ISSN: 2583-9888
DOI: doi.org/10.51219/JAIMLD/pallavi-priya-patharlagadda/235

URF PUBLISHERS
connect with research world

Journal of Artificial Intelligence, Machine Learning and Data Science

https://urfpublishers.com/journal/artificial-intelligence

Vol: 1 & Iss: 4

Research Article

Gorm DB Deep Dive and Techniques to Update Default Values to DB

Pallavi Priya Patharlagadda*

Citation: Patharlagadda PP. Gorm DB Deep Dive and Techniques to Update Default Values to DB. *J Artif Intell Mach Learn & Data Sci 2023*, 1(4), 989-994. DOI: doi.org/10.51219/JAIMLD/pallavi-priya-patharlagadda/235

Received: 03 December, 2023; **Accepted:** 28 December, 2023; **Published:** 30 December, 2023

***Corresponding author:** Pallavi Priya Patharlagadda, USA, E-mail: Pallavipriya527.p@gmail.com

Copyright: © 2023 Patharlagadda PP., This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

ABSTRACT

The application usually stores the data in a Database. So, The Data from object-oriented programming languages must be converted into relational database formats and vice versa using an ORM. GORM is one of the most widely used open-source ORM libraries for Go. Relational databases and the Go programming language are two distinct system types that can have their data flows matched and synchronized. Go contains the struct data type, functions, and interfaces that enable object-oriented programming even if it is not a fully object-oriented language. These capabilities also allow you to design models (struct types that reflect certain database tables) using GORM. GORM enables you to perform operations like updating the table data etc. One of the issues we faced in our project is GORM updates() function doesn't update the default values like false, nil, zero, etc to DB when the data to update is provided as a structure. This paper provides the techniques on how to update the default values when using GORM.

1. Introduction

A robust Go library called GORM offers an Object-Relational Mapping (ORM) foundation to make database interactions simpler. Developers can use object-oriented programming paradigms to work with relational databases by utilizing the ORM approach. GORM removes the complexity associated with SQL statements and database connections, making database queries, data manipulation, and management easier. By offering an easy interface for utilizing Go struct types and functions to communicate with databases, GORM transforms database administration in Go. Beyond just making database operations simpler, GORM also supports several database backends, encourages maintainable code, and removes several labor-intensive manual procedures related to running raw SQL queries. You'll see increased productivity and longer-lasting codebases when you incorporate GORM into your Go projects. We shall delve deeper in the upcoming sections.

2. Problem Statement

The GORM library is widely used for Database operations while using the Go language. But currently, the GORM Updates function doesn't update to default values like false, nil, zero, etc. If we pass a structure that contains both default and non-default values, then GORM updates() only update the non-default values. The paper addresses this problem and provides techniques to solve this problem.

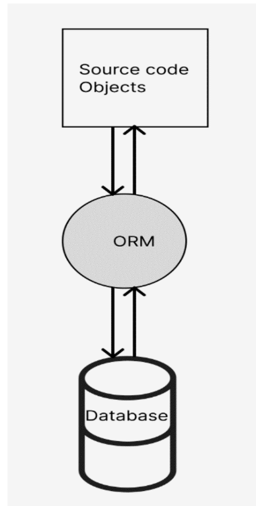
3. What is ORM

ORM stands for Object-Relational Mapping. ORM is a technique that lets you query and manipulate data from a database using an object-oriented paradigm. Object(O) corresponds to the programming language. Relational(R) corresponds to the Relational Database Systems. Mapping(M) is the bridge between the objects and the Database. Popular ORMs are the Prisma ORM for JavaScript, TypeScript, Hibernate for Java, and SQLAlchemy for Python.

Advantages of ORM:

- Development times are accelerated by using ORM.
- The ORM tools help in eliminating SQL injection attacks largely thereby increasing the security.

- You may utilize the database of your choice in the language of your choice since ORM is compatible with a wide range of programming languages.
- You may utilize custom queries using ORM as it supports them



Disadvantages of ORM:

- There is a learning curve to know about ORM tools.
- In the case of extremely complex queries, ORM performance degrades.
- ORMs are usually slower than directly using a database.

4. Gorm

The most widely used ORM in the Go environment is called GORM. A full-featured, developer-friendly, Code-first ORM framework called GORM (Go-ORM) streamlines the creation, retrieval, updating, and deletion of records by automating database operations. The GORM library provides a standardized interface to work with your data, independent of the underlying database, and supports several databases, including PostgreSQL and MySQL. Along with support for various database drivers and common SQL databases including MySQL, SQLite, PostgreSQL, and Microsoft SQL server, GORM offers drivers and functionality like associations, auto migration, SQL creation, logging, and hooks for database operations.

5. Advantages of Gorm

Below are some of the advantages of GORM.

Enhanced Database Operations:

By abstracting away the intricacies of SQL queries, GORM facilitates the execution of frequently used database operations, including INSERT, UPDATE, DELETE, and SELECT.

Database-Agnostic:

You can switch databases without having to rewrite your code since GORM supports a variety of database backends. A variety of databases are supported, including PostgreSQL, MySQL, and SQLite.

Model-Driven Development:

GORM promotes a model-driven methodology in which Go struct types are used to create your database schema. This method guarantees that the database schema and the data structure of your application are consistent.

Automatic Migrations:

In place of manual schema migration scripts, GORM may automatically generate or update database tables in response to modifications in your Go struct types.

Query Building:

With GORM's extensive collection of query-building techniques, you may use a fluent API to create intricate queries.

6. Disadvantage of GORM

elow are some of the disadvantages of using GORM.

- For developers who are unfamiliar with GORM or ORMs in general, there can be a steep learning curve and mastery of the tool may require a substantial time commitment.
- A thorough understanding of association structs tags, which are used to specify foreign keys and establish associations between tables, is necessary for generating a DB schema with GORM.
- Restricted authority over the database optimizer and underlying SQL queries
- According to benchmark results, GORM may have performance overhead, particularly for multiple-row Insert and Update operations, when working with sophisticated queries or big data sets.

Before utilizing GORM, it is advised to check the benchmarks and consider any potential performance impact if your project comprises such tasks.

This article will go into detail on how to use the GORM library in Go to execute database operations. The steps for configuring the development environment, establishing a connection to a PostgreSQL database, executing fundamental Create and Update tasks, and exploring the advanced functionalities of the ORM library. This article will offer a thorough guide for utilizing the GORM library to manage data efficiently.

7. Prerequisites

Make sure the following prerequisites are met before beginning the installation procedure.

- Knowledge of the basics of the Go programming language.
- Good to have some experience with relational databases, like PostgreSQL or MySQL.
- Understanding fundamental SQL syntax, tables, columns, and queries (Create and Update operations) can aid you in understanding the database-related ideas covered in this tutorial.
- Install Go on your computer.
- An existing PostgreSQL database is configured, and access to the database is authorized for a user.

8. Installing Gorm

The go-get command can be used to install GORM by retrieving the required packages from the Go module repository. Run the following command on an open terminal or command prompt.

go get -u github.com/go-gorm/gorm

This command would fetch the most recent version of the GORM and its dependencies, ensuring you have the most up-to-date version of the library.

The PostgreSQL database driver needs to be installed. Only then GORM library will be able to connect to the PostgreSQL database and execute database operations without any issues. To install the PostgreSQL driver, use the following command.

go get -u gorm.io/driver/postgres

By doing this, the PostgreSQL driver made especially for the GORM library will be installed.

Additionally, database drivers for MySQL, SQLite, SQL Server, TiDB, and Clickhouse are available through the GORM library. To learn how to utilize these database drivers and build your own, check the GORM library documentation.

We can connect to your PostgreSQL database and carry out database operations

9. EStablishing a database connection with the gorm library

1. Importing the GORM library and the PostgreSQL driver package is the first step.

```
package main
import (
   "gorm.io/driver/postgres"
   "gorm.io/gorm"
)
```

The PostgreSQL database connection information, including the host, user, password, dbname, and port, must then be provided. The Data Source Name (DSN), a connection string, is created using these details. The DSN provides connection details to the database.

```
func Dsn(host, user, password, dbname string, port int, sslmode string) string {
        dsn := fmt.Sprintf(
                "host=%s user=%s password=%s dbname=%s port=%d sslmode=%s",
                host, user, password, dbname, port, sslmode,
        )
        return dsn
}
```

Include the subsequent code in your main.

```
func main() {
   dsn := Dsn(host, user, password, dbname, port, sslmode)
   dialector := postgres.Open(dsn)
   db, err := gorm.Open(dialector, &gorm.Config{}
)
   if err != nil {
     panic(«Database connection Failed: « + err.Error())
   }
}
```

The above code performs below functionality.

The gorm is used to make a connection to the PostgreSQL database. The Open() function accepts two arguments. PostgreSQL driver DSN details is the first argument for gorm.Open(dsn), and the configuration, &gorm.Config{}, is the second argument. An error (err) and a database instance (db) are returned by this function.

Lastly, it determines whether an error (err!= nil) happened during the database connection. The panic() function is executed in the event of an error, concatenating the error message "Database Connection Failed:" + err.Error(). As a result, the application crashes and prints the error message that was supplied along with the actual error, which shows that the database connection attempt was unsuccessful.

10. Performing create a record with the Gorm library

The next step is to construct a Go struct that represents the model for the associated database table after you have connected to the database using the GORM package. The schema or design for interacting with the data in the table will be provided by this struct. Here is an example.

```
import (
   «time»
)
type Person struct {
   gorm.Model
   FirstName      string   `gorm:»uniqueIndex»`
   LastName       string   `gorm:»uniqueIndex»`
   Email          string   `gorm:»not null»`
   City           string   `gorm:»not null»`
   Role           string   `gorm:»not null»`
   Age            int      `gorm:»not null;size:۳»`
   HasSubscription bool    `gorm:»default:false»`
   CreatedAt      time.Time `gorm:»autoCreateTime»`
   UpdatedAt      time.Time `gorm:»autoUpdateTime»`
   DeletedAt      gorm.DeletedAt
}
```

NOTE: The GORM library provides extra metadata or instructions to the ORM framework through tags, which are annotations applied to struct fields using backticks (``). These tags are essential for configuring table associations, defining column names, mapping struct fields to database columns, declaring data types, and imposing restrictions. They offer crucial details regarding the composition and properties of the database table and its columns.

The time package is imported in the code example above to keep track of the creation, updates, and deletions of Person records. The fields in the Person struct are then defined to match the columns in the database table. The functions of each field and the related tag are as follows:

Gorm. Model:

This field contains an embed of the gorm.Model struct, which offers standard fields for tracking the metadata of the model, such as ID, CreatedAt, UpdatedAt, and DeletedAt.

FirstName and LastName:

These fields include the person's first and last name. They also feature the gorm: "uniqueIndex" tag, which indicates that the first and last name combination is unique in the database and that two-person entries cannot have the same first and last name.

Email, Country, Role:

These are fields that reflect the person's country, email address, and role. They are marked with the gorm:"not null"

tag, indicating that they are mandatory fields that cannot have an empty value.

Age:

The gorm: "not null;size:3" tag is present in this field, which indicates the Person's age. There is a three-digit limit on the size of this integer field.

HasSubscription:

This field indicates if the person has a subscription or not. gorm: "default: false" tag, indicates that false is the default value.

CreatedAt and UpdatedAt:

The timestamps for the creation and updating of the Person record are indicated by the fields CreatedAt and UpdatedAt. They belong to a certain type. When a new Person record is created or edited, time and have the gorm: "autoCreateTime" and gorm: "autoUpdateTime" tags, respectively, indicating that they should be automatically populated with the current timestamp.

DeletedAt:

The type of this field is gorm.DeletedAt and are utilized in GORM to manage soft-deletes. Instead of physically deleting a record from the database, it enables GORM to handle logical deletions and keep track of the deletion timestamp of a Person's record.

"Soft delete" refers to a technique in the GORM library where records are tagged as deleted rather than being physically destroyed from the database. When a record is deleted, the GORM library sets the value of a DeletedAt field with the gorm. DeletedAt type in your Go struct, signaling that the record is deleted but is still stored in the database. Benefits include data integrity control, fast recovery and restoration of erased records, and preservation of a history of modifications. You can use the GORM library to do a hard delete by physically removing the entries from the database. when you choose not to use the soft delete feature you can omit the DeletedAt field from your Go struct.

11. Create a record with Gorm library

Adding records to the database is a basic database activity. Using the Create() function from the GORM library, a new instance of the matching struct-in this case, Person-must be defined and saved to the database. By producing the required SQL commands for insertion automatically, the GORM library streamlines the procedure**.**

Whether you specify field names in struct using capital or lowercase letters when using the GORM library, the field names will be automatically transformed to lowercase letters in the database. An underscore (_) will be used in the database to separate words in field names that contain more than one.

```
import "fmt"

func main() {

  newPerson := Person{
    FirstName:     «Alice»,
    LastName:      «Bob»,
    Email:         «AliceBob@gmail.com»,
    Country:       «Sweden»,
    Role:          «Artist»,
    Age:           ٣٠,
    Hassubscription: true,
  }
  // ... Create a new Person record
  result := db.Create(&newPerson)
  if result.Error != nil {
     panic(«failed to create record newPerson: « + result.Error.
Error())
  }
  // ... Handle successful creation ...
  fmt.Printf(«New Person ٪s ٪s was created successfully!\\n",
newPerson.FirstName, newPerson.LastName)
}
```

In the above example, a new Person struct instance with the desired field values for the person, such as first_name, last_name, email, country, role, age, and subscription. The above example code performs the below functions.

- Establishes a new Person record in the database using the db.Create(&newPerson) function. The GORM library can update the newPerson struct with an auto-generated primary key and other database-managed fields by using the &newPerson pointer, which points to the newPerson struct.
- Look for any mistakes that might have happened throughout the creation procedure. The program panics and shows the error message "failed to create record newPerson: " + result if a problem occurs.Failed.Error() in addition to the real error.
- Finally, if the creation of the Person record is successful, fmt.Printf is used to print a confirmation message stating that the new Person has been created. The newly added Person's initial and last name are included in the message.

On executing the above go file, the new person will be added successfully to the Database. You can connect to the DB and check for the entry manually.

12. Update a record with Gorm library

GORM provides multiple ways of updating a record based on requirements.

13. Save

It saves all the fields specified in the structure. Save is a combination function. If the saved value does not contain a primary key, it will execute Create, otherwise, it will execute Update (with all fields). Below is an example.

```
var person Person
result := db.First(&person, ١)
if result.Error != nil {
   panic(«failed to retrieve person from DB: « + result.Error.
Error())
}
// Modify the attributes of the retrieved record
person.FirstName = «Bob»
person.LastName = «Smith»
person.Email = «BobSmith@example.com»

// Save the changes back to the database
result = db.Save(&person)
if result.Error != nil {
   panic(«failed to update person: « + result.Error.Error())
}
```

The code above uses the first method of the db object to fetch a particular Person record from the database. The code will panic and display the error message if a retrieval process error occurs.

Next, it makes changes to the first_name, last_name, and email of the record that was retrieved. The Save method is used to save the modifications to the database. The code will panic and display the error message if an error occurs during the updating process.

NOTE:

1. Save with Model should not be used; it is an undefinable behavior.

Advantages of Using Save:

Simpleness:

All fields are updated with a single method call.

Complete Update:

Assures that the current state of the struct is the same as the record in the database.

Disadvantages of Using Save:

Performance:

It may not be essential to update every column, which could cause operations to lag and be inefficient.

Danger of Overwriting Data:

Some fields that were previously filled in the database will be overwritten if the struct has default zero values for such fields.

14. Update

The update method updates specific fields in the structure. Here is an example.

```
result := db.Model(&Person{}).Where("id = ?", 1).Update("role", "Programmer")
if result.Error != nil {
    panic("failed to update Person: " + result.Error.Error())
}
```

In the above example, we are fetching the record with id=1 and update the role. After a successful update, the role of the Person should be updated to Programmer.

Advantages of Using Update:

Greater control over the data as it updates only the provided fields, preventing inadvertent negative consequences on related data.

Improved efficiency while changing individual fields because the database isn't overwritten needlessly.

Disadvantages of Using Update:

Additional intricacy because each relationship needs to be handled explicitly while updating connected data.

Possibility of errors or inconsistent data if the developer neglects to manually manage updates to related data.

15. Updates

Updates update multiple columns of struct. Here is an example.

```
result := db.Model(&Person{}).Where("id = ?", 1).Updates(Person{
FirstName: "Alice",
    LastName: "Bob",
    Email: "AliceBob@example.com",
        Hassubscription: false
 })
 if result.Error != nil {
    panic("failed to update Person: " + result.Error.Error())
 }
```

In the above example, we are updating the fields FirstName, LastName, Email, and Has Subscription fields. After executing the above code, you will be surprised to see that HasSubscription would still hold the value of True while the other attributes like FirstName, LastName, and Email get updated. The problem here is that Updates() doesn't update the field if we provide Fiels's default value.

In several projects, updates were preferred over update and save as it updates only specific fields by providing the structure thereby increasing the performance. But we miss the point that Updates won't update the default values when update data is provided as a structure. So, how can we overcome the problem?

This can be solved in two ways.

- Have a separate update function for updating the default values. If we have multiple values to update like the above example, you can give the non-default values and those will be updated. To Update to default values, have a separate update function that would perform the update. Below is an example.

```
result := db.Model(&Person{}).Where("id = ?", 1).Update("HasSubscription", false)
 if result.Error != nil {
    panic("failed to update Person: " + result.Error.Error())
 }
```

This would update the HasSubscription to false

- The second way is to define everything in a map instead of structure to updates. This would update even though default values are provided. Below is an example.

```
result := db.Model(&Person{}).Where("id = ?", 1).Updates(map[string]interface{}{
    "FirstName": "Alice",
    "LastName": "Bob",
    "Email": "AliceBob@example.com",
    "Hassubscription": false
 })
 if result.Error != nil {
    panic("failed to update Person: " + result.Error.Error())
 }
```

To maximize the performance of your Go applications with the GORM library, consider the below best practices:

Recognize the scope of Your Tasks:

When you need to make sure the whole object graph accurately representing the state of your application's model, use Save. If you are simply making changes to a portion of the model and require accuracy, use Update.

Carefully Manage Associations: You will have to carefully manage associated data updates while using Update. Take care of this to prevent inconsistent data.

Performance Aspects: An update can help in improving the performance when only a portion of the data must be updated with huge datasets or high-load scenarios.

16. Conclusion

As discussed, GORM is a strong ORM package for Go that makes database management easier and boosts efficiency.

GORM offers two effective techniques for updating database entries: Save and Update. Gaining an understanding of their distinctions and the proper context for each will greatly improve the dependability and performance of your application. You can guarantee that your database interactions are productive and efficient by selecting the appropriate approach for the given circumstance.

17. References

1. https://gorm.io/docs/
2. https://stackoverflow.com/questions/1279613/what-is-an-orm-how-does-it-work-and-how-should-i-use-one
3. https://hyperskill.org/learn/step/20695?utm_source=medium_hs&utm_medium=social&utm_campaign=hsms-193_gorm&utm_content=article&utm_term=09.03.23
4. https://www.freecodecamp.org/news/what-is-an-orm-the-meaning-of-object-relational-mapping-database-tools/
5. https://dev.to/techschoolguru/how-to-handle-db-errors-in-golang-correctly-11ek
6. https://medium.com/readytowork-org/updating-nil-value-with-gorm-mapping-using-reflect-212cec203822
7. https://medium.com/@speedforcerun/golang-how-to-solve-gorm-not-updating-data-when-you-may-have-none-zero-field-724ccb351e8b
8. https://stackoverflow.com/questions/64330504/update-method-does-not-update-zero-value
9. https://medium.com/@kadergenc/what-is-orm-why-is-it-used-what-are-its-pros-and-cons-3ed77c0e6ed2
10. https://www.red-gate.com/simple-talk/featured/how-to-use-any-sql-database-in-go-with-gorm/
11. https://techwasti.com/simplifying-database-interactions-a-comprehensive-guide-to-installing-and-setting-up-gorm-in-go