

# Работа с SQL

---

Golang Developer. Professional



Проверить, идет ли запись

# Меня хорошо видно & слышно?






# Хохлов Александр

**Архитектор платформенных решений в ГК Иннотех**

- **25+** лет в Информационных технологиях, последние 10 лет - архитектура и все что вокруг нее
- Занимаюсь вопросами проектирования, разработки (db/back/front/mobile), инфраструктуры, управления и развития технических команд.
- Преподаватель курсов:
  - Golang Developer. Basic
  - Golang Developer. Professional
  - System Design
  - Highload Architecture

 @khorost

 alexander.khokhlov@gmail.com

# Правила вебинара



Активно  
участвуем



Off-topic обсуждаем  
в учебной группе **#канал**  
**группы**



Задаем вопрос  
в чат или голосом



Вопросы вижу в чате,  
могу ответить не сразу

# Карта курса



**СТАРТ**

Начало работы с Go

Concurrency в Go

Работа с сетью и БД

Стандартные библиотеки и практики

Микросервисы

Проектная работа

**ФИНИШ**



# Маршрут вебинара

Установка и работа с PostgreSQL

Подключение к СУБД и настройка пула соединений

Выполнение запросов

Транзакции и SQL инъекции

# Цели вебинара

К концу занятия вы сможете

1. Изучить как выполнять запросы к PostgreSQL из Go

2. Понять особенности применения `sql.DB`, `sql.Rows`, `sql.Tx`



# Установка и работа с PostgreSQL

---



# Напишите в чат о вашем опыте с PostgreSQL

Например:

- есть опыт работы с PostgreSQL в Go
- есть опыт работы с другим SQL БД в Go
- есть опыт работы с PostgreSQL, но не в Go
- нет опыта работы с PostgreSQL, но есть опыт с другими SQL БД
- нет опыта работы с SQL БД



# 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
```



# 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 pg_data:/var/lib/postgresql/data \  
  -p 5432:5432 \  
  postgres
```

Ждём немного, пока СУБД поднимется

Подключаемся к серверу:

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



# Работаем с PostgreSQL локально

Работаем в клиенте СУБД:

```
postgres=# create database exampledb;
```

```
postgres=# create user otus_user with encrypted password 'otus_password';
```

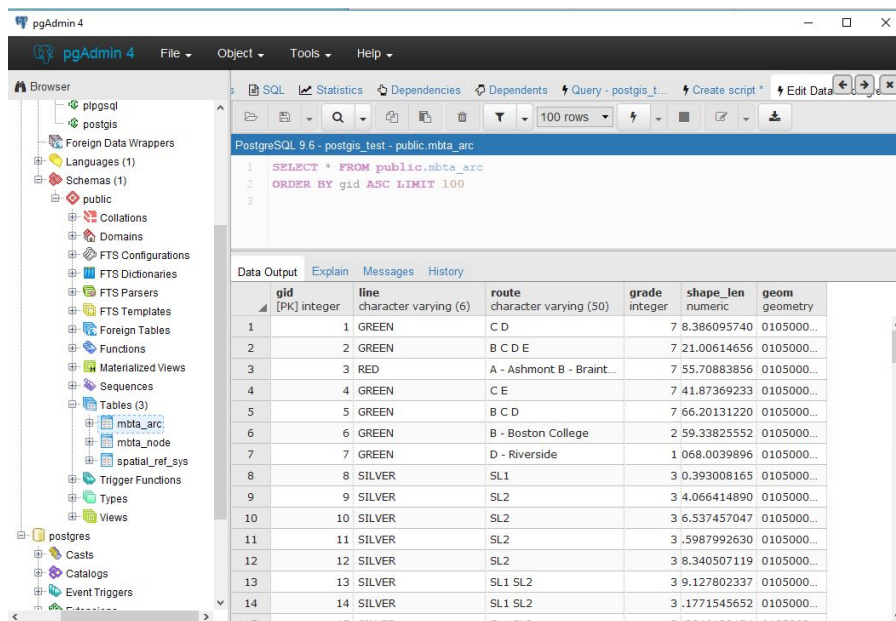
```
postgres=# grant all privileges on database exampledb to otus_user;
```



# Работаем с PostgreSQL локально

Удобный клиент с графическим интерфейсом:

<https://www.pgadmin.org/download/>



# Data Definition Language (DDL)

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
```



# Data Manipulation Language (DML)

Добавление строки:

```
insert into events (owner, title, descr, start_date, end_date)
values (
  42, 'new year', 'watch the irony of fate',
  '2024-12-31', '2024-12-31')
returning id;
```

Обновление строки:

```
update events
set end_date = '2025-01-01'
where id = 1;
```



# Data Query Language (DQL)

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'
```

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



# Вопросы



если есть вопросы



если вопросов нет

---

# Подключаем Go

---

# Подключение к PostgreSQL из Go

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
```

Обновление строки:

```
err := db.PingContext(ctx)
if err != nil {
    return fmt.Errorf("failed to connect to db: %w", 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

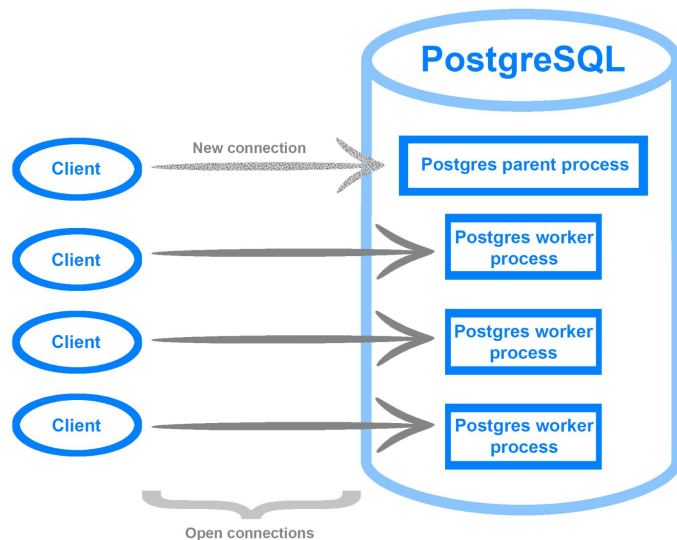
```
"user=myuser dbname=mydb sslmode=verify-full password=mypass"
```

- **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>

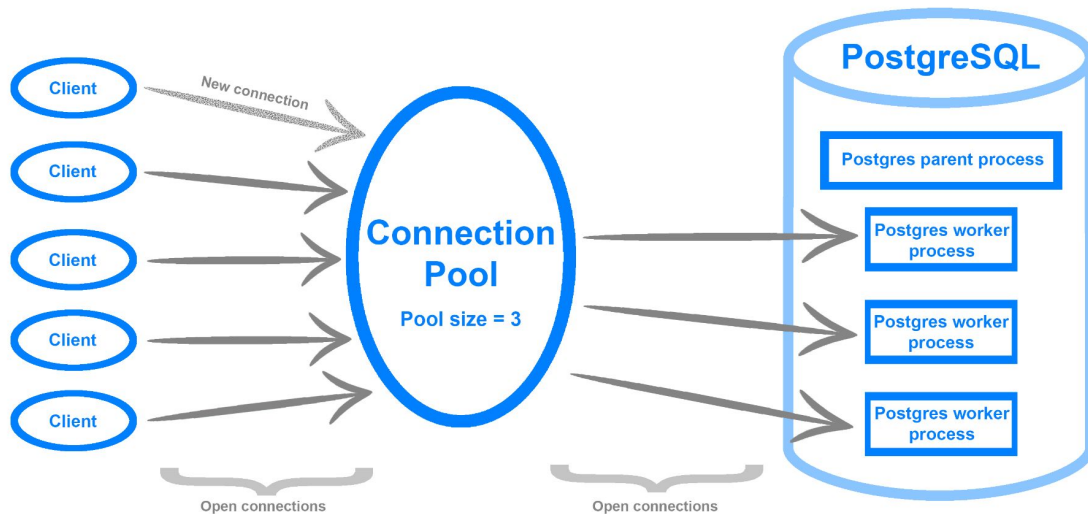


# Пул соединений



**Without Connection Pool**

# Пул соединений



With Connection Pool

# 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
// (по умолчанию - 0, без ограничений)
db.SetMaxOpenConns(n int)

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

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

// Макс. время ожидания подключения в пуле
db.SetConnMaxIdleTime(d time.Duration)
```

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



# 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)
}
```



# 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
}
```



# PreparedStatement

**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`.

Работа с транзакциями в Go:

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

// 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>



# 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
```



# Какие проблемы, ошибки вы видите в примере

```
_, 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:

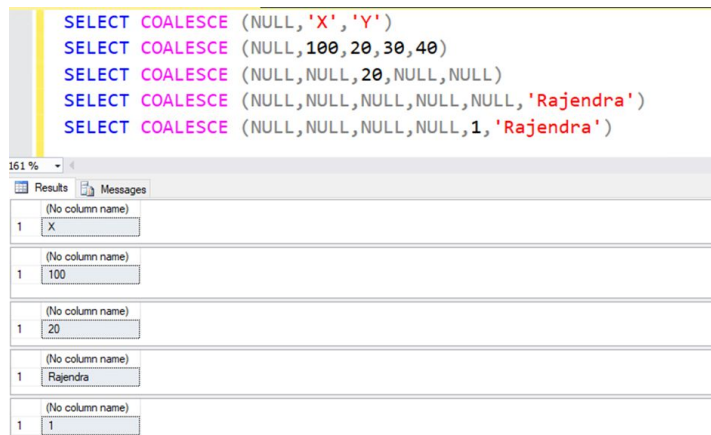
```
var id, realAge int64  
var name string  
var age sql.NullInt64  
err := db.QueryRowContext(ctx, "select * from users where id = 1").Scan(&id, &name, &age)  
  
if age.Valid {  
  realAge = age.Int64  
} else {  
  // обработка на ваше усмотрение  
}
```



# COALESCE

Функция **COALESCE** возвращает первый попавшийся аргумент, отличный от NULL. Если же все аргументы равны NULL, результатом тоже будет NULL. Это часто используется при отображении данных для подстановки некоторого значения по умолчанию вместо значений NULL

```
SELECT COALESCE(description, short_description, '(none)') ...
```



The screenshot shows a SQL query editor with five queries using the COALESCE function. Below the queries is a results pane showing the output of each query. The results pane has a zoom level of 161% and tabs for 'Results' and 'Messages'. The results are as follows:

	(No column name)
1	X
1	100
1	20
1	Rajendra
1	1

# 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"
```



# SQL Injection

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>



# Сборка сложного запроса

Сборка безопасного динамического запроса с условием и постраничной навигацией

```
var at []interface{}
idx := 1

if sm!="" {
    at = append(at, sm)
    query += fmt.Sprintf(` AND sm = $%d `, idx)
    idx++
}

total, err = queryAllCountByArg(at, query, "")
at = append(at, from)
at = append(at, count)

rows, err := db.QueryContext(ctx, query +
    fmt.Sprintf(` ORDER BY ss.doc DESC OFFSET $%d LIMIT $%d` , idx, idx+1), at...)
```



# 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 именованных 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 структуры:

```
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>

# Миграции

- <https://github.com/pressly/goose> - можно использовать как cli-тулзу и как библиотеку
- <https://flywaydb.org/> - пожалуй, самая популярная штука для миграций

*Protip:* flyway можно запускать из докер-контейнера перед запуском основного приложения, см. <https://github.com/flyway/flyway-docker>

# ORM

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

# SQL builder - squirrel

<https://github.com/Masterminds/squirrel>

Поиск данных:

```
sql, args, err := sq.Select( "id", "name", "phone").
    From( "phonebook" ).
    Where(sq.Eq{ "id": id}).
    PlaceholderFormat(sq.Dollar).
    ToSql()

if err != nil {
    log.Errorf( "Unable to build SELECT query: %v" , err)
    w.WriteHeader(500)
    return
}

log.Infof( "Executing SQL: %s" , sql)
row := conn.QueryRow(context.Background(), sql, args...)
```



# SQL builder - squirrel

Вставка данных:

```
sql, args, err := sq.Insert("phonebook").
    Columns("name", "phone").
    Values(rec.Name, rec.Phone).
    Suffix("RETURNING id").
    PlaceholderFormat(sq.Dollar).
    ToSql()

if err != nil {
    log.Errorf("Unable to build INSERT query: %v", err)
    w.WriteHeader(500)
    return
}

log.Infof("Executing SQL: %s", sql)
row := conn.QueryRow(context.Background(), sql, args...)
```



# SQL builder - goqu

<https://github.com/doug-martin/goqu>

Вставка данных:

```
ds := goqu.Insert("user").
    Cols("first_name", "last_name").
    Vals(
        goqu.Vals{"Greg", "Farley"},
        goqu.Vals{"Jimmy", "Stewart"},
        goqu.Vals{"Jeff", "Jeffers"},
    )

insertSQL, args, _ := ds.ToSQL()
```

```
type User struct {
    FirstName string `db:"first_name"`
    LastName  string `db:"last_name"`
}

ds := goqu.Insert("user").Rows(
    User{FirstName: "Greg", LastName: "Farley"},
    User{FirstName: "Jimmy", LastName: "Stewart"},
    User{FirstName: "Jeff", LastName: "Jeffers"},
)

insertSQL, args, _ := ds.ToSQL()
```





# Вопросы



если есть вопросы



если вопросов нет

---

# Практика

---

# Итоги занятия

## Рефлексия

---

# Домашнее задание

1. Используем скелет сервиса Календарь.
2. Добавляем из занятия работу с PostgreSQL



# Цели вебинара

Проверим достижение целей

1. Познакомились с библиотеками Go для работы в Postgresql
2. Знаем как особенности при работе с Go могут возникнуть и как их решать



# Список материалов для изучения

1. [PostgreSQL: Linux downloads \(Ubuntu\)](#)
2. [postgres - Official Image | Docker Hub](#)
3. [Как работать с Postgres в Go: практики, особенности, нюансы](#)
4. [Go database/sql tutorial](#)
5. [database/sql - Go Packages](#)
6. [Illustrated Guide to SQLX](#)



# Рефлексия



С какими впечатлениями уходите с вебинара?



Как будете применять на практике то, что узнали на вебинаре?

# Приглашаем на следующий вебинар

30 апреля 2025

## Работа с NoSQL



Ссылка на вебинар будет  
в ЛК за 15 минут



Материалы  
к занятию в ЛК —  
можно изучать



Обязательный материал  
обозначен красной  
лентой



# Заполните, пожалуйста, опрос о занятии

Мы читаем все ваши сообщения  
и берем их в работу 🖋️❤️

**You are our  
OTUS heroes**

