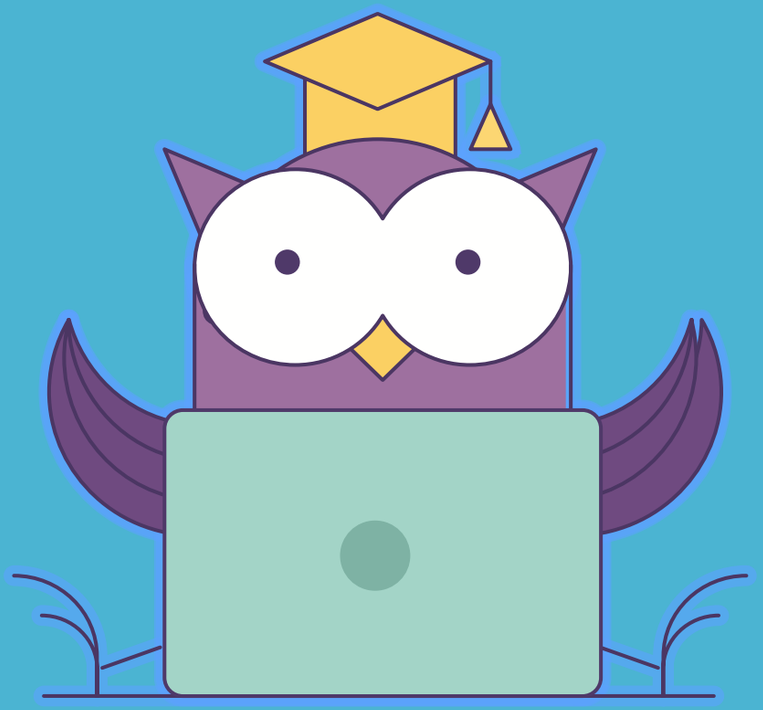
ОНЛАЙН-ОБРАЗОВАНИЕ

Как меня слышно и видно?



> Напишите в чат

+ если все хорошо

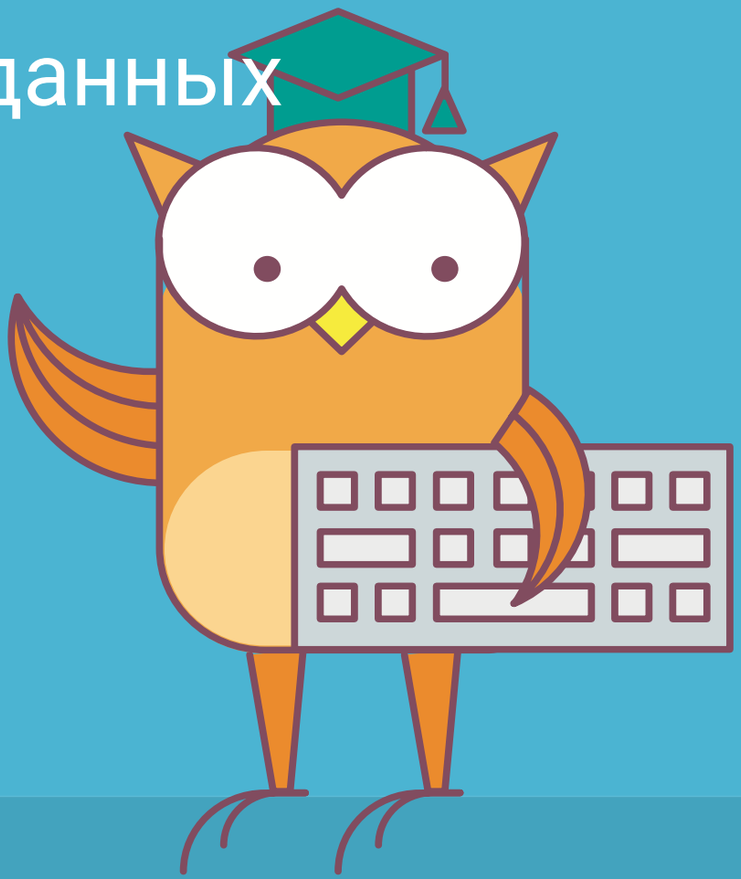
- если есть проблемы со звуком или с видео

!проверить запись!

OTUS

Работа с реляционными базами данных

Дмитрий Смаль



План занятия

- Установка и работа с PostgreSQL
- Простейшие SQL запросы
- 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

Установка PostgreSQL

OTUS

В консоли:

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

# установить PostgreSQL сервер и клиент
$ sudo apt-get install postgresql-10

# запустить PostgreSQL
$ sudo systemctl start postgresql

# Подключаемся под пользователем "по-умолчанию"
$ sudo -u postgres psql
```

В клиенте СУБД:

```
-- создаем "проектного" пользователя СУБД
postgres=# create user myuser password 'mypass';
CREATE ROLE

-- создаем "проектную" базу данных
postgres=# create database mydb owner myuser;
CREATE DATABASE
```

Создание простой таблицы и индекса (файл 001.sql):

```
create table events (
  id          serial primary key,
  owner       bigint,
  title       text,
  descr       text,
  start_date  date not null,
  start_time  time,
  end_date    date not null,
  end_time    time
);
create index owner_idx on events (owner);
create index start_idx on events using btree (start_date, start_time);
```

Выполнение SQL скрипта из консоли:

```
psql 'host=localhost user=myuser password=mypass dbname=mydb'  < 001.sql
```

Добавление строки:

```
insert into events(owner, title, descr, start_date, end_date)
values(42, 'new year', 'watch the irony of fate',
        '2019-12-31', '2019-12-31')
returning id;
```

Обновление строки:

```
update events
set end_date = '2020-01-01'
where id = 1;
```

Получение одной строки:

```
select * from events where id = 1
```

Получение нескольких строк:

```
select id, title, descr
from events
where owner = 42
  and start_date = '2020-01-01'
```

ResultSet:

```
 id(bigint) |  title(text)    |          descr(text)
------------+-----------------+-------------------------
  2         | new year        | watch the irony of fate
  3         | prepare ny      | make some olive
```

Создание подключения:

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

dns := "..."
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
```

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.

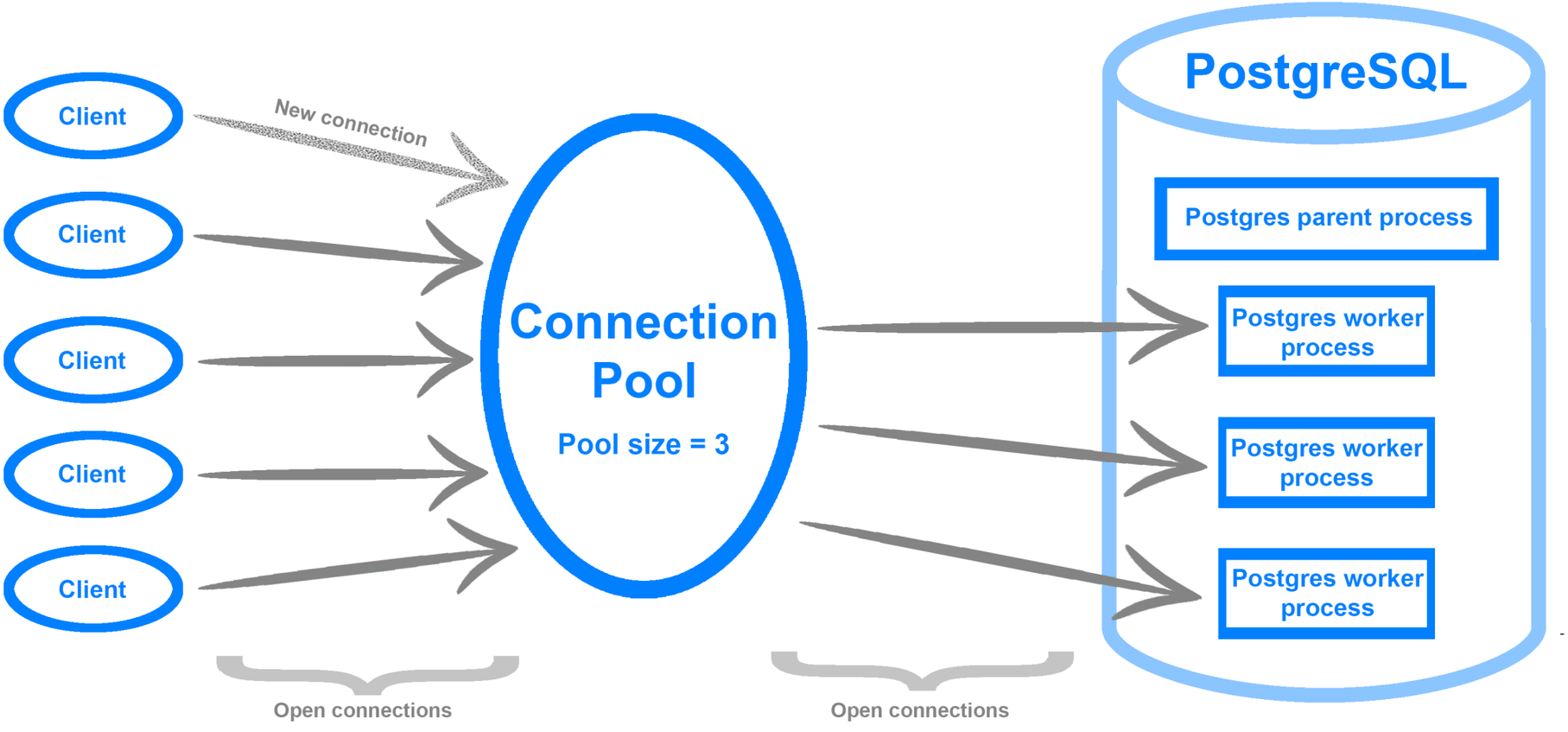
`sql.DB` - безопасен для конкурентного использования (так же как `http.Client`)

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)
```



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
```

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.Error() 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
}
```

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)
  }
}
```

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 := tx.ExecContext(ctx, query1, arg1, arg2)

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

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`.

Работа с транзакциями в Go:

```
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
```

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)
```

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
}
```

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)
```

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 на `database/sql`, прозрачно расширяющая стандартный функционал.

- `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 := sql.Open("pgx", dsn)  // *sqlx.DB

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

...
```

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",
})
```

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)
}
```

Драйвер для Postgres

Стандартный драйвер: https://github.com/lib/pq

Альтернатива: https://github.com/jackc/pgx
- Лучшая производительность
- Поддержка ~60 Postgres-специфичных типов данных
- Many more: https://github.com/jackc/pgx#features

Ссылки

http://go-database-sql.org/index.html https://golang.org/pkg/database/sql *https://jmoiron.github.io/sqlx

Заполните пожалуйста опрос

https://otus.ru/polls/4749/

Спасибо за внимание!