Go with GORM

In Chapter-9 of our Golang Tutorial, we touched upon 'Go Database/SQL'. In this chapter, let's explore 'Go with GORM'.

The GORM is fantastic ORM library for Golang, aims to be developer friendly. It is an ORM library for dealing with relational databases. This gorm library is developed on the top of *database/sql* package.

The overview and feature of ORM are:

- Full-Featured ORM (almost)
- Associations (Has One, Has Many, Belongs To, Many To Many, Polymorphism)
- Callbacks (Before/After Create/Save/Update/Delete/Find)
- Preloading (eager loading)
- Transactions
- Composite Primary Key
- SQL Builder
- Logger
- Developer Friendly

To install GORM just use the following command :

go get "github.com/jinzhu/gorm"

In order to use, just import this package into your project along with the database drivers as you want

```
import (
 _ "github.com/go-sql-driver/mysql"
 "github.com/jinzhu/gorm"
 _ "github.com/jinzhu/gorm/dialects/mysql" //You could import dialect
 )
```

Now use the gorm to do the operations on the database.

In order to connect to the database, just use the following syntax.

db, err := gorm.Open("mysql", "user:password@/dbname?charset=utf8&parseTime=True&loc=Local")

NOTE: In order to handle time.Time, you need to use parseTime parameter

Here you have to manually create the database before you connect.

For PostgreSQL, *db, err := gorm.Open("postgres", "user=gorm dbname=gorm sslmode=disable")*

And remember to close the database when it is not in use using defer defer db.Close()

```
main.go
 import (
 "log"
 "github.com/jinzhu/gorm"
 _ "github.com/jinzhu/gorm/dialects/mysql"
)

func main() {
 db, err := gorm.Open("mysql", "root:root@tcp(127.0.0.1:3306)/ormdemo?
charset=utf8&parseTime=True")
defer db.Close()
 if err!=nil{
 log.Println("Connection Failed to Open")
 }
 log.Println("Connection Established")
}
```

Creating Models and Tables

Define the models before creating tables and based on the model the table will be created.

```
type User struct {
 ID int
 Username string
}

func main() {
 // After db connection is created.
```

```
 db.CreateTable(&User{})

 // Also some useful functions
 db.HasTable(&User{}) // =>;; true
 db.DropTableIf Exists(&User{}) //Drops the table if already exists
}
```

Auto Migration

This Auto Migration feature will automatically migrate your schema. It will automatically create the table based on your model. We don't need to create the table manually.

```
db.Debug().AutoMigrate(&User{}) //Model or Struct

 main.go
 package main
import (
 _ "github.com/go-sql-driver/mysql"
 "github.com/jinzhu/gorm"
 _ "github.com/jinzhu/gorm/dialects/mysql"
 "log"
)

type UserModel struct{
 Id int `gorm:"primary_key"`
 Name string
 Address string
}

func main(){
 db, err := gorm.Open("mysql", "root:root@tcp(127.0.0.1:3306)/ormdemo?
charset=utf8&parseTime=True")
 if err != nil {
 log.Panic(err)
 }
 log.Println("Connection Established")
 db.Debug().DropTableIfExists(&UserModel{})
//Drops table if already exists
 db.Debug().AutoMigrate(&UserModel{})
//Auto create table based on Model
}
```

```
Console
<terminated> <exit code: 0> godemo - 26_GO_GORM_demo [Go Application] E:\golar
2017/09/29 08:40:07 Connection Established
```

Now just go and check the "ormdemo" database, there you will find the table along with the columns.

Note: You need to create the database manually.

The gorm also has given model definition including fields like Id, CreatedAt, UpdatedAt, DeletedAt. If you want to use just embed *gorm.Model* in your model/struct.

```
// Model's definition given by gorm
type Model struct {
 ID uint `gorm:"primary_key"`
 CreatedAt time.Time
 UpdatedAt time.Time
 DeletedAt *time.Time
}

type User struct {
 gorm.Model // fields `ID`, `CreatedAt`, `UpdatedAt`, `DeletedAt`will be added
 Name string
}
```

In the gorm.Model the fields

CreatedAt – used to store records created time

UpdatedAt – used to store records updated time

DeletedAt – used to store records deleted time, It won't delete the records just set the value of DeletedAt's field to the current time and you won't find the record when querying i.e. what we call soft deletion.

If you want to set some SQL parameters to the model fields then you can do like this

```
type UserModel struct{
 Id int `gorm:"primary_key";"AUTO_INCREMENT"`
 Name string `gorm:"size:255"`
 Address string `gorm:"type:varchar(100)"`
}
```

Gorm is creating a table with the plural version of the model name like if your model name is UserModel then gorm is creating the tables in its plural version user_models. So in order to avoid this just do *db.SingularTable(true)*.

In gorm, ID field is automatically set to a Primary key field with

auto increment property.

CRUD Operations

The query for the SQL using gorm can be specified like this

Create/Insert

In order to create or insert a record, you need to use the Create() function. The save() is also there to that will return the primary key of the inserted record.

```
user:=&UserModel{Name:"John",Address:"New York"}
db.Create(user)

 Internally it will create the query like
 INSERT INTO `user_models` (`name`,`address`) VALUES ('John','New York')

 //You can insert multiple records too
var users []UserModel = []UserModel{
 UserModel{name: "Ricky",Address:"Sydney"},
 UserModel{name: "Adam",Address:"Brisbane"},
 UserModel{name: "Justin",Address:"California"},
}

for _, user := range users {
 db.Create(&user)
}
```

Update

In order to update the records in the table using gorm, look into the below sample example.

```
user:=&UserModel{Name:"John",Address:"New York"}
// Select, edit, and save
db.Find(&user)
user.Address = "Brisbane"
db.Save(&user)

// Update with column names, not attribute names
db.Model(&user).Update("Name", "Jack")

db.Model(&user).Updates(
 map[string]interface{}{
 "Name": "Amy",
 "Address": "Boston",
 })

// UpdateColumn()
db.Model(&user).UpdateColumn("Address", "Phoenix")
db.Model(&user).UpdateColumns(
 map[string]interface{}{
 "Name": "Taylor",
 "Address": "Houston",
 })
// Using Find()
db.Find(&user).Update("Address", "San Diego")

// Batch Update
db.Table("user_models").Where("address = ?", "california").Update("name", "Walker")
```

Delete

In order to delete the record from the table, gorm has provided Delete() as given in below examples

```
// Select records and delete it
db.Table("user_models").Where("address= ?", "San Diego").Delete(&UserModel{})

//Find the record and delete it
db.Where("address=?", "Los Angeles").Delete(&UserModel{})

// Select all records from a model and delete all
db.Model(&UserModel{}).Delete(&UserModel{})
```

Queries

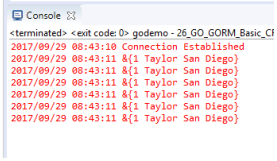
In order to fetch the records from the database and do some SQL stuffs gorm has given some query functions. We'll now do a quick discussion on it.

```
// Get first record, order by primary key
db.First(&user)
// Get last record, order by primary key
db.Last(&user)
// Get all records
db.Find(&users)
// Get record with primary key (only works for integer primary key)
db.First(&user, 10)

Query with Where() [some SQL functions]

db.Where("address = ?", "Los Angeles").First(&user)
//SELECT * FROM user_models WHERE address='Los Angeles' limit 1;

db.Where("address = ?", "Los Angeles").Find(&user)
//SELECT * FROM user_models WHERE address='Los Angeles';

db.Where("address <> ?", "New York").Find(&user)
//SELECT * FROM user_models WHERE address<>'Los Angeles';

// IN
db.Where("name in (?)", []string{"John", "Martin"}).Find(&user)

// LIKE
```

```
db.Where("name LIKE ?", "%ti%").Find(&user)

// AND
db.Where("name = ? AND address >= ?", "Martin", "Los Angeles").Find(&user)
```

Now just go through the program.

```
main.go
package main

import (
 _ "github.com/go-sql-driver/mysql"
 "github.com/jinzhu/gorm"
 _ "github.com/jinzhu/gorm/dialects/mysql"
 "log"
)

type UserModel struct{
 Id int `gorm:"primary_key";"AUTO_INCREMENT"`
 Name string `gorm:"size:255"`
 Address string `gorm:"type:varchar(100)"`
}

func main(){
 db, err := gorm.Open("mysql", "root:root@tcp(127.0.0.1:3306)/ormdemo?
charset=utf8&parseTime=True")
 if err != nil {
 log.Panic(err)
 }
 log.Println("Connection Established")
 db.DropTableIfExists(&UserModel{})
 db.AutoMigrate(&UserModel{})

 user:=&UserModel{Name:"John",Address:"New York"}
 newUser:=&UserModel{Name:"Martin",Address:"Los Angeles"}

 //To insert or create the record.
 //NOTE: we can insert multiple records too
 db.Debug().Create(user)
 //Also we can use save that will return primary key
 db.Debug().Save(newUser)

 //Update Record
 db.Debug().Find(&user).Update("address", "California")
 //It will update John's address to California

 // Select, edit, and save
 db.Debug().Find(&user)
 user.Address = "Brisbane"
 db.Debug().Save(&user)

 // Update with column names, not attribute names
 db.Debug().Model(&user).Update("Name", "Jack")

 db.Debug().Model(&user).Updates(
 map[string]interface{}{
 "Name": "Amy",
 "Address": "Boston",
 })

 // UpdateColumn()
 db.Debug().Model(&user).UpdateColumn("Address", "Phoenix")
 db.Debug().Model(&user).UpdateColumns(
 map[string]interface{}{
 "Name": "Taylor",
 "Address": "Houston",
 })
 // Using Find()
 db.Debug().Find(&user).Update("Address", "San Diego")

 // Batch Update
 db.Debug().Table("user_models").Where("address = ?", "california").Update("name", "Walker")

 // Select records and delete it
 db.Debug().Table("user_models").Where("address= ?", "San Diego").Delete(&UserModel{})

 db.Debug().Where("address = ?", "Los Angeles").First(&user)
 log.Println(user)
 db.Debug().Where("address = ?", "Los Angeles").Find(&user)
 log.Println(user)
 db.Debug().Where("address <> ?", "New York").Find(&user)
 log.Println(user)
 // IN
 db.Debug().Where("name in (?)", []string{"John", "Martin"}).Find(&user)
 log.Println(user)
 // LIKE
 db.Debug().Where("name LIKE ?", "%ti%").Find(&user)
 log.Println(user)
 // AND
 db.Debug().Where("name = ? AND address >= ?", "Martin", "Los Angeles").Find(&user)
 log.Println(user)

 //Find the record and delete it
 db.Where("address=?", "Los Angeles").Delete(&UserModel{})

 // Select all records from a model and delete all
 db.Debug().Model(&UserModel{}).Delete(&UserModel{})
}
```



Transaction

```
tx := db.Begin()
err := tx.Create(&user).Error
if err != nil {
 tx.Rollback()
}
tx.Commit()
```

Associations

The relationship defines how the structs or models interact with each other. So for this, you need to create/define what kind of relationship at both ends.

One To One Relationship

One to One Relationship specifies how the fields of one models are related with other by specifying one to one mapping. For now, I've considered and done one to one mapping between Place and Town struct/model. Here one Town belongs to one Place relational mapping I've created.

```
Place.go
package model
import ()
type Place struct {
 ID int `gorm:primary_key`
 Name string
 Town Town
 TownId int `gorm:"ForeignKey:id"` //this foreignKey tag didn't works
}

Town.go
package model
import ()
type Town struct {
 ID int `gorm:"primary_key"`
 Name string
}

main.go
package main

import (
 _ "database/sql"
 _ "github.com/go-sql-driver/mysql"
 "github.com/jinzhu/gorm"
 "26_GO_GORM/One2One_Relationship/model"
 "fmt"
)

//var Db *gorm.Db

func main() {
 //Init Db connection

 Db, _ := gorm.Open("mysql", "root:root@tcp(127.0.0.1:3306)/testmapping?
charset=utf8&parseTime=True")
 defer Db.Close()

 Db.DropTableIfExists(&model.Place{}, &model.Town{})

 Db.AutoMigrate(&model.Place{}, &model.Town{})
 //We need to add foreign keys manually.
 Db.Model(&model.Place{}).AddForeignKey("town_id", "towns(id)", "CASCADE", "CASCADE")

 t1 := model.Town{
 Name: "Pune",
 }
 t2 := model.Town{
 Name: "Mumbai",
 }
 t3 := model.Town{
 Name: "Hyderabad",
 }

 p1 := model.Place{
 Name: "Katraj",
 Town: t1,
 }
 p2 := model.Place{
 Name: "Thane",
 Town: t2,
 }
 p3 := model.Place{
 Name: "Secundarabad",
 Town: t3,
 }

 Db.Save(&p1) //Saving one to one relationship
 Db.Save(&p2)
 Db.Save(&p3)

 fmt.Println("t1==>", t1, "p1==>", p1)
 fmt.Println("t2==>", t2, "p2s==>", p2)
 fmt.Println("t2==>", t3, "p2s==>", p3)

 //Delete
 Db.Where("name=?", "Hyderabad").Delete(&model.Town{})

 //Update
 Db.Model(&model.Place{}).Where("id=?", 1).Update("name", "Shivaji Nagar")

 //Select
 places := model.Place{}
 towns := model.Town{}
 fmt.Println("Before Association", places)
 Db.Where("name=?", "Shivaji Nagar").Find(&places)
 fmt.Println("After Association", places)
 err := Db.Model(&places).Association("town").Find(&places.Town).Error
 fmt.Println("After Association", towns, places)
 fmt.Println("After Association", towns, places, err)

 defer Db.Close()
}
```

```
Console
<terminated> <exit code: 0> godemo - 26_GO_GORM_One2One_Relationship [Go Applic
t1==> {0 Pune} p1==> {1 Katraj {1 Pune} 1}
t2==> {0 Mumbai} p2s==> {2 Thane {2 Mumbai} 2}
t2==> {0 Hyderabad} p2s==> {3 Secundarabad {3 Hyderabad} 3}
Before Association {0  {0 } 0}
After Association {1 Shivaji Nagar {0 } 1}
```

```
After Association {0 } {1 Shivaji Nagar {1 Pune} 1}
After Association {0 } {1 Shivaji Nagar {1 Pune} 1} <nil>
```

Note: Here in the example, you need to create the foreign keys manually using AddForeignKey() function because auto-migration of the foreign key is not happening.

One To Many Relationship

In One to Many relationships, models of two classes are related by specifying one to many mapping. Here in the example, I've created the mapping like one customer has many contacts.

```
main.go
package main
import (
 "fmt"
 _ "github.com/go-sql-driver/mysql"
 "github.com/jinzhu/gorm"
)
type Customer struct {
 CustomerID int `gorm:"primary_key"`
 CustomerName string
 Contacts []Contact `gorm:"ForeignKey:CustId"` //you need to do like this
}

type Contact struct {
 ContactID int `gorm:"primary_key"`
 CountryCode int
 MobileNo uint
 CustId int
}

func main() {

 db, err := gorm.Open("mysql", "root:root@tcp(127.0.0.1:3306)/testmapping?
charset=utf8&parseTime=True")
 if err != nil {
 panic(err.Error())
 }
 defer db.Close()

 db.DropTableIfExists(&Contact{}, &Customer{})
 db.AutoMigrate(&Customer{}, &Contact{})
 db.Model(&Contact{}).AddForeignKey("cust_id", "customers(customer_id)", "CASCADE", "CASCADE") //
Foreign key need to define manually

 Custs1 := Customer{CustomerName: "John", Contacts: []Contact{
 {CountryCode: 91, MobileNo: 956112},
 {CountryCode: 91, MobileNo: 997555}}}

 Custs2 := Customer{CustomerName: "Martin", Contacts: []Contact{
 {CountryCode: 90, MobileNo: 808988},
 {CountryCode: 90, MobileNo: 909699}}}

 Custs3 := Customer{CustomerName: "Raym", Contacts: []Contact{
 {CountryCode: 75, MobileNo: 798088},
 {CountryCode: 75, MobileNo: 965755}}}

 Custs4 := Customer{CustomerName: "Stoke", Contacts: []Contact{
 {CountryCode: 80, MobileNo: 805510},
 {CountryCode: 80, MobileNo: 758863}}}

 db.Create(&Custs1)
 db.Create(&Custs2)
 db.Create(&Custs3)
 db.Create(&Custs4)

 customers := &Customer{}
 contacts := &Contact{}

 db.Debug().Where("customer_name=?","Martin").Preload("Contacts").Find(&customers)
 //db.Debug().Where("customer_name=?","John").Preload("Contacts").Find(&customers)
 fmt.Println("Customers", customers)
 fmt.Println("Contacts", contacts)

 //Update
 db.Debug().Model(&Contact{}).Where("cust_id=?", 3).Update("country_code", 77)
 //Delete
 db.Debug().Where("customer_name=?", customers.CustomerName).Delete(&customers)
 fmt.Println("After Delete", customers)

}
```

```
Console
<terminated> <exit code: 0> godemo - 26_GO_GORM_One2Many_Relationship [Go Appl
Customers &{2 Martin [{3 90 808988 2} {4 90 909699 2}]}
Contacts &{0 0 0 0}
Before Delete &{2 Martin [{3 90 808988 2} {4 90 909699 2}]}
```

Many To Many Relationship

User belongs to many languages and 'user_languages' will be a join table.

```
main.go
package main
import (
 _ "database/sql"
 "fmt"
 _ "github.com/go-sql-driver/mysql"
 "github.com/jinzhu/gorm"
)

type UserL struct {
 ID int `gorm:"primary_key"`
 Uname string
 Languages []Language `gorm:"many2many:user_languages";"ForeignKey:UserId"`
```

```
 //Based on this 3rd table user_languages will be created
}

type Language struct {
 ID int `gorm:"primary_key"`
 Name string
}

type UserLanguages struct {
 UserLId int
 LanguageId int
}

func main() {
 db, _ := gorm.Open("mysql", "root:root@tcp(127.0.0.1:3306)/testmapping?
charset=utf8&parseTime=True")
 defer db.Close()
 db.DropTableIfExists(&UserLanguages{}, &Language{}, &UserL{})
 db.AutoMigrate(&UserL{}, &Language{}, &UserLanguages{})

 //All foreign keys need to define here
 db.Model(UserLanguages{}).AddForeignKey("user_l_id", "user_ls(id)", "CASCADE", "CASCADE")
 db.Model(UserLanguages{}).AddForeignKey("language_id", "languages(id)", "CASCADE", "CASCADE")

 langs := []Language{{Name: "English"}, {Name: "French"}}
 //log.Println(langs)

 user1 := UserL{Uname: "John", Languages: langs}
 user2 := UserL{Uname: "Martin", Languages: langs}
 user3 := UserL{Uname: "Ray", Languages: langs}
 db.Save(&user1) //save is happening
 db.Save(&user2)
 db.Save(&user3)

 fmt.Println("After Saving Records")
 fmt.Println("User1", &user1)
 fmt.Println("User2", &user2)
 fmt.Println("User3", &user3)

 //Fetching
 user := &UserL{}
 db.Debug().Where("uname=?", "Ray").Find(&user)
 err := db.Debug().Model(&user).Association("Languages").Find(&user.Languages).Error
 fmt.Println("User is now coming", user, err)

 //Deletion
 fmt.Println(user, "to delete")
 db.Debug().Where("uname=?", "John").Delete(&user)

 //Updation
 db.Debug().Model(&UserL{}).Where("uname=?", "Ray").Update("uname", "Martin")

}
```

```
Console
<terminated> <exit code: 0> godemo - 26_GO_GORM_Many2Many_Relationship [G
After Saving Records
User1 &{1 John [{1 English} {2 French}]}
User2 &{2 Martin [{1 English} {2 French}]}
User3 &{3 Ray [{1 English} {2 French}]}
User is now coming &{3 Ray [{1 English} {2 French}]} <nil>
&{3 Ray [{1 English} {2 French}]} to delete
```

Here, you will find the complete documentation about gorm http://jinzhu.me/gorm/

SHOWING 2 COMMENTS

Vinodh Kumar *March 8th, 2019 01:00 PM*

I seen your tutorial. it's good. I have some doubts. what is the associations? means creation of tables, or saving the data or writing the queries like joins, inner joins or left joins. I confused please help me.

Mevlana Ayas *June 25th, 2019 08:34 AM*

thanks for a very clear tutorial

LEAVE A COMMENT

LEAVE YOUR COMMENT

FULL NAME | EMAIL ADDRESS | WEBSITE

Save my name, email, and website in this browser for the next time I comment.

POST COMMENT

GET IN TOUCH

FULL NAME

EMAIL

SHORT MESSAGE

SUBMIT