Week 4, Golang Application Development

Azamat Serek, PhD, Assist.Prof.

Create folder for the code

Create a folder for your code

To begin, create a folder for the code you'll write.

1. Open a command prompt and change to your home directory.

   On Linux or Mac:

   ```
   $ cd
   ```

   On Windows:

   ```
   C:\> cd %HOMEPATH%
   ```

   For the rest of the tutorial we will show a $ as the prompt. The commands we use will work on Windows too.

2. From the command prompt, create a directory for your code called data-access.

   ```
   $ mkdir data-access
   $ cd data-access
   ```

Go mod init

2. From the command prompt, create a directory for your code called data-access.

```
$ mkdir data-access
$ cd data-access
```

3. Create a module in which you can manage dependencies you will add during this tutorial.

   Run the `go mod init` command, giving it your new code's module path.

```
$ go mod init example/data-access
go: creating new go.mod: module example/data-access
```

   This command creates a go.mod file in which dependencies you add will be listed for tracking. For more, be sure to see Managing dependencies.

   Note: In actual development, you'd specify a module path that's more specific to your own needs. For more, see Managing dependencies.

Next, you'll create a database.

Setting up a database

Set up a database

In this step, you'll create the database you'll be working with. You'll use the CLI for the DBMS itself to create the database and table, as well as to add data.

You'll be creating a database with data about vintage jazz recordings on vinyl.

The code here uses the MySQL CLI, but most DBMSes have their own CLI with similar features.

1. Open a new command prompt.
2. At the command line, log into your DBMS, as in the following example for MySQL.

```
$ mysql -u root -p
Enter password:

mysql>
```

3. At the `mysql` command prompt, create a database.

```
mysql> create database recordings;
```

4. Change to the database you just created so you can add tables.

```
mysql> use recordings;
Database changed
```

5. In your text editor, in the data-access folder, create a file called create-tables.sql to hold SQL script for adding tables.

6. Into the file, paste the following SQL code, then save the file.

```
DROP TABLE IF EXISTS album;
CREATE TABLE album (
  id         INT AUTO_INCREMENT NOT NULL,
  title      VARCHAR(128) NOT NULL,
  artist     VARCHAR(255) NOT NULL,
  price      DECIMAL(5,2) NOT NULL,
  PRIMARY KEY (`id`)
);

INSERT INTO album
  (title, artist, price)
VALUES
  ('Blue Train', 'John Coltrane', 56.99),
  ('Giant Steps', 'John Coltrane', 63.99),
  ('Jeru', 'Gerry Mulligan', 17.99),
  ('Sarah Vaughan', 'Sarah Vaughan', 34.98);
```

7. From the `mysql` command prompt, run the script you just created.

   You'll use the `source` command in the following form:

```
mysql> source /path/to/create-tables.sql
```

8. At your DBMS command prompt, use a SELECT statement to verify you've successfully created the table with data.

```
mysql> select * from album;
+----+---------------+----------------+-------+
| id | title         | artist         | price |
+----+---------------+----------------+-------+
|  1 | Blue Train    | John Coltrane  | 56.99 |
|  2 | Giant Steps   | John Coltrane  | 63.99 |
|  3 | Jeru          | Gerry Mulligan | 17.99 |
|  4 | Sarah Vaughan | Sarah Vaughan  | 34.98 |
+----+---------------+----------------+-------+
4 rows in set (0.00 sec)
```

Next, you'll write some Go code to connect so you can query.

Database driver

Find and import a database driver

Now that you've got a database with some data, get your Go code started.

Locate and import a database driver that will translate requests you make through functions in the `database/sql` package into requests the database understands.

1. In your browser, visit the SQLDrivers wiki page to identify a driver you can use.

   Use the list on the page to identify the driver you'll use. For accessing MySQL in this tutorial, you'll use Go-MySQL-Driver.

2. Note the package name for the driver – here, `github.com/go-sql-driver/mysql`.

3. Using your text editor, create a file in which to write your Go code and save the file as main.go in the data-access directory you created earlier.

4. Into main.go, paste the following code to import the driver package.

```
package main

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

Connect to database

In this code, you:

- Add your code to a `main` package so you can execute it independently.
- Import the MySQL driver `github.com/go-sql-driver/mysql`.

With the driver imported, you'll start writing code to access the database.

Get a database handle and connect

Now write some Go code that gives you database access with a database handle.

You'll use a pointer to an `sql.DB` struct, which represents access to a specific database.

1. Into main.go, beneath the `import` code you just added, paste the following Go code to create a database handle.

```
var db *sql.DB

func main() {
    // Capture connection properties.
    cfg := mysql.Config{
        User:   os.Getenv("DBUSER"),
        Passwd: os.Getenv("DBPASS"),
        Net:    "tcp",
        Addr:   "127.0.0.1:3306",
        DBName: "recordings",
    }
    // Get a database handle.
    var err error
    db, err = sql.Open("mysql", cfg.FormatDSN())
    if err != nil {
        log.Fatal(err)
    }

    pingErr := db.Ping()
    if pingErr != nil {
        log.Fatal(pingErr)
    }
    fmt.Println("Connected!")
}
```

main.go

2. Near the top of the main.go file, just beneath the package declaration, import the packages you'll need to support the code you've just written.

   The top of the file should now look like this:

```
package main

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

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

3. Save main.go.

Run the code

Run the code

1. Begin tracking the MySQL driver module as a dependency.

   Use the `go get` to add the github.com/go-sql-driver/mysql module as a dependency for your own module. Use a dot argument to mean "get dependencies for code in the current directory."

   ```
   $ go get .
   go get: added github.com/go-sql-driver/mysql v1.6.0
   ```

   Go downloaded this dependency because you added it to the `import` declaration in the previous step. For more about dependency tracking, see Adding a dependency.

2. From the command prompt, set the DBUSER and DBPASS environment variables for use by the Go program.

   On Linux or Mac:

   ```
   $ export DBUSER=username
   $ export DBPASS=password
   ```

   On Windows:

   ```
   C:\Users\you\data-access> set DBUSER=username
   C:\Users\you\data-access> set DBPASS=password
   ```

Go run

3. From the command line in the directory containing main.go, run the code by typing `go run` with a dot argument to mean "run the package in the current directory."

```
$ go run .
Connected!
```

Query for multiple rows

Query for multiple rows

In this section, you'll use Go to execute an SQL query designed to return multiple rows.

For SQL statements that might return multiple rows, you use the `Query` method from the `database/sql` package, then loop through the rows it returns. (You'll learn how to query for a single row later, in the section Query for a single row.)

Write the code

1. Into main.go, immediately above `func main`, paste the following definition of an `Album` struct. You'll use this to hold row data returned from the query.

```
type Album struct {
    ID     int64
    Title  string
    Artist string
    Price  float32
}
```

2. Beneath `func main`, paste the following `albumsByArtist` function to query the database.

```
// albumsByArtist queries for albums that have the specified artist name.
func albumsByArtist(name string) ([]Album, error) {
    // An albums slice to hold data from returned rows.
    var albums []Album

    rows, err := db.Query("SELECT * FROM album WHERE artist = ?", name)
    if err != nil {
        return nil, fmt.Errorf("albumsByArtist %q: %v", name, err)
    }
    defer rows.Close()
    // Loop through rows, using Scan to assign column data to struct fields.
    for rows.Next() {
        var alb Album
        if err := rows.Scan(&alb.ID, &alb.Title, &alb.Artist, &alb.Price); err != nil {
            return nil, fmt.Errorf("albumsByArtist %q: %v", name, err)
        }
        albums = append(albums, alb)
    }
    if err := rows.Err(); err != nil {
        return nil, fmt.Errorf("albumsByArtist %q: %v", name, err)
    }
    return albums, nil
}
```

Update main function

3. Update your `main` function to call `albumsByArtist`.

   To the end of `func main`, add the following code.

```
albums, err := albumsByArtist("John Coltrane")
if err != nil {
    log.Fatal(err)
}
fmt.Printf("Albums found: %v\n", albums)
```

   In the new code, you now:

   - Call the `albumsByArtist` function you added, assigning its return value to a new `albums` variable.
   - Print the result.

Go run

Run the code

From the command line in the directory containing main.go, run the code.

```
$ go run .
Connected!
Albums found: [{1 Blue Train John Coltrane 56.99} {2 Giant Steps John Coltrane 63.99}]
```

Query for a single row

In this section, you'll use Go to query for a single row in the database.

For SQL statements you know will return at most a single row, you can use `QueryRow`, which is simpler than using a `Query` loop.

Write the code

1. Beneath `albumsByArtist`, paste the following `albumByID` function.

```
// albumByID queries for the album with the specified ID.
func albumByID(id int64) (Album, error) {
    // An album to hold data from the returned row.
    var alb Album

    row := db.QueryRow("SELECT * FROM album WHERE id = ?", id)
    if err := row.Scan(&alb.ID, &alb.Title, &alb.Artist, &alb.Price); err != nil {
        if err == sql.ErrNoRows {
            return alb, fmt.Errorf("albumsById %d: no such album", id)
        }
        return alb, fmt.Errorf("albumsById %d: %v", id, err)
    }
    return alb, nil
}
```

2. Update `main` to call `albumByID`.

   To the end of `func main`, add the following code.

```
// Hard-code ID 2 here to test the query.
alb, err := albumByID(2)
if err != nil {
    log.Fatal(err)
}
fmt.Printf("Album found: %v\n", alb)
```

   In the new code, you now:

   - Call the `albumByID` function you added.
   - Print the album ID returned.

Run the code

From the command line in the directory containing main.go, run the code.

```
$ go run .
Connected!
Albums found: [{1 Blue Train John Coltrane 56.99} {2 Giant Steps John Coltrane 63.99}]
Album found: {2 Giant Steps John Coltrane 63.99}
```

Add data

In this section, you'll use Go to execute an SQL `INSERT` statement to add a new row to the database.

You've seen how to use `Query` and `QueryRow` with SQL statements that return data. To execute SQL statements that *don't* return data, you use `Exec`.

Write the code

1. Beneath `albumByID`, paste the following `addAlbum` function to insert a new album in the database, then save the main.go.

```
// addAlbum adds the specified album to the database,
// returning the album ID of the new entry
func addAlbum(alb Album) (int64, error) {
    result, err := db.Exec("INSERT INTO album (title, artist, price) VALUES (?, ?, ?)", alb.Title, alb.Artist, alb.Price)
    if err != nil {
        return 0, fmt.Errorf("addAlbum: %v", err)
    }
    id, err := result.LastInsertId()
    if err != nil {
        return 0, fmt.Errorf("addAlbum: %v", err)
    }
    return id, nil
}
```

2. Update `main` to call the new `addAlbum` function.

   To the end of `func main`, add the following code.

```
albID, err := addAlbum(Album{
    Title:  "The Modern Sound of Betty Carter",
    Artist: "Betty Carter",
    Price:  49.99,
})
if err != nil {
    log.Fatal(err)
}
fmt.Printf("ID of added album: %v\n", albID)
```

   In the new code, you now:

   - Call `addAlbum` with a new album, assigning the ID of the album you're adding to an `albID` variable.

Run the code

From the command line in the directory containing main.go, run the code.

```
$ go run .
Connected!
Albums found: [{1 Blue Train John Coltrane 56.99} {2 Giant Steps John Coltrane 63.99}]
Album found: {2 Giant Steps John Coltrane 63.99}
ID of added album: 5
```

Practice exercise

Make CRUD on models of your choice

References

https://gorm.io/docs/index.html