Работа с БД в Go

Как проходит занятие

- Активно участвуем - задаем вопросы.
- Чат вижу - могу ответить не сразу.
- После занятия - оффтопик, ответы на любые вопросы.

План занятия

- Установка и работа с PostgreSQL
- Подключение к СУБД и настройка пула подключений
- Выполнение запросов и получение результатов
- Стандартные интерфейсы sql.DB, sql.Rows, sql.Tx
- Использование транзакций
- SQL инъекции и борьба с ними
- Тестирование

Работаем с PostgreSQL локально

Устанавливаем сервер из консоли (пример для Ubuntu):

```
# обновить пакеты
$ sudo apt-get update

# установить PostgreSQL сервер и клиент
$ sudo apt-get -y install postgresql

# запустить PostgreSQL
$ sudo systemctl start postgresql

# подключиться под пользователем, созданным по умолчанию
$ sudo -u postgres psql
```

https://www.postgresql.org/download/linux/ubuntu/

Работаем с PostgreSQL локально через Docker

Запускаем контейнер с сервером PostgreSQL:

```
docker run -d \
    --name pg \
    -e POSTGRES_PASSWORD=postgres \
    -e PGDATA=/var/lib/postgresql/data/pgdata \
    -v /Users/estepankevich/psqldata:/var/lib/postgresql/data \
    -p 5432:5432 \
    postgres
```

Ждём немного, пока СУБД поднимется

Подключаемся к серверу:

```
docker exec -it pg psql -Upostgres -dpostgres
```

https://hub.docker.com/_/postgres

Работаем с PostgreSQL локально

Работаем в клиенте СУБД:

```
postgres=# create database exampledb;

postgres=# create user anthony with encrypted password 'ozon_course';

postgres=# grant all privileges on database exampledb to ozon_course;
```

Удобный клиент с графическим интерфейсом: https://www.pgadmin.org/download/

Миграции: При чем здесь гусь?

Goose

Goose — менеджер миграции базы данных общего назначения. Идея проста:

https://github.com/pressly/goose

- Вы предоставляете файлы схемы SQL, которые следуют определенному соглашению об именах
- Вы предоставляете простой dbconf.ymlфайл, который говорит Goose, как подключиться к вашим различным базам данных
- Goose предоставляет вам простые инструменты для обновления схемы (goose up), проверки (goose status) и даже возврата (goose down).

Пример миграции

```
-- +goose Up
CREATE table books (
                    id              serial primary key,
                    title           text,
                    description     text,
                    meta            jsonb,
                    created_at      timestamptz not null default now(),
                    updated_at      timestamptz
);

INSERT INTO books (title, description, meta, updated_at)
VALUES
 ('Мастер и Маргарита', 'test description 1', '{}', now()),
 ('Граф Монте-Кристо', 'test description 2', null, null),
 ('Марсианин', 'test description 3', '{"author": "Энди Вейер"}', now());

-- +goose Down
drop table books;
```

Подключение к PostgreSQL из Go, простейший вариант

Создание подключения:

```
import "database/sql"
import _ "github.com/jackc/pgx/stdlib"

dsn := "..."
db, err := sql.Open("pgx", dsn)  // *sql.DB
if err != nil {
  log.Fatalf("failed to load driver: %v", err)
}
// создан пул соединений
```

Использование подключения:

```
err := db.PingContext(ctx)
if err != nil {
  return xerrors.Errorf("failed to connect to db: %v", err)
}
// работаем с db
```

http://go-database-sql.org/importing.html
http://go-database-sql.org/accessing.html

DataSourceName

DSN - строка подключения к базе, содержит все необходимые опции.
Синтаксис DSN зависит от используемой базы данных и драйвера.

Например для PostgreSQL:

```
"postgres://myuser:mypass@localhost:5432/mydb?sslmode=verify-full"
```

Или

```
"user=myuser dbname=mydb sslmode=verify-full password=mypass"
```

- `host` - Сервер базы данных или путь к UNIX-сокету (по-умолчанию localhost)
- `port` - Порт базы данных (по-умолчанию 5432)
- `dbname` - Имя базы данных
- `user` - Пользователь в СУБД (по умолчанию - пользователь OS)
- `password` - Пароль пользователя

Подробнее: https://godoc.org/github.com/lib/pq

Пул соединений

`sql.DB` - это пул соединений с базой данных. Соединения будут открываться по мере необходимости.

`sql.DB` - безопасен для конкурентного использования (так же как `http.Client`)

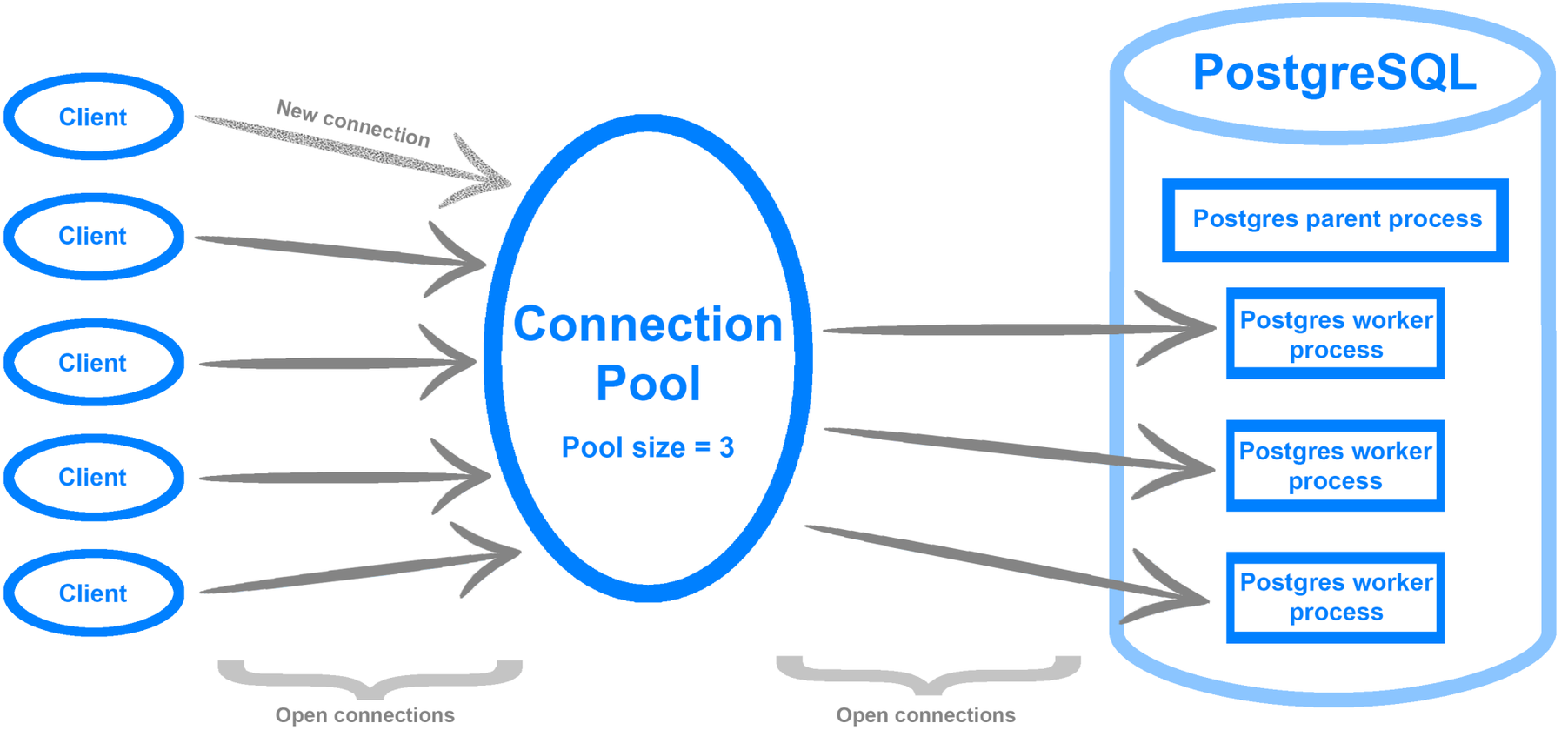
Настройки пула:

```
// Макс. число открытых соединений от этого процесса
db.SetMaxOpenConns(n int)

// Макс. число открытых неиспользуемых соединений
db.SetMaxIdleConns(n int)

// Макс. время жизни одного подключения
db.SetConnMaxLifetime(d time.Duration)
```

http://go-database-sql.org/connection-pool.html

Пул соединений. Какую проблему решает?

Пул соединений



With Connection Pool

Выполнение запросов

```
query := `insert into events(owner, title, descr, start_date, end_date)
  values($1, $2, $3, $4, $5)`

result, err := db.ExecContext(ctx, query,
  42, "new year", "watch the irony of fate", "2019-12-31", "2019-12-31"
) // sql.Result
if err != nil {
  // обработать ошибку
}

// Авто-генерируемый ID (SERIAL)
eventId, err := result.LastInsertId()      // int64

// Количество измененных строк
rowsAffected, err := result.RowsAffected() // int64
```

http://go-database-sql.org/retrieving.html

Получение результатов

```
query := `
  select id, title, descr
  from events
  where owner = $1 and start_date = $2
`
rows, err := db.QueryContext(ctx, query, owner, date)
if err != nil {
  // ошибка при выполнении запроса
}
defer rows.Close()

for rows.Next() {
  var id int64
  var title, descr string
  if err := rows.Scan(&id, &title, &descr); err != nil {
    // ошибка сканирования
  }
  // обрабатываем строку
  fmt.Printf("%d %s %s\n", id, title, descr)
}
if err := rows.Err(); err != nil {
    // ошибка при получении результатов
}
```

Объект sql.Rows

```
// возвращает имена колонок в выборке
rows.Columns() ([]string, error)

// возвращает типы колонок в выборке
rows.ColumnTypes() ([]*ColumnType, error)

// переходит к следующей строке или возвращает false
rows.Next() bool

// заполняет переменные из текущей строки
rows.Scan(dest ...interface{}) error

// закрывает объект Rows
rows.Close()

// возвращает ошибку, встреченную при итерации
rows.Err() error
```

Получение одной строки

```
query := "select * from events where id = $1"

row := db.QueryRowContext(ctx, query, id)

var id int64
var title, descr string

err := row.Scan(&id, &title, &descr)

if err == sql.ErrNoRows {
  // строки не найдено
} else if err != nil {
  // "настоящая" ошибка
}
```

PreparedStatements

PreparedStatement - это заранее разобранный запрос, который можно выполнять повторно. *PreparedStatement* - временный объект, который *создается в СУБД* и живет в рамках сессии, или пока не будет закрыт.

```
// создаем подготовленный запрос
stmt, err := db.Prepare("delete from events where id = $1") // *sql.Stmt
if err != nil {
    log.Fatal(err)
}

// освобождаем ресурсы в СУБД
defer stmt.Close()

// многократно выполняем запрос
for _, id := range ids {
  _, err := stmt.Exec(id)
  if err != nil {
    log.Fatal(err)
  }
}
```

http://go-database-sql.org/prepared.html

Работа с соединением

`*sql.DB` - это пул соединений. Даже последовательные запросы могут использовать *разные* соединения с базой.

Если нужно получить одно конкретное соединение, то

```
conn, err := db.Conn(ctx)  // *sql.Conn

// вернуть соединение в pool
defer conn.Close()

// далее - обычная работа как с *sql.DB
err := conn.ExecContext(ctx, query1, arg1, arg2)

rows, err := conn.QueryContext(ctx, query2, arg1, arg2)
```

Транзакции

Транзакция - группа запросов, которые либо выполняются, либо не выполняются вместе. Внутри транзакции все запросы видят "согласованное" состояние данных.

На уровне SQL для транзакций используются отдельные запросы: `BEGIN`, `COMMIT`, `ROLLBACK`.

```
tx, err := db.BeginTx(ctx, nil)  // *sql.Tx
if err != nil {
  log.Fatal(err)
}

// далее - обычная работа как с *sql.DB
err := tx.ExecContext(ctx, query1, arg1, arg2)
rows, err := tx.QueryContext(ctx, query2, arg1, arg2)

err := tx.Commit()  // или tx.Rollback()
if err != nil {
  // commit не прошел, данные не изменились
}

// далее объект tx не пригоден для использования
```

http://go-database-sql.org/modifying.html

Обертка над транзакциями

```go
// WithTxFunc custom type of func to helper
type WithTxFunc func(ctx context.Context, tx sqlutil.Tx) error

// WithTx runs code with transaction
func (d *tx) WithTx(ctx context.Context, fn WithTxFunc) error {
    t, err := d.db.Begin(ctx, nil)
    if err != nil {
        return errors.Wrap(err, "Tx.Begin")
    }
    if err = fn(ctx, t); err != nil {
        if errRollback := t.Rollback(); errRollback != nil {
            return errors.Wrap(err, "Tx.Rollback")
        }
        return errors.Wrap(err, "Tx.WithTxFunc")
    }
    if err = t.Commit(); err != nil {
        return errors.Wrap(err, "Tx.Commit")
    }
    return nil
}
```

Основные методы

Определены у `*sql.DB`, `*sql.Conn`, `*sql.Tx`, `*sql.Stmt`:

```
// изменение данных
ExecContext(ctx context.Context, query string, args ...interface{}) (Result, error)

// получение данных (select)
QueryContext(ctx context.Context, query string, args ...interface{}) (*Rows, error)

// получение одной строки
QueryRowContext(ctx context.Context, query string, args ...interface{}) *Row
```

Внимание, ошибка:

```
_, err := db.QueryContext(ctx, "delete from events where id = $1", 42)
```

NULL

В SQL базах любая колонка может быть объявлена к NULL / NOT NULL. NULL - это не 0 и не пустая строка, это отсутствие значения.

```
create table users (
  id           serial primary key,
  name         text not null,
  age          int null
);
```

Для обработки `NULL` в Go предлагается использовать специальные типы:

```
var id, realAge int64
var name string
var age sql.NullInt64
err := db.QueryRowContext(ctx, "select * from users where id = 1").Scan(&id, &name, &age)

if age.Valid {
  realAge = age.Int64
} else {
    // обработка на ваше усмотрение
}
```

SQL Injection

Опасно:

```
query := "select * from users where name = '" + name + "'"
query := fmt.Sprintf("select * from users where name = '%s'", name)
```

Потому что в `name` может оказаться что-то типа:

```
"jack'; truncate users; select 'pawned"
```

Правильный подход - использовать `placeholders` для подстановки значений в SQL:

```
row := db.QueryRowContext(ctx, "select * from users where name = $1", name)
```

Однако это не всегда возможно. Так работать не будет:

```
db.QueryRowContext(ctx, "select * from $1 where name = $2", table, name)
db.QueryRowContext(ctx, "select * from user order by $1 limit 3", order)
```

Проверить код на инъекции (и другие проблемы безопасности): https://github.com/securego/gosec

```
sql, args, err := sq.
    Insert("users").Columns("name", "age").
    Values("moe", 13).Values("larry", sq.Expr("? + 5", 12)).
    ToSql()

sql == "INSERT INTO users (name,age) VALUES (?,?),(?,? + 5)"
```

Проблемы database/sql

- placeholder зависят от базы: (`$1` в Postgres, `?` в MySQL, `:name` в Oracle)
- Есть только базовые типы, но нет, например `sql.NullDate`
- `rows.Scan(arg1, arg2, arg3)` - неудобен, нужно помнить порядок и типы колонок.
- Нет возможности `rows.StructScan(&event)`

Расширение sqlx

`jmoiron/sqlx` - обертка, прозрачно расширяющая стандартную библиотеку `database/sql`.

- `sqlx.DB` - обертка над `*sql.DB`
- `sqlx.Tx` - обертка над `*sql.Tx`
- `sqlx.Stmt` - обертка над `*sql.Stmt`
- `sqlx.NamedStmt` - PreparedStatement с поддержкой именованых параметров

Подключение `jmoiron/sqlx`:

```
import "github.com/jmoiron/sqlx"

db, err := sqlx.Open("pgx", dsn)  // *sqlx.DB

rows, err := db.QueryContext("select * from events") // *sqlx.Rows

...
```

sqlx: именованные placeholder'ы

Можно передавать параметры запроса в виде словаря:

```
sql := "select * from events where owner = :owner and start_date = :start"
rows, err := db.NamedQueryContext(ctx, sql, map[string]interface{}{
  "owner": 42,
  "start": "2019-12-31",
})
```

Или структуры:

```
type QueryArgs{
  Owner int64
  Start string
}
sql := "select * from events where owner = :owner and start_date = :start"
rows, err := db.NamedQueryContext(ctx, sql, QueryArgs{
  Owner: 42,
  Start: "2019-12-31",
})
```

sqlx: сканирование

Можно сканировать результаты в словарь:

```
sql := "select * from events where start_date > $1"

rows, err := db.QueryContext(ctx, sql, "2020-01-01") // *sqlx.Rows

for rows.Next() {
  results := make(map[string]interface{})
  err := rows.MapScan(results)
  if err != nil {
    log.Fatal(err)
  }
  // обрабатываем result
}
```

sqlx: сканирование

Можно сканировать результаты структуру:

```
type Event {
  Id          int64
  Title       string
  Description string   `db:"descr"`
}

sql := "select * from events where start_date > $1"

rows, err := db.NamedQueryContext(ctx, sql, "2020-01-01") // *sqlx.Rows

events := make([]Event)

for rows.Next() {
  var event Event
  err := rows.StructScan(&event)
  if err != nil {
    log.Fatal(err)
  }
  events = append(events, event)
}
```

Драйверы для Postgres

- Лучший драйвер на текущий момент: https://github.com/jackc/pgx
- Другой часто используемый драйвер (менее производительный): https://github.com/lib/pq

ORM

- https://gorm.io/ - использует пустые интерфейсы :(
- https://github.com/go-reform/reform - использует кодогенерацию, но разработка немного заброшена

Тестирование

https://github.com/DATA-DOG/go-txdb

```
func init() {
    txdb.Register("txdb", "mysql", "root@/txdb_test")
}

func main() {
    db, err := sql.Open("txdb", "identifier")
    if err != nil {
        log.Fatal(err)
    }
    defer db.Close()
    if _, err := db.Exec(`INSERT INTO users(username) VALUES("gopher")`); err != nil {
        log.Fatal(err)
    }
}
```

Моки

https://github.com/DATA-DOG/go-sqlmock

```
func TestShouldUpdateStats(t *testing.T) {
    db, mock, err := sqlmock.New()
    if err != nil {
        t.Fatalf("an error '%s' was not expected when opening a stub database connection", err)
    }
    defer db.Close()

    mock.ExpectBegin()
    mock.ExpectExec("INSERT INTO product_viewers").WithArgs(2, 3).WillReturnResult(sqlmock.NewResult(1,
    mock.ExpectCommit()

    if err = recordStats(db, 2, 3); err != nil {
        t.Errorf("error was not expected while updating stats: %s", err)
    }

    if err := mock.ExpectationsWereMet(); err != nil {
        t.Errorf("there were unfulfilled expectations: %s", err)
    }
}
```

Фикстуры

https://github.com/go-testfixtures/testfixtures

```yaml
# comments.yml
- id: 1
  post_id: 1
  content: A comment...
  author_name: John Doe
  author_email: john@doe.com
  created_at: 2020-12-31 23:59:59
  updated_at: 2020-12-31 23:59:59

- id: 2
  post_id: 2
  content: Another comment...
  author_name: John Doe
  author_email: john@doe.com
  created_at: 2020-12-31 23:59:59
  updated_at: 2020-12-31 23:59:59

# ...
```

Фикстуры

```
fixtures, err := testfixtures.New(
        testfixtures.Database(db),
        testfixtures.Dialect("postgres"),
        testfixtures.Paths(
                "fixtures/orders.yml",
                "fixtures/customers.yml",
                "common_fixtures/users"
        ),
)
if err != nil {
        ...
}
```

Другие ресурсы для изучения

- [ru] https://habr.com/ru/company/oleg-bunin/blog/461935/
- [en] http://go-database-sql.org/index.html
- [en] https://golang.org/pkg/database/sql
- [en] https://jmoiron.github.io/sqlx