GO-08 04: Использование GORM для работы с БД

Описание:

В прошлых заданиях этого модуля мы использовали для работы с БД нативные sql-запросы. И библиотеку-драйвер pgx, которая позволяет их выполнять и обрабатывать ответ. Такой подход очень распространен и, можно сказать, является основным для работы с БД на Go. Он максимально гибкий и производительный. Но бывают случаи, когда код для работы с БД становится громоздким и не всегда удобным. Нам приходится руками писать однотипные sql-запросы, вычитывать строки и каждую колонку в переменные, закрывать объект rows для избежания утечек соединений, использовать подготовленные запросы и т.д. Если представить, что наше приложение должно работать с большим количеством сущностей, то такой подход становится не самым оптимальным. Например, для получения из базы одной сущности с разными наборами параметров нам приходится писать отдельные sql-запросы. А если сущность имеет большое количество полей, то получение ее из базы выглядит совсем уж рутиной:

```
q := `INSERT INTO public.client_app ( ...
"work_actual_addr_countryCode", "work_actual_addr_countryName",
"work_actual_addr_regionName", "work_actual_addr_regionSocr" ...)
  VALUES($1, $2, $3, $4, $5, $6, $7, $8, $9, $10, $11, $12, $13, $14,
$15, $16, $17, $18, $19, $20, $21, $22, $23, $24, $25, $26, $27, $28,
$29, $30, $31, $32, $33, $34, $35, $36, $37, $38, $39, $40, $41, $42,
$43, $44, $45, $46, $47, $48, $49, $50, $51, $52, $53, $54, $55, $56,
$57, $58, $59, $60, $61, $62, $63, $64, $65, $66, $67, $68, $69, $70,
$71, $72, $73, $74, $75, $76, $77, $78, $79, $80, $81, $82, $83, $84,
$85, $86, $87, $88, $89, $90, $91, $92, $93, $94, $95, $96, $97, $98,
$99, $100, $101, $102, $103, $104, $105, $106, $107, $108, $109, $110,
$111, $112, $113, $114, $115, $116, $117, $118, $119, $120, $121,
$122, $123, $124, $125, $126, $127, $128, $129, $130, $131, $132,
$133, $134, $135, $136, $137, $138, $139, $140, $141, $142, $143,
$144, $145, $146, $147, $148, $149, $150, $151, $152, $153, $154,
$155, $156, $157, $158, $159, $160, $161, $162, $163, $164, $165,
$166, $167, $168, $169, $170, $171, $172, $173, $174, $175, $176,
$177, $178, $179, $180, $181, $182, $183) RETURNING id;`
dbQ, err := c.Db.Prepare(q)

//Тут будут 183 строки с переменными
```

```
args = []interface{}{
      ...
      app.WorkCountryCode,
app.WorkCountry,
app.WorkRegion,
app.WorkRegionType,
app.WorkArea,
app.WorkAreaType,
...

}
err = dbQ.Query(ctx, args...).Scan(&id)
```

И такой код, в целом, трудно поддерживать. Как раз для решения подобных сложностей используются ORM. (Для тех, кто еще не знает, что такое ORM, можно ознакомиться с этим в статье). Для Go написано несколько неплохих ORM, и одна из популярных - это GORM, которую мы и рассмотрим в этом задании.
Сначала создадим отдельную базу данных, в которой будут храниться все наши ORM сущности.

```
$ cd ~/rebrain/
$ docker-compose exec pgdb psql -U db_user -c 'CREATE DATABASE gorm'
```

Дальше создадим новый проект module08_04, проинициализируем gomod и в cmd/app/main.go сконфигурируем подключение к БД для GORM.

```
//main.go
package main

import (
  "gorm.io/driver/postgres"
  "gorm.io/gorm"
  "module08_04/internal/models"
)

func main() {
  dsn := "host=localhost user=db_user password=pwd123 dbname=gorm
port=54320 sslmode=disable"
  db, err := gorm.Open(postgres.Open(dsn), &gorm.Config{})
  if err != nil {
     panic(err)
  }
```

```
  err = models.InitModels(db)
  if err != nil {
     panic(err)
  }
}
```

Тут подключаемся к БД postgres и вызываем функцию models.InitModels, в которой инициализируем сущности. Дальше создаем пакет для сущностей internal/models/models.go.

```
//файл models.go
package models

import "gorm.io/gorm"

type User struct {
  ID       int
  Name     string
  Age      int
  IsVerify bool
}


func InitModels (db *gorm.DB) error {
  err := db.AutoMigrate(&User{})
  if err != nil {
     return err
  }

  return nil
}
```

В пакете models описываем первую структуру User. А дальше передаем ссылку на структуру User в функцию AutoMigrate. На этом наша сущность создана, и GORM при запуске автоматически создаст в БД таблицу с нужными полями для сущности User. Запустим приложение.

```
2020/09/22 16:03:46
/Users/dhnikolas/mygo/module08_04/vendor/gorm.io/driver/postgres/migra
tor.go:108
```

```
[101.283ms] [rows:0] SELECT count(*) FROM information_schema.tables
WHERE table_schema =  CURRENT_SCHEMA() AND table_name = 'users' AND
table_type = 'BASE TABLE'

2020/09/22 16:03:46
/Users/dhnikolas/mygo/module08_04/internal/models/models.go:14
[126.486ms] [rows:0] CREATE TABLE "users" ("id" bigserial,"name"
text,"age" bigint,"is_verify" boolean,PRIMARY KEY ("id"))

                            Table "public.users"
  Column   | Type  | Collation | Nullable |            Default
-----------+---------+-----------+----------+-----------------------
----------
 id        | bigint  |           | not null |
nextval('users_id_seq'::regclass)
 name      | text    |           |          |
 age       | bigint  |           |          |
 is_verify | boolean |           |          |
Indexes:
    "users_pkey" PRIMARY KEY, btree (id)
```

В выводе видим, что GORM создал таблицу, исходя из заданной структуры. Это очень удобно и просто, когда даже не нужно писать sql-запросы. При этом имена колонок - это названия полей структуры, которые написаны через underscore. Также объявлять сущности можно с использованием тегов. Тегами можно конфигурировать поведение ORM. Например, нам нужно, чтобы поле IsVerify в таблице БД называлось trusted, а не is_verify. Для этого нужно указать специальный тег.

```
type User struct {
  ID         int
  Name    string
  Age      int
  IsVerify bool `gorm:"column:trusted"`
}
```

Или если мы вовсе не хотим, чтоб поле IsVerify сохранялось в БД, то нужно в теге указать -, например, так - IsVerify bool `gorm:"-"` . Также через теги можно установить тип колонки, ее свойства и многое другое. Все теги можно посмотреть в документации к библиотеке. Теперь давайте создадим первого пользователя. Создадим файл internal/app/app.go.

```
//файл app.go
package app
```

```
import (
  "gorm.io/gorm"
  "module08_04/internal/models"
)

func Start(db *gorm.DB) {
  create(db)
}

func create (db *gorm.DB){
  u := &models.User{
     Name: "Nikolai",
     Age: 30,
     IsVerify: true,
  }
  result := db.Create(u)
  if result.Error != nil {
     panic(result.Error)
  }
fmt.Printf("User created with ID: %d", u.ID)

}
```

В функции main добавим вызов app.Start(db) и запустим программу.
User created with ID: 1
Пользователь успешно создан. И в поле ID структуры User GORM сразу присвоил значение. Также, если передать в функцию Create []User, то будут созданы несколько пользователей. Они будут созданы батчами, это так называемое пакетное добавление.
Теперь давайте получим пользователя из базы и изменим его. В пакете app добавим и вызовем функцию selectAndUpdate.

```
//файл app.go
package app

import (
  "fmt"
  "gorm.io/gorm"
  "module08_04/internal/models"
)
```

```
func Start(db *gorm.DB) {
  //create(db)
  selectAndUpdate(db)
}

func selectAndUpdate(db *gorm.DB) {
  user := &models.User{}
  db.Where("name = ?", "Nikolai").First(&user)
  if user.ID > 0 {
     user.Name = "Dimitri"
     db.Save(user)
  }
}
...
```

Тут мы получили пользователя Nikolai, изменили имя в поле структуры и сохранили его снова. Эти же действия можно было сделать и другим способом.
`db.Model(&models.User{}).Where("name = ?", “Nikolai”).Update("name", "Dimitri")`

То есть, необязательно получать сущность в структуру, чтобы ее изменить. Достаточно указать, для какой сущности (таблицы) производятся действия и задать условия Where. А затем обновить все, что попало в выборку.
Также в качестве поля сущности может быть другая сущность, то есть связь между таблицами в БД. Например, у User может быть связь с сущностью Card.

```
//файл models.go
package models

import "gorm.io/gorm"

type User struct {
  ID       int
  Name     string
  Age      int
  IsVerify bool
  Cards    []Card `gorm:"foreignKey:UserID"`
}
```

```
type Card struct {
  ID int
  Number string
  Type string
  UserID int
}

func InitModels (db *gorm.DB) error {
  err := db.AutoMigrate(&User{}, &Card{})
  if err != nil {
     return err
  }

  return nil
}
```

Создали новую структуру Card с тремя полями.
У пользователя может быть несколько карт, поэтому поле Cards - это слайс структур Card.
В тегах указываем, что связь происходит по внешнему ключу - UserID карты.

```
Cards    []Card `gorm:"foreignKey:UserID"`
```

В функцию AutoMigrate добавим еще одну структуру, чтобы GORM создал для нее таблицу в БД. Теперь при создании пользователя мы можем создать карту и связь пользователя с картой.

```
u := &models.User{
  Name: "Nikolai",
  Age: 30,
  IsVerify: true,
  Cards: []models.Card{models.Card{
     Number: "34534534534",
     Type:   "VISA",
  }},
}
result := db.Create(u)
if result.Error != nil {
  panic(result.Error)
}
```

GORM - достаточно мощный инструмент, у которого много возможностей. Можно создавать любые типы связей и индексов на уровне структур, создавать хуки, работать с

несколькими репликами базы, производить миграции, логировать запросы и ответы, делать транзакции, использовать нативные sql-запросы и другое. Но в то же время вы получаете непростой инструмент со своей спецификой и документацией, которую необходимо внимательно изучить перед тем, как использовать GORM в боевых приложениях. Если ваше приложение имеет большое количество сущностей и эти сущности имеют большое количество полей, то GORM может облегчить и ускорить работу с базой.

Полезные ссылки:

- ORM или как забыть о проектировании БД
- GORM Guides

Задание

1. Форкните репозиторий module08_04 с кодом данного задания - в группу с вашими репозиториями - golang_users_repos/<your_gitlab_id>.
2. Создайте у себя в проекте module08 из ветки master ветку 04_task.
3. Используя gorm, создайте 30 пользователей со случайными именами и возрастом 18-70 лет. Также у каждого пользователя должна быть от 1 до 3 карт со случайным типом (VISA, MASTERCARD, MIR) и случайным номером.
4. Используя gorm, сделайте выборку пользователей, у которых больше двух карт.
5. Используя gorm, сделайте выборку пользователей, у которых есть карта VISA.
6. Используя gorm, сделайте выборку карт MIR, пользователи которых старше 50ти лет.
7. В ответе пришлите ссылку на merge request в ветку master своего проекта ветки 04_task.