en el siguiente enlace:

https://pkg.go.dev/database/sql



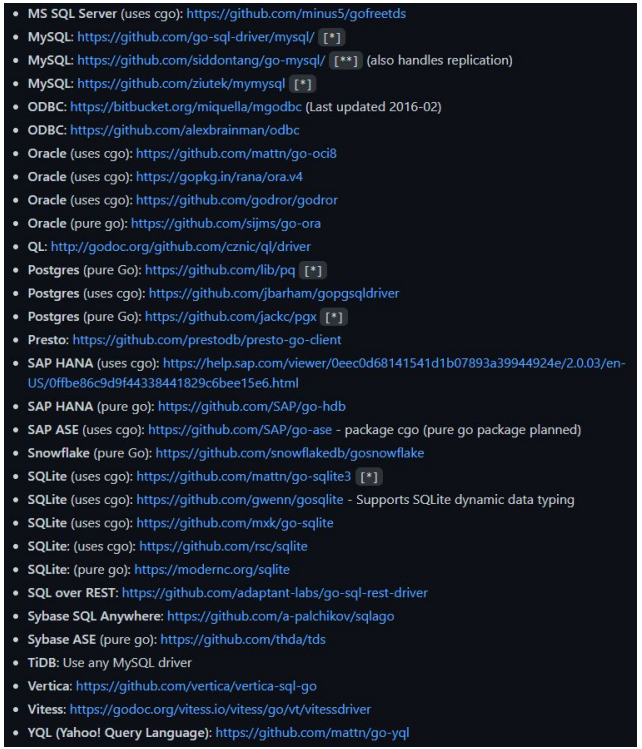
DB DRIVERS

STORAGE IMPLEMENTATION

Sql Drivers

Previamente se indicó que el package database/sql requiere de algún driver para poder cumplir su misión de interactuar con la base de datos. Esto es así, porque cada motor de base de datos opera de forma distinta, a pesar que tengan elementos de funcionamiento en común. Oracle y MySql son ambas DB relacionales y basadas en SQL. Pero no funcionan ni operan de la misma forma. Por esto existe un driver distinto para cada una de ellas. En el siguiente enlace está un listado con todos los drivers disponibles:

https://github.com/golang/go/wiki/SQLDrivers



- **MS SQL Server** (uses cgo): https://github.com/minus5/gofreetds
- **MySQL**: https://github.com/go-sql-driver/mysql/ [*]
- **MySQL**: https://github.com/siddontang/go-mysql/ [**] (also handles replication)
- **MySQL**: https://github.com/ziutek/mymysql [*]
- **ODBC**: https://bitbucket.org/miquella/mgodbc (Last updated 2016-02)
- **ODBC**: https://github.com/alexbrainman/odbc
- **Oracle** (uses cgo): https://github.com/mattn/go-oci8
- **Oracle** (uses cgo): https://gopkg.in/rana/ora.v4
- **Oracle** (uses cgo): https://github.com/godror/godror
- **Oracle** (pure go): https://github.com/sijms/go-ora
- **QL**: http://godoc.org/github.com/cznic/ql/driver
- **Postgres** (pure Go): https://github.com/lib/pq [*]
- **Postgres** (uses cgo): https://github.com/jbarham/gopgsqldriver
- **Postgres** (pure Go): https://github.com/jackc/pgx [*]
- **Presto**: https://github.com/prestodb/presto-go-client
- **SAP HANA** (uses cgo): https://help.sap.com/viewer/0eec0d68141541d1b07893a39944924e/2.0.03/en-US/0ffbe86c9d9f44338441829c6bee15e6.html
- **SAP HANA** (pure go): https://github.com/SAP/go-hdb
- **SAP ASE** (uses cgo): https://github.com/SAP/go-ase - package cgo (pure go package planned)
- **Snowflake** (pure Go): https://github.com/snowflakedb/gosnowflake
- **SQLite** (uses cgo): https://github.com/mattn/go-sqlite3 [*]
- **SQLite** (uses cgo): https://github.com/gwenn/gosqlite - Supports SQLite dynamic data typing
- **SQLite** (uses cgo): https://github.com/mxk/go-sqlite
- **SQLite**: (uses cgo): https://github.com/rsc/sqlite
- **SQLite**: (pure go): https://modernc.org/sqlite
- **SQL over REST**: https://github.com/adaptant-labs/go-sql-rest-driver
- **Sybase SQL Anywhere**: https://github.com/a-palchikov/sqlago
- **Sybase ASE** (pure go): https://github.com/thda/tds
- **TiDB**: Use any MySQL driver
- **Vertica**: https://github.com/vertica/vertica-sql-go
- **Vitess**: https://godoc.org/vitess.io/vitess/go/vt/vitessdriver
- **YQL (Yahoo! Query Language)**: https://github.com/mattn/go-yql

REPOSITORY

STORAGE IMPLEMENTATION

// ¿Qué es la capa Repository?

Los repositorios son componentes que encapsulan la lógica necesaria para el acceso a una fuente de datos.

Interface

Como primer paso diseñamos la interfaz de nuestro Repository:

{}

```go
type Repository interface {
    Store(name, productType string, count int, price float64) (models.Product, error)
    GetOne(id int)
    Update(product models.Product) (models.Product, error)
    GetAll() ([]models.Product, error)
    Delete(id int) error
}
type repository struct {}
func NewRepo() Repository {
    return &repository{}
}
```

{}

```go
package models
type Product struct {
    ID    int     `json:"id"`
    Name  string  `json:"nombre"`
    Type  string  `json:"tipo"`
    Count int     `json:"cantidad"`
    Price float64 `json:"precio"`
}
```

Será necesario también definir un struct llamado Product que será nuestro modelo

Interface

La interfaz diseñada del repository, es la que usará el service para orquestar o generar cada uno de los procesos que requiera. La interfaz debe satisfacer las distintas necesidades de lectura, escritura, actualización y borrado de registros en la DB.

IMPLEMENTACIÓN STORE

STORAGE IMPLEMENTATION

Store

```go
func (r *repository) Store(product models.Product) (models.Product, error) {
    db := db.StorageDB // se inicializa la base
    stmt, err := db.Prepare("INSERT INTO products(name, type, count, price) VALUES( ?, ?, ?, ? )") // se prepara el SQL
    if err != nil {
        log.Fatal(err)
    }
    defer stmt.Close() // se cierra la sentencia al terminar. Si quedan abiertas se genera consumos de memoria
    var result sql.Result
    result, err = stmt.Exec(product.Name, product.Type, product.Count, product.Price) // retorna un sql.Result y un error
    if err != nil {
        return models.Product{}, err
    }
    insertedId, _ := result.LastInsertId() // del sql.Resul devuelto en la ejecución obtenemos el Id insertado
    product.ID = int(insertedId)

    return product, nil
}
```

Store

El bloque de código anterior es la implementación del método Store, el cual permite almacenar un producto nuevo en la DB. Y esto se logra aprovechando los métodos del objeto db. Los dos métodos elementales usados para interactuar con la base en este caso son:

- db.Prepare: Devuelve el statement o sentencia lista para incorporar valores y ejecutar.

- stmt.Exec: Este método pertenece al objeto statement y ejecuta la query preparada con los parámetros que recibe de entrada

IMPLEMENTACIÓN GETONE

STORAGE IMPLEMENTATION

GetOne

{}

```
func (r *repository) GetOne(id int) models.Product {
    var product models.Product
    db := db.StorageDB
    rows, err := db.Query("select * from products where id = ?", id)
    if err != nil {
        log.Println(err)
        return product
    }
    for rows.Next() {
        if err := rows.Scan(&product.ID, &product.Name, &product.Type, &product.Count, &product.Price); err != nil {
            log.Println(err.Error())
            return product
        }
    }
    return product
}
```

GetOne

La implementación de GetOne se fundamenta en tres funciones principales:

- db.Query: Devuelve las filas (objeto rows) encontradas para la consulta dada y los parámetros correspondientes. Devuelve también un error en los casos que corresponda.

- rows.Next: Permite recorrer los resultados obtenidos.

- rows.Scan: Lee los campos de la fila obtenida, y los copia o almacena en las posiciones de memoria de las variables que se indican por parámetros.

IMPLEMENTACIÓN UPDATE

STORAGE IMPLEMENTATION

Update

```
{}
```

```go
func (r *repository) Update(product models.Product) (models.Product, error) {
    db := db.StorageDB       // se inicializa la base
    stmt, err := db.Prepare("UPDATE products SET name = ?, type = ?, count = ?, price = ? WHERE id = ?") // se prepara la sentencia SQL a ejecutar
    if err != nil {
        log.Fatal(err)
    }
    defer stmt.Close()     // se cierra la sentencia al terminar. Si quedan abiertas se genera consumos de memoria
    _, err = stmt.Exec(product.Name, product.Type, product.Count, product.Price, product.ID)
    if err != nil {
        return models.Product{}, err
    }
    return product, nil

}
```

Update

De forma análoga al método Store, Update utiliza los métodos:

- db.Prepare: Devuelve el statement o sentencia lista para incorporar valores y ejecutar.

- stmt.Exec: Este método pertenece al objeto statement y ejecuta la query preparada con los parámetros que recibe de entrada

OPTIMIZACIÓN

STORAGE IMPLEMENTATION

Optimización

- Una buena práctica en la implementación de repositorios, es abstraer todas las queries usadas en constantes fuera del código.

- Al revisar cada método implementado, se evidencia que cada uno de ellos define una variable de esta forma: `db := db.StorageDB`. Esto es repetir código innecesario que puede también abstraerse, esto se efectuará al final de la clase

Optimización

Por ejemplo ahora el método GetOne podría expresarse de esta forma

{}

```
const (
    GetProduct    = "SELECT * FROM products WHERE id = ?"
)
func (r *repository) GetOne(id int) models.Product {
    var product models.Product
    db := db.StorageDB
    rows, err := db.Query(GetProduct, id)
    if err != nil {
        log.Println(err)
        return product
    }
    for rows.Next() {
        err := rows.Scan(&product.ID, &product.Name, &product.Type, &product.Count, &product.Price)
        if err != nil {
            log.Fatal(err)
            return product
        }
    }
    return product
}
```

RECURSOS

STORAGE IMPLEMENTATION

Recursos

Para implementar el repository, es necesario la creación de la DB con la que pueda interactuar, además de su tabla respectiva.

```
CREATE DATABASE IF NOT EXISTS `storage` DEFAULT CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci;
USE `storage`;

CREATE TABLE `products` (
  `id` int(11) NOT NULL,
  `name` varchar(60) NOT NULL,
  `type` varchar(60) NOT NULL,
  `count` int(11) NOT NULL,
  `price` float NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

ALTER TABLE `products`
  ADD PRIMARY KEY (`id`);

ALTER TABLE `products`
  MODIFY `id` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=4;
COMMIT;
```

Recursos

Posterior a la creación de la DB, será necesario comprobar su correcto funcionamiento. Para esto puede diseñarse algún test sencillo que lo valide. Ejemplo:

```go
func TestStore(t *testing.T) {
    product := models.Product{
        Name: "test",
    }
    myRepo := NewRepo()
    productResult, err := myRepo.Store(product)
    if err != nil {
        log.Println(err)
    }
    assert.Equal(t, product.Name, productResult.Name)
}
```

Gracias.

IT BOARDING

BOOTCAMP

Autor: Nelber Mora
Email: nelber.mora@digitalhouse.com
Última fecha de actualización: 17-07-21

IT BOARDING
BOOTCAMP