MySQL Database

Introduction

At some point in time, you want your web application to store and retrieve data from a database. This is almost always the case when you're dealing with dynamic content, serving up forms for users to enter data or storing login and password credentials for your users to authenticate them. For this purpose, we have databases.

Databases come in all forms and shapes. One commonly used database across all of the web is the MySQL database. It has been around for ages and has proven its position and stability more times than you can count.

In this example, we will dive into the fundamentals of database access in Go, create database tables, store data and retrieve it back again.

Installing the `go-sql-driver/mysql` package

The Go programming language comes with a handy package called `database/sql` to query all sorts of SQL databases. This is useful as it abstracts all common SQL features into a single API for you to use. What Go does not include is are database drivers. In Go, database driver is a package which implements the low level details of a specific database (in our case MySQL). As you might have already guessed, this is useful to stay forward compatible. Since, at the time of creating all Go packages, the authors cannot foresee every single database coming to live in the future and supporting every possible database out there would be a large amount of maintenance work.

To install the MySQL database driver, go to your terminal of choice and run:

```
go get -u github.com/go-sql-driver/mysql
```

Connecting to a MySQL database

The first thing we need to check after installing all necessary packages is, if we can connect to our MySQL database successfully. If you don't have a MySQL database server already running, you can start a new instance with Docker easily. Here are the official docs for the Docker MySQL image: https://hub.docker.com/_/mysql

To check if we can connect to our database, import the `database/sql` and the `go-sql-driver/mysql` package and open up a connection like so:

```
import "database/sql"
import _ "go-sql-driver/mysql"


// Configure the database connection (always check errors)
db, err := sql.Open("mysql", "username:password@(127.0.0.1:3306)/dbname?parseTime=t


 // Initialize the first connection to the database, to see if everything works correctly.
// Make sure to check the error.
err := db.Ping()
```

Creating our first database table

Every data entry in our database is stored in a specific table. A database table consists of columns and rows. The columns give each data entry a label and specify the type of it. The rows are the inserted data values. In our first example we want to create a table like this:

id	username	password	created_at
1	johndoe	secret	2019-08-10 12:30:00

Translated to SQL, the command for creating the table will look like this:

```
CREATE TABLE users (
    id INT AUTO_INCREMENT,
    username TEXT NOT NULL,
    password TEXT NOT NULL,
    created_at DATETIME,
    PRIMARY KEY (id)
);
```

Now that we have our SQL command, we can use the `database/sql` package to create the table in our MySQL database:

```
query := `
    CREATE TABLE users (
        id INT AUTO_INCREMENT,
        username TEXT NOT NULL,
        password TEXT NOT NULL,
        created_at DATETIME,
        PRIMARY KEY (id)
    );`

// Executes the SQL query in our database. Check err to ensure there was no error.
_, err := db.Exec(query)
```

Inserting our first user

If you are familiar with SQL inserting new data to our table is as easy as creating our table. One thing to notice is: By default Go uses prepared statements for inserting dynamic data into our SQL queries, which is a way to securely pass user supplied data to our database without the risk of any damage. In the early days of web programming, programmers passed the data directly with the query to the database which caused massive vulnerabilities and could break an entire web application. Please do not do that. It's easy to get it right.

To insert our first user into our database table, we create a SQL query like the following. As you can see, we omit the id column, as it gets automatically set by MySQL. The questionmark tells the SQL driver, that they are placeholders for actual data. This is the where you can see the prepared statements we talked about.

```
INSERT INTO users (username, password, created_at) VALUES (?, ?, ?)
```

We can now use this SQL query in Go and insert a new row into our table:

```
import "time"

username := "johndoe"
password := "secret"
createdAt := time.Now()

// Inserts our data into the users table and returns with the result and a possible
// The result contains information about the last inserted id (which was auto-gener
result, err := db.Exec(`INSERT INTO users (username, password, created_at) VALUES (
```

To grab the newly created id for your user simply get it like this:

```
userID, err := result.LastInsertId()
```

Querying our users table

Now that we have a user in our table, we want to query it and get back all of its information. In Go we have two possibilities to query our tables. There is `db.Query` which can query multiple rows, for us to iterate over and there is `db.QueryRow` in case we only want to query a specific row.

Querying a specific row works basically like every other SQL command we've covered before.
Our SQL command to query one single user by its ID looks like this:

```
SELECT id, username, password, created_at FROM users WHERE id = ?
```

In Go we first declare some variables to store our data in and then query a single database row like so:

```
var (
    id        int
    username  string
    password  string
    createdAt time.Time
)

// Query the database and scan the values into out variables. Don't forget to check
query := `SELECT id, username, password, created_at FROM users WHERE id = ?`
err := db.QueryRow(query, 1).Scan(&id, &username, &password, &createdAt)
```

Querying all users

In the section before we've covered how to query a single user row. Many applications have use cases where you want to query all existing users. This works similar to the example above, but with a bit more coding involved.

We can use the SQL command from the example above and trim off the `WHERE` clause. This way, we query all existing users.

```
SELECT id, username, password, created_at FROM users
```

In Go we first declare some variables to store our data in and then query a single database row like so:

```
type user struct {
    id        int
    username  string
    password  string
    createdAt time.Time
}

rows, err := db.Query(`SELECT id, username, password, created_at FROM users`) // ch
defer rows.Close()

var users []user
for rows.Next() {
    var u user
    err := rows.Scan(&u.id, &u.username, &u.password, &u.createdAt) // check err
    users = append(users, u)
}
err := rows.Err() // check err
```

The users slice now might contain something like this:

```
users {
    user {
        id:        1,
        username:  "johndoe",
        password:  "secret",
        createdAt: time.Time{wall: 0x0, ext: 63701044325, loc: (*time.Location)(nil
    },
    user {
        id:        2,
        username:  "alice",
        password:  "bob",
        createdAt: time.Time{wall: 0x0, ext: 63701044622, loc: (*time.Location)(nil
    },
}
```

Deleting a user from our table

Finally, deleting a user from our table is as straight forward as the `.Exec` from the sections above:

```
_, err := db.Exec(`DELETE FROM users WHERE id = ?`, 1) // check err
```

The Code (for copy/paste)

This is the complete code that you can use to try out the things you've learned from this example.

```
package main

import (
    "database/sql"
    "fmt"
    "log"
    "time"

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

func main() {
    db, err := sql.Open("mysql", "root:root@(127.0.0.1:3306)/root?parseTime=true")
    if err != nil {
        log.Fatal(err)
    }
    if err := db.Ping(); err != nil {
        log.Fatal(err)
    }

    { // Create a new table
```

```
        query := `
            CREATE TABLE users (
                id INT AUTO_INCREMENT,
                username TEXT NOT NULL,
                password TEXT NOT NULL,
                created_at DATETIME,
                PRIMARY KEY (id)
            );`

        if _, err := db.Exec(query); err != nil {
            log.Fatal(err)
        }
    }

    { // Insert a new user
        username := "johndoe"
        password := "secret"
        createdAt := time.Now()

        result, err := db.Exec(`INSERT INTO users (username, password, created_at) \
        if err != nil {
            log.Fatal(err)
        }

        id, err := result.LastInsertId()
        fmt.Println(id)
    }

    { // Query a single user
        var (
            id        int
            username  string
            password  string
            createdAt time.Time
        )

        query := "SELECT id, username, password, created_at FROM users WHERE id = ?
        if err := db.QueryRow(query, 1).Scan(&id, &username, &password, &createdAt)
            log.Fatal(err)
        }

        fmt.Println(id, username, password, createdAt)
    }
```

```
    { // Query all users
        type user struct {
            id        int
            username  string
            password  string
            createdAt time.Time
        }

        rows, err := db.Query(`SELECT id, username, password, created_at FROM users`
        if err != nil {
            log.Fatal(err)
        }
        defer rows.Close()

        var users []user
        for rows.Next() {
            var u user

            err := rows.Scan(&u.id, &u.username, &u.password, &u.createdAt)
            if err != nil {
                log.Fatal(err)
            }
            users = append(users, u)
        }
        if err := rows.Err(); err != nil {
            log.Fatal(err)
        }

        fmt.Printf("%#v", users)
    }

    {
        _, err := db.Exec(`DELETE FROM users WHERE id = ?`, 1)
        if err != nil {
            log.Fatal(err)
        }
    }
}
```