Dopey Object Relational Mapper

COS 316 Precept 9

Outline

- Conversion Between CamelCase and underscore_case
- Database / SQL Workflow and SQLite Basics

Outline

- Conversion Between CamelCase and underscore_case
- Database / SQL Workflow and SQLite Basics

CamelCase and underscore_case

CamelCase: often used in Go struct identifiers

underscore_case: often used in SQL columns

CamelCase	underscore_case
ComputerScience	computer_science
COSFiles	cos_files
camelCase	camel_case

Define a Word in CamelCase

1. any sequence of uppercase letters that is *not* followed by a non-uppercase character.
2. one uppercase letter followed by any number of non-uppercase characters.
3. an initial sequence of non-uppercase characters.
4. non-alphabetical characters like numbers are treated as non-uppercase characters

examples:

User2FA	==>	[“User2”, “FA”]
COSFiles	==>	[“COS”, “Files”]
camelCase	==>	[“camel”, “Case”]
EMail	==>	[“E”, “Mail”]
OldCOSFiles	==>	[“Old”, “COS”, “Files”]

Convert CamelCase to underscore_case

Step1: split the CamelCase identifier into words;
Step2: lowercase each word;
Step3: join them with _

example:
ComputerScience ==> [“Computer”, “Science”] ==> [“computer”, “science”] ==> computer_science

OldCOSFiles ==> [“Old”, “COS”, “Files”] ==> [“old”, “cos”, “files”] ==> old_cos_files

Code Example: split words

```
func splitCamelWords(s string) []string {
    runes := []rune(s)      // each item is a character
    var words []string
    n := len(runes)
    i := 0

    // Rule 3: initial sequence of non-uppercase characters
    if i < n && !unicode.IsUpper(runes[i]) {
        start := i
        for i < n && !unicode.IsUpper(runes[i]) {
            i++
        }
        words = append(words, string(runes[start:i]))
    }

.....
}
```

Code Example: split words (continued)

```
func splitCamelWords(s string) []string {
......
    for i < n {
        // Now runes[i] should be uppercase
        start := i
        i++

        // Consume consecutive uppercase letters
        for i < n && unicode.IsUpper(runes[i]) { i++  }
        if i-start > 1 && i < n && !unicode.IsUpper(runes[i]) {
            words = append(words, string(runes[start:i-1]))
            start = i - 1
        }

        // Consume following non-uppercase letters for a capitalized word
        for i < n && !unicode.IsUpper(runes[i]) { i++ }

        words = append(words, string(runes[start:i]))
    }

    return words
}
```

Outline

- Conversion Between CamelCase and underscore_case
- Database / SQL Workflow and SQLite Basics

What does a database do?

A Go program has variables in memory.
Example:

```
u := User{ID: 1, Name: "Alice"}
```

But memory disappears when the program exits.
A **database** stores data persistently, so the data is still there later.

- Go structs live in memory
- database rows live on disk
- the ORM helps translate between them

Basic Database Picture

A relational database stores data in **tables**.
A table has:
- **columns**: the kinds of data
- **rows**: individual records

Example table **users**:

id	name	age
1	Alice	20
2	Bob	21

Corresponding Go struct:

```
type User struct {
    ID int
    Name string
    Age int
}
```

- **Go struct type ↔ SQL table**
- **Go struct field ↔ SQL column**
- **Go struct instance ↔ SQL row**

SQL

SQL is the language used to talk to the database.

You do things like:
- create a table
- insert a row
- ask for rows
- update rows
- delete rows

SQL: Create a table

Example:

```
CREATE TABLE users (
    id INTEGER,
    name TEXT,
    age INTEGER
);
```

What this means:

- create a table named `users`
- it has three columns:
  - `id`
  - `name`
  - `age`

This is like defining the schema.

You can connect it to Go:

```
type User struct {
        ID   int
        Name string
        Age  int
}
```

SQL: Insert one row

Example:

```
INSERT INTO users (id, name, age)
VALUES (1, 'Alice', 20);
```

Meaning:

- put one new row into users
- set id = 1
- set name = 'Alice’
- set age = 20

After this, the table looks like:

id	name	age
1	Alice	20

SQL: Read rows

Example:

```
SELECT id, name, age FROM users;
```

Meaning:

- read those columns
- from the users table

Result:

id	name	age
1	Alice	20

If you want just one column:

```
SELECT name FROM users;
```

Result:

name
Alice

SQL: Filter with WHERE

Usually we do not want all rows. We want matching rows.

Example:

```
SELECT id, name, age
FROM users
WHERE id = 1;
```

Meaning:

- look in `users`
- only keep rows where `id` is 1

SQL: Update a row

Example:

```
UPDATE users
SET age = 21
WHERE id = 1;
```

Meaning:

- find the row whose id is 1
- change its age to 21

Now the table becomes:

id	name	age
1	Alice	21

SQL: Delete a row

Example:

```
DELETE FROM users
WHERE id = 1;
```

Meaning:

- remove the row whose `id` is 1

Now that row is gone.

Primary key

One of the most important database concept for this assignment: **primary key**.

A primary key is a column used to uniquely identify each row.

Example:

```
CREATE TABLE users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT
);
```

Here:

- `id` is the primary key
- each row gets a unique `id`

So you can think of `id` as the row's identity.

AUTOINCREMENT

When you write:
id INTEGER PRIMARY KEY AUTOINCREMENT
SQLite can generate the ID for you automatically.
So if you insert a row without manually choosing the ID, SQLite creates a fresh unique one.
Example:
INSERT INTO users (name) VALUES ('Alice');
Then SQLite may create:

id	name
1	Alice

If you insert another row:
INSERT INTO users (name) VALUES ('Bob');
you may get:

id	name
1	Alice
2	Bob

Go database / sql workflow

The big picture

There are three main operations:

- **`Exec`**: for statements that do not return rows

like `CREATE TABLE`, `INSERT`, `UPDATE`, `DELETE`

- **`Query`**: for statements that return **many rows**
- **`QueryRow`**: for statements that should return **one row**

Go database / sql workflow: open connection

Start by opening the database.

```
import (
        "database/sql"
        "log"
)

func main() {
        conn, err := sql.Open("sqlite3",
"file:test.db?mode=memory")
        if err != nil {
                log.Fatal(err)
        }
        defer conn.Close()
}
```

Go database / sql workflow: Exec

Use `Exec` when you do **not** expect rows back.

Examples:

```
_, err = conn.Exec(`
        CREATE TABLE users (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                name TEXT
        )
`)
if err != nil {
        log.Fatal(err)
}
```

and:

```
name := "Alice"
_, err = conn.Exec(`INSERT INTO users (name) VALUES (?)`, name)
if err != nil {
        log.Fatal(err)
}
```

Go database / sql workflow: Exec

Use `Exec` for:

- `CREATE TABLE`
- `INSERT`
- `UPDATE`
- `DELETE`

This matches the test helper style exactly.

A good short rule is:

If I am not reading rows back, I probably want `Exec`.

Go database / sql workflow: Query

When you want multiple rows, use `Query`.
Example:

```
rows, err := conn.Query(`SELECT id, name FROM users`)
if err != nil {
        log.Fatal(err)
}
defer rows.Close()

type User struct {
        ID   int
        Name string
}

var result []User

for rows.Next() {
        ...
}
```

Go database / sql workflow: QueryRow

When you expect at most one row, use `QueryRow`.
Example:

```
var u User
err = conn.QueryRow(
        `SELECT id, name FROM users WHERE id = ?`,
        1,
).Scan(&u.ID, &u.Name)
```

Why DORM exists?

DORM exists to package those repeated patterns into reusable logic.
Instead of writing this every time:

```
rows, err := conn.Query(`SELECT id, name, age FROM users`)
...
err := rows.Scan(&u.ID, &u.Name, &u.Age)
```

you want to be able to say something more like:

```
db.Find(&users)
```

or

```
db.First(&user)
```

or

```
db.Create(&user)
```

DORM turns a repetitive manual workflow to a reusable library interface
Build an abstraction on top of a lower-level API