GO-08 03: Миграции БД

Описание:

В этом задании мы рассмотрим, как делать миграции в Go. Но сперва давайте вспомним, для чего вообще нужны миграции БД.
Миграции необходимы для версионирования состояния БД, для поддержки соответствия версии приложения и версии состояния базы данных. Например, если в приложении добавилось еще одно поле сущности, а в базе данных такой колонки в таблице еще нет, то необходимо создать миграцию с определенной версией, которая создаст нужную колонку в базе данных. В этом разделе мы рассмотрим автоматическое применение миграций из кода самого приложения. То есть приложение само будет знать, какую миграцию необходимо применить в момент запуска. В этом нам поможет отличная библиотека Goose. Ее преимущество в том, что миграции могут быть написаны как в sql-файле, так и в go-файле. В последнем случае goose позволяет добавлять в миграции изменения не только структуры БД, но и ее данных с помощью логики, написанной на Go. То есть, произвести любое действие в БД в качестве миграции. Что на практике очень удобно.
Например, в нашем приложении из предыдущего задания нам в какой-то момент понадобилась информация о том, когда последний раз менялись данные пользователя. Эту дату мы можем хранить в таблице, в колонке updated_at. Но изначально в таблице такой колонки не было. Поэтому, когда логика приложения будет предусматривать эту колонку, нам необходимо добавить соответствующую миграцию. Давайте сделаем это.
Начнем с инициализации библиотеки goose. Для нее на вход необходимо передать подключения из стандартного драйвера database/sql. Делается это следующим образом. В файле main.go после инициализации основного подключения через pgxpool добавим следующий код:

```
//файл main.go
import (
//...
  "database/sql"
  _ "github.com/lib/pq"
  "github.com/pressly/goose"
//...
)

//...
mdb, _ := sql.Open("postgres", poolConfig.ConnString())
err = mdb.Ping()
if err != nil {
  panic(err)
```

```
}
err = goose.Up(mdb, "/var")
if err != nil {
  panic(err)
}
```

Тут происходит подключение к нашей БД через стандартный драйвер database/sql. Так как мы используем postgres, нам необходимо импортировать стандартную библиотеку для работы с ним _ "github.com/lib/pq". Затем через sql.Open создаем подключение, используя уже имеющийся у нас конфиг poolConfig.ConnString(). И наконец вызываем goose.Up для миграции. Таким образом мы применим все невыполненные sql миграции, которые находятся в директории /var, а также все go миграции, которые определены в пакете migrations (об этом чуть дальше). Отлично. На данный момент наше приложение будет при каждом запуске накатывать новые миграции. Теперь осталось создать их.
В библиотеке github.com/pressly/goose есть утилита goose, которая позволяет с помощью cli команды создавать шаблоны миграций. Необходимо установить библиотеку через go get .

```
$ go get -u github.com/pressly/goose/cmd/goose
```

И через утилиту goose сгенерировать новый файл миграции. Можем сгенерировать как sql-файл, так и go. В нашем случае создадим go-файл. Файлы миграций расположим в директории internal/migrations.

```
$ mkdir internal/migrations
$ cd  internal/migrations
$ ~/go/bin/goose create add_updated_at go
2020/09/19 19:24:55 Created new file: 20200919192455_add_updated_at.go
```

Мы сгенерировали go файл миграции со следующим содержимым:

```
package migrations

import (
  "database/sql"
  "github.com/pressly/goose"
)

func init() {
  goose.AddMigration(upAddUpdatedAt, downAddUpdatedAt)
}

func upAddUpdatedAt(tx *sql.Tx) error {
```

```
  // This code is executed when the migration is applied.
  return nil
}

func downAddUpdatedAt(tx *sql.Tx) error {
  // This code is executed when the migration is rolled back.
  return nil
}
```

Тут все просто. Каждая миграция - это файл в пакете migrations. Нужно описать логику, которая выполнится в момент применения и отката миграции (upAddUpdatedAt, downAddUpdatedAt). На вход в функции передается объект транзакции, с помощью которого можно выполнять запросы в базу. Обратите внимание также на название файла, так как оно очень важно для версионирования и последовательности исполнения - 20200919192455_add_updated_at.go. Префикс файла содержит дату создания миграции. Исходя из этих дат, библиотека применяет миграции в правильной последовательности. И также эта дата является версией миграции. Каждый новый файл миграции - это новая версия состояния БД, между которыми можно переключаться.
Давайте через миграцию 20200919192455_add_updated_at.go создадим новую колонку в БД updated_at с временем последнего обновления пользователя.

```
package migrations

import (
  "database/sql"
  "github.com/pressly/goose"
)

func init() {
  goose.AddMigration(upAddUpdatedAt, downAddUpdatedAt)
}

func upAddUpdatedAt(tx *sql.Tx) error {
  _, err := tx.Exec(`
        ALTER TABLE users ADD COLUMN IF NOT EXISTS updated_at
TIMESTAMP DEFAULT now();


    `)
  if err != nil {
    return err
  }
```

```
  return nil
}

func downAddUpdatedAt(tx *sql.Tx) error {
  _, err := tx.Exec(`
       ALTER TABLE users DROP COLUMN IF EXISTS updated_at;
    `)
  if err != nil {
     return err
  }

  return nil
}
```

Миграция создана и теперь самое время подключить пакет миграций в main.go. Делается это очень просто. Нужно импортировать через нижнюю черту пакет migrations.

```
import (
//...
  _ "godb/internal/migrations"
//...
)
```

Теперь при старте приложение goose применит все ранее не выполненные миграции. Давайте запустим приложение.

```
Connection OK!
2020/09/19 20:10:51 OK    20200919192455_add_updated_at.go
2020/09/19 20:10:51 goose: no migrations to run. current version:
20200919192455
Ping OK!
```

Goose сообщает нам, что он накатил одну миграцию и показал нам текущую версию БД. Теперь в таблице users появилась колонка updated_at. Также в базе db_test появилась новая таблица goose_db_version, которую создал goose для отслеживания версий БД.

```
db_test=# \dt
              List of relations
 Schema |       Name       | Type |  Owner
--------+------------------+------+---------
 public | goose_db_version | table | db_user
```

```
 public | users            | table | db_user
(2 rows)
```

Теперь каждый раз, когда мы добавим новую миграцию, наше приложение будет выполнять ее при старте. И тем самым мы добились контроля над состоянием БД из кода приложения. Для изменения БД нам всего-навсего необходимо создать файл миграции через команду goose create, в нем описать саму миграцию и перезапустить приложение.
Мы рассмотрели миграции на go. Но у нас есть еще возможность использовать миграции в sql-файле. Для этого нужно создать миграцию с типом sql $ ~/go/bin/goose create some_migration sql и поместить ее в директорию, указанную в функции up.

```
//файл main.go

err = goose.Up(mdb, "/var")
if err != nil {
  panic(err)
}
```

А дальше все то же самое, что и с go миграциями. Goose отследит версию миграции по названию файла и выполнит ее.
Откат миграции происходит с помощью функции goose.Down или goose.DownTo. Первая функция откатит миграцию на одну назад. А во втором случае нужно передать версию, до которой необходимо откатиться. Откаты миграции в Go можно организовать разными способами. Например, собрать для приложения отдельный бинарник, при запуске которого будет откатываться одна версия назад с помощью функции goose.Down. Или собрать бинарник, при запуске которого в качестве аргумента передается версия, до которой нужно откатиться goose.DownTo.
Использовать библиотеку в Go приложении - это не единственный способ управлять миграциями. В Goose существуют cli команды, которые могут выполнить любые действия с миграциями. Более подробно об этом можно узнать в документации

Задание:

1. Форкните репозиторий module08_03 с кодом данного задания в группу с вашими репозиториями - golang_users_repos/<your_gitlab_id>.
2. Создайте у себя в проекте module08_03 из ветки master ветку 03_task.
3. Создайте в sql-файле миграцию, которая добавит в таблицу users колонку last_name varchar(100).
4. Создайте в go-файле в пакете migrations миграцию, которая разделит имя и фамилию по разным колонкам name и last_name. (После прошлого задания имя и фамилия у нас хранятся в одном столбце name).
5. При обновлении записей в базе необходимо обновлять время в колонке updated_at.

6. Миграции должны примениться автоматически при запуске приложения.
7. В ответе пришлите ссылку на merge request в ветку master своего проекта ветки 03_task.