GO-08 02: Работа с БД

Описание:

В прошлом задании мы достаточно подробно рассмотрели установку соединения с базой данных. И научились делать это с помощью пула соединений. Теперь можно приступить к выполнению CRUD запросов и рассмотреть некоторые особенности при работе с БД в Go. Для выполнения этого задания предлагаю использовать проект godb, который мы создавали в прошлом задании.

```
~/mygo
    └── godb
        └── cmd
            └── app
                └── main.go
```

Организуем работу более структурированно: общий код для работы с БД вынесем в директорию helpers, сами обращения к базе будем выполнять из пакета в директории internal, а конфигурировать подключение - в файле main.go.
Начнем с создания пакета-обертки для pgxpool для нашего удобства.

```
~/mygo
    └── godb
        └── cmd
            └── app
                └── main.go
        └── pkg
            └── helpers
                └── pg
                    └── pg.go
```

```go
//файл pg.go

package pg

import (
  "context"
  "fmt"
  "github.com/jackc/pgx/v4/pgxpool"
```

```
  "net/url"
)

//Структура конфига, которая включает в себя необходимые нам настройки соединения (сюда можно добавить любые другие поля для postgres типа ssl и т.д.)
type Config struct {
  Host     string
  Port     string
  Username string
  Password string
  DbName   string
  Timeout  int
}

//Функция для более удобного создания строки подключения к БД
func NewPoolConfig(cfg *Config) (*pgxpool.Config, error) {
  connStr :=
fmt.Sprintf("%s://%s:%s@%s:%s/%s?sslmode=disable&connect_timeout=%d",
     "postgres",
     url.QueryEscape(cfg.Username),
     url.QueryEscape(cfg.Password),
     cfg.Host,
     cfg.Port,
     cfg.DbName,
     cfg.Timeout)

  poolConfig, err := pgxpool.ParseConfig(connStr)
  if err != nil {
     return nil, err
  }

  return poolConfig, nil
}

//Функция-обертка для создания подключения с помощью пула
func NewConnection(poolConfig *pgxpool.Config) (*pgxpool.Pool, error)
{
  conn, err := pgxpool.ConnectConfig(context.Background(), poolConfig)
  if err != nil {
```

```
    return nil, err
  }

  return conn, nil
}
```

Пакет pg нам нужен для более удобного и красивого создания пула подключений к базе. Есть функция NewPoolConfig, которая возвращает конфиг для пула. В нем мы сможем конфигурировать более детально сам пул. И есть функция NewConnection, которая принимает конфиг и создает соединения.
Дальше, используя новый пакет pg, создадим пул соединений в главном файле main.go.

```
~/mygo
   └── godb
       └── cmd
           └── app
               └── main.go
       └── pkg
           └── helpers
               └── pg
                   └── pg.go
```

```
//файл main.go

package main

import (
  "context"
  "fmt"
  "godb/internal/godb"
  "godb/pkg/helpers/pg"
  "os"
)

func main()  {

  //Задаем параметры для подключения к БД (в прошлом задании мы поднимали контейнер с этими credentials)
  cfg := &pg.Config{}
```

```
  cfg.Host = "localhost"
  cfg.Username = "db_user"
  cfg.Password = "pwd123"
  cfg.Port = "54320"
  cfg.DbName = "db_test"
  cfg.Timeout = 5

  //Создаем конфиг для пула
  poolConfig, err := pg.NewPoolConfig(cfg)
  if err != nil {
     fmt.Fprintf(os.Stderr, "Pool config error: %v\n", err)
     os.Exit(1)
  }

  //Устанавливаем максимальное количество соединений, которые могут
находиться в ожидании
  poolConfig.MaxConns = 5


  //Создаем пул подключений
  c, err := pg.NewConnection(poolConfig)
  if err != nil {
     fmt.Fprintf(os.Stderr, "Connect to database failed: %v\n", err)
     os.Exit(1)
  }
  fmt.Println("Connection OK!")

  //Проверяем подключение
  _, err = c.Exec(context.Background(), ";");
  if err != nil {
     fmt.Fprintf(os.Stderr, "Ping failed: %v\n", err)
     os.Exit(1)
  }
  fmt.Println("Ping OK!")



  ins := &godb.Instance{Db: c}
  ins.Start()
```

```
}
```

Тут, используя структуру Config из пакета pg, мы задаем основные параметры для подключения к БД (хост, порт и т.д.). Дальше получаем структуру конфига именно для пула. И задаем максимальное количество соединений в ожидании (idle) - 5.

```
poolConfig.MaxConns = 5
```

С помощью функции NewConnection получаем структуру с с пулом подключений. Проверяем работоспособность с помощью Ping и передаем ее дальше в структуру Instance пакета godb, который, собственно, сейчас и создадим.

```
~/mygo
   └── godb
       └── cmd
           └── app
               └── main.go
       └── pkg
           └── helpers
               └── pg
                   └── pg.go
       └── internal
           └── godb
               └── godb.go
```

```
//файл godb.go

package godb

import (
  "fmt"
  "github.com/jackc/pgx/v4/pgxpool"
)

type Instance struct {
  Db *pgxpool.Pool
}

func (i *Instance) Start()  {
  fmt.Println("Project godb started!")
  //тут будем писать код для запросов в БД
}
```

Пакет godb содержит основную структуру Instance, которая принимает в зависимость подключение.

```
type Instance struct {
  Db *pgxpool.Pool
}
```

Благодаря этому, мы имеем возможность делать запросы в базу прямо из пакета godb.
На данный момент у нас получилось достаточно неплохое разделение ответственности между пакетами. Для упрощения работы с pgxpool мы вынесли переиспользуемый код в директорию helpers, пакет pg. Все, что касается конфигурации подключения, мы оставили в main.go (так как в пакете main должен содержаться только служебный код и никакой бизнес-логики), а сконфигурированное подключение передали в пакет godb.

```
  ins := &godb.Instance{Db: c}
  ins.Start()
```

Этот пакет является внутренним для нашего проекта и содержит его бизнес-логику.
Дальше вызвали функцию Start для запуска приложения.
Давайте запустим сборку и убедимся в том, что мы все сделали правильно и все работает.

```
$ go run cmd/godb/main.go
Connection OK!
Ping OK!
Project godb started!
```

Отлично! Видим, подключение успешно установлено и запущен проект godb!
Теперь создадим руками таблицу users в нашей БД db_test (дальше изменения в структуру базы мы будем делать только через миграции).

```
$ docker-compose exec pgdb psql  db_test -U db_user  -c 'CREATE TABLE
users (id SERIAL NOT NULL, created_at TIMESTAMP, name VARCHAR(100),
age INTEGER, verify BOOLEAN);'

CREATE TABLE
```

Это таблица с теми типами, с которыми чаще всего приходится работать.

```
                                        Table "public.users"
   Column   |            Type                  | Collation      |
Nullable |             Default
```

```
------------+-----------------------------+-----------+----------+----
-------------------------------
 id                | integer                     |                | not
null | nextval('users_id_seq'::regclass)
 created_at        | timestamp without time zone |                |
|
 name              | character varying(100)      |                |
|
 age               | integer                     |                |
|
 verify            | boolean                     |                |
|
```

Давайте добавим в эту таблицу первого user. Создадим в пакете godb для структуры Instance функцию addUser и в ней будем добавлять запись в базу. В psql это был бы следующий запрос - INSERT INTO users (created_at, name, age, verify) VALUES ('2020-07-21 15:20:00', 'Vladimir', 27, true); . А в Go с использованием библиотеки github.com/jackc/pgx/ он будет выглядеть так

```
func (i *Instance) addUser(ctx context.Context, name string, age int,
isVerify bool) {
  commandTag, err := i.Db.Exec(ctx, "INSERT INTO users (created_at,
name, age, verify) VALUES ($1, $2, $3, $4)", time.Now(), name, age,
isVerify)
  if err != nil {
     fmt.Println(err)
     return
  }

  fmt.Println(commandTag.String())
  fmt.Println(commandTag.RowsAffected())
}
```

Мы создали функцию addUser для структуры Instance. В ней получаем ссылку на pool соединений i.Db и вызываем функцию Exec. В нее передается структура контекста, строка sql запроса с переменными в виде $1, $2, $3, $4 и далее аргументы для этих переменных в соответствующей последовательности. Все функции запросов в этой библиотеке поддерживают подготовленный sql. И для безопасности все значения в sql строку нужно

передавать через подготовленные переменные. Функция Exec возвращает CommandTag и ошибку. CommandTag - это структура для работы с выводом postgres-а после исполнения запроса (INSERT 0 1). Можно, например, получить количество строк, затронутых запросом и т.д. Функция Exec не возвращает записи из базы, поэтому ее необходимо использовать для всех запросов, кроме SELECT.
Давайте теперь вызовем функцию addUser и передадим на вход данные пользователя.

```
func (i *Instance) Start()  {
 fmt.Println("Project godb started!")

 i.addUser(context.Background(), "Vladimir", 27, false)

}
```

В выводе fmt.Println(ct.String()) видим нативный ответ postgres при insert. А в fmt.Println(ct.RowsAffected()) возвращается int64 - количество затронутых строк. То есть, мы успешно добавили запись в нашу таблицу users.

```
$ go run cmd/godb/main.go
Connection OK!
Ping OK!
Project godb started!
INSERT 0 1
1
```

Отлично! Только давайте добавим еще одного пользователя в базу. Изменим в вызове функции addUser имя на Dmitri, а возраст - на 25. И запустим нашу программу еще раз. Это нам понадобится для рассмотрения SELECT запросов.
Для SELECT запросов используются 2 функции - Query и QueryRow.
Query возвращает структуру pgx.Rows и ошибку. Пробегаясь по структуре pgx.Rows циклом, мы вычитываем функцией Scan строки, вернувшиеся из базы данных. Вычитывать данные из базы нужно в заранее определенные поля или переменные. То есть, вначале мы выделяем память под соответствующий тип, а затем присваиваем туда значения, которые мы получили из базы. Также в sql запросе необходимо задать конкретные колонки, из которых будет производиться выборка. И в этой же последовательности произойдет чтение вернувшихся строк. Давайте реализуем выборку с помощью Query.
Создадим структуру пользователя в пакете godb:

```
type User struct {
  Name string
```

```
  Age int
  IsVerify bool
}
```

Добавим функцию getAllUsers, которую рассмотрим подробно:

```
func (i *Instance) getAllUsers(ctx context.Context) {
  var users []User

  rows, err := i.Db.Query(ctx, "SELECT name, age, verify FROM users;")
  if err == pgx.ErrNoRows {
      fmt.Println("No rows")
      return
  } else if err != nil {
      fmt.Println(err)
      return
  }
  defer rows.Close()

  for rows.Next() {
     user := User{}
     rows.Scan(&user.Name, &user.Age, &user.IsVerify)
     users = append(users, user)
  }

  fmt.Println(users)
}
```

Данные из БД будем присваивать в структуру User. Для этого мы создадим переменную с типом слайс User. Для запроса в данном случае мы используем функцию Query, она вернет нам все строки из базы, соответствующие sql запросу или, если по запросу не найдено строк, вернет ошибку pgx.ErrNoRows. Query возвращает структуру Rows, которую в цикле мы обрабатываем с помощью метода Next. Этот метод вернет true в случае наличия следующей строки. При каждой итерации создаем структуру User. Далее вычитываем методом Scan значения в поле структуры rows.Scan(&user.Name, &user.Age, &user.IsVerify). Аргументы в метод Scan должны передаваться в том же порядке, в котором возвращаются колонки таблицы в запросе.
Обратите внимание, что поля или переменные должны передаваться как указатель.
После этого добавляем структуру в []User .

```
users = append(users, user)
```

Обязательно нужно закрывать объект Rows defer rows.Close() после завершения вычитывания строк. Это необходимо для того, чтобы высвободить подключение к БД и избежать утечек. Никогда не забывайте об этом. Иначе с каждым запросом пул подключений будет нарастать, и вы упретесь в лимит доступных коннектов, а затем залипните на ожидании подключения, которое никогда не освободится. В результате, ваше приложение зависнет.
Теперь давайте закомментируем в функции Start пакета godb вызов addUser и добавим вызов getAllUsers

```
func (i *Instance) Start()  {
  fmt.Println("Project godb started!")

  //i.addUser(context.Background(), "Dmitri", 25, false)
  i.getAllUsers(context.Background())
}
```

Запустим программу:

```
Connection OK!
Ping OK!
Project godb started!
[{Vladimir 27 false} {Dmitri 25 false}]
```

Видим, что мы получили слайс и в нем 2 структуры User с данными пользователей из базы. Теперь для структуры Instance создадим 2 метода: метод с запросом на обновления возраста пользователя и метод с запросом пользователя по имени.

```
//...
func (i *Instance) updateUserAge(ctx context.Context, name string, age
int)  {
  _, err := i.Db.Exec(ctx, "UPDATE  users SET age=$1 WHERE name=$2;",
age, name)
  if err != nil {
     fmt.Println(err)
     return
  }
}

func (i *Instance) getUserByName(ctx context.Context, name string)  {
  //Выполнение самого запроса. И получение структуры rows, которая
содержит в себе строки из базы данных.
  user := &User{}
```

```
  //Тут используется метод QueryRow, который предполагает возвращение только одной строки из базы. В нашем случае это гарантирует LIMIT 1 в sql-запросе.
  //Если будет возвращено несколько строк из базы, QueryRaw обработает только одну.
  err := i.Db.QueryRow(ctx, "SELECT name, age, verify FROM users WHERE name=$1 LIMIT 1;", name).Scan(&user.Name, &user.Age, &user.IsVerify)
  if err != nil {
     fmt.Println(err)
     return
  }
  fmt.Printf("User by name: %v\n", user)
}
//..
```

Вызовем новые методы в функции Start:

```
func (i *Instance) Start()  {
  fmt.Println("Project godb started!")

  //i.addUser(context.Background(), "Dmitri", 25, false)
  i.getAllUsers(context.Background())
  i.updateUserAge(context.Background(), "Dmitri", 26)
  i.getUserByName(context.Background(), "Dmitri")
}
```

Запустим программу и получим вывод. Возраст пользователя Dmitri изменился.

```
Connection OK!
Ping OK!
Project godb started!
[{Vladimir 27 false} {Dmitri 25 false}]
User by name: &{Dmitri 26 false}
```

Задание

1. Форкните репозиторий module08_02 с кодом данного задания в группу с вашими репозиториями - golang_users_repos/<your_gitlab_id>.
2. Создайте у себя в проекте module08_02 из ветки master ветку 02_task.

3. Используя пакет godb, добавьте в таблицу users 100 пользователей со случайными именами, возрастом и параметром isVerify со случайным значением true или false.
4. Затем каждому подтвержденному пользователю (isVerify=true) добавьте фамилию в колонку name (Например, было Dmitri, стало Dmitri Ivanov).
5. Удалите из базы пользователей, у которых нет фамилии.
6. Для каждого запроса в базу должен быть отдельный метод в структуре Instance.
7. В ответе пришлите ссылку на merge request в ветку master своего проекта ветки 02_task.