Лекция 4: Работа с базами данных из приложения. Профессиональный подход с ORM

Введение: Почему не "SQL в строках"?

Здравствуйте! На прошлой лекции мы создали наше первое API на Node.js и Express. Мы научились обрабатывать запросы и возвращать данные, которые временно хранили в массиве. Теперь нам нужно сделать следующий шаг — научиться сохранять эти данные надежно и надолго.

Мы уже знакомы с языком SQL. И первая мысль, которая может прийти в голову: "Почему бы просто не писать SQL-запросы в виде строк прямо в нашем JavaScript-коде?".

```
// ПЛОХОЙ, ОЧЕНЬ ПЛОХОЙ КОД!
const userId = req.params.id;
const query = "SELECT * FROM users WHERE id = " + userId;
const result = await db.execute(query);
res.json(result);
```

На первый взгляд, это просто. Но на самом деле, такой подход — это **бомба замедленного действия**, заложенная в фундамент вашего приложения. Он несет в себе три огромные проблемы:

1. **Уязвимость к SQL-инъекциям:** Это одна из самых старых и самых опасных атак. Если злоумышленник вместо 123 передаст в userId строку вроде '123 OR 1=1', он сможет обойти любую защиту и получить доступ ко всем данным в таблице. Конкатенация строк с пользовательским вводом — это прямой путь к взлому.
2. **Сложность поддержки:** Код, перемешанный со строками на другом языке (SQL), тяжело читать, отлаживать и изменять. Если вам понадобится добавить новое поле, придется вручную искать и исправлять десятки таких строк по всему проекту.
3. **Зависимость от конкретной СУБД:** Диалекты SQL у PostgreSQL, MySQL и MS SQL немного отличаются. Если вы решите "переехать" на другую базу данных, вам придется переписывать все запросы.

Профессиональные разработчики так не делают. Вместо этого они используют специальный слой абстракции — **ORM**.

1. Концепция ORM: Ваш личный "библиотекарь-переводчик"

ORM (Object-Relational Mapping — объектно-реляционное отображение) — это технология, которая создает "мост" между объектно-ориентированным миром вашего приложения (классы, объекты, методы в JavaScript) и реляционным миром базы данных (таблицы, строки, столбцы).

Аналогия: Представьте, что база данных — это огромный, строго организованный библиотечный архив, где все говорят на древнем, формальном языке SQL. Пытаться говорить с ним напрямую (писать "сырой" SQL) — долго и чревато ошибками.

ORM — это ваш личный, умный библиотекарь. Вы ему говорите на простом, современном языке:
"Найди мне, пожалуйста, пользователя с ID 123"

```
const user = await prisma.user.findUnique({ where: { id: 123 } });
```

А "библиотекарь" (ORM) сам:

1. Переводит ваш запрос на строгий и безопасный язык SQL (SELECT * FROM "users" WHERE "id" = $1).
2. Использует параметризованные запросы, чтобы защититься от SQL-инъекций.
3. Выполняет запрос к базе данных.
4. Приносит вам результат в виде удобного JS-объекта.

Используя ORM, мы получаем **безопасность**, **продуктивность** и **независимость от конкретной СУБД**.

2. Обзор инструментов: Prisma vs Sequelize

В мире Node.js есть несколько популярных ORM. Мы рассмотрим два основных "игрока", чтобы вы понимали ландшафт.

- **Sequelize:** Это "классика", ветеран мира Node.js. Очень зрелый и стабильный проект. Он использует паттерн *Active Record*, где модели — это "умные" объекты, у которых есть методы .save(), .update() и т.д. Он очень мощный, но его API может быть довольно сложным для новичков.
- **Prisma:** Это представитель "нового поколения" ORM. Prisma — это не совсем ORM, а скорее **type-safe database toolkit** (безопасный к типам инструментарий для баз данных). Его ключевая идея — **"Schema-first"**:
    1. Вы описываете схему вашей базы данных на специальном, простом языке в файле schema.prisma.
    2. Затем вы запускаете команду, и Prisma **генерирует** для вас идеально типизированный JavaScript-клиент для работы с этой схемой.

В рамках нашего курса мы будем использовать **Prisma**, потому что она проще для новичков, заставляет писать более безопасный и предсказуемый код и наглядно демонстрирует все современные концепции работы с БД. Поняв принципы Prisma, вы легко освоите Sequelize или любую другую ORM.

3. Практика с Prisma: Подключаем "память" к нашему API

Давайте заменим наш временный массив users на настоящую базу данных PostgreSQL, управляемую через Prisma.

Шаг 1: Настройка окружения

Мы не будем устанавливать PostgreSQL на наши компьютеры. Вместо этого мы используем **Docker** (о котором подробно поговорим в следующем семестре). Мы просто создадим файл

docker-compose.yml, который одной командой "поднимет" для нас готовый к работе сервер PostgreSQL в контейнере.

Шаг 2: Установка Prisma

```
# Устанавливаем Prisma CLI как зависимость для разработки
npm install prisma --save-dev
# Устанавливаем Prisma Client как обычную зависимость
npm install @prisma/client
```

Шаг 3: Инициализация Prisma

Выполняем команду, которая создаст для нас всю необходимую структуру:

```
npx prisma init --datasource-provider postgresql
```

Эта команда создаст папку prisma с файлом schema.prisma и дополнит наш .env файл строкой подключения к базе данных (DATABASE_URL).

Шаг 4: Определение модели данных

Открываем файл prisma/schema.prisma и описываем нашу модель User. Этот файл — наш единственный "источник правды" о структуре базы.

```
// prisma/schema.prisma
model User {
  id        Int      @id @default(autoincrement())
  email     String   @unique
  name      String?
  password  String
  createdAt DateTime @default(now())
}
```

Шаг 5: Миграция

Теперь самое интересное. Нам нужно, чтобы Prisma посмотрела на нашу схему и создала в реальной базе данных соответствующую таблицу. Для этого используется механизм **миграций**.

```
# Prisma сама сравнит схему и состояние БД, сгенерирует SQL и применит его
npx prisma migrate dev --name init_user_model
```

Эта команда создаст в папке prisma/migrations SQL-файл с командой CREATE TABLE "User" (...) и выполнит его. Теперь наша база данных готова к работе! Миграции, как и код, хранятся в Git, что позволяет всей команде поддерживать структуру БД в одинаковом состоянии.

4. CRUD через Prisma: Переписываем наше API

Теперь мы можем переписать наши обработчики роутов, заменив работу с массивом на асинхронные вызовы к Prisma Client.

Шаг 1: Инициализация клиента

В нашем основном файле (index.js или app.js) мы создаем экземпляр Prisma Client.

```
const { PrismaClient } = require('@prisma/client');
const prisma = new PrismaClient();
```

Шаг 2: Рефакторинг роутов

GET /users — получение всех пользователей:

```
// БЫЛО: res.json(users);
// СТАЛО:
router.get('/', async (req, res) => {
  const allUsers = await prisma.user.findMany({
    // Prisma автоматически уберет поле password из ответа
    select: { id: true, email: true, name: true }
  });
  res.json(allUsers);
});
```

GET /users/:id — получение одного пользователя:

```
// БЫЛО: const user = users.find(u => u.id === id);
// СТАЛО:
router.get('/:id', async (req, res) => {
  const { id } = req.params;
  const user = await prisma.user.findUnique({
    where: { id: parseInt(id) },
    select: { id: true, email: true, name: true }
  });
  // ... логика обработки, если не найдено ...
  res.json(user);
});
```

POST /users — создание пользователя:

```
// БЫЛО: users.push(newUser);
// СТАЛО:
router.post('/', async (req, res) => {
  const newUser = await prisma.user.create({
    data: {
      email: req.body.email,
      name: req.body.name,
      // В реальном приложении здесь будет хешированный пароль!
      password: req.body.password
    }
  });
  res.status(201).json(newUser);
});
```

Как видите, код стал даже более читаемым и декларативным. Мы описываем, **что** мы хотим (findUnique, create), а Prisma сама решает, **как** это сделать, генерируя безопасный и эффективный SQL-запрос.

Заключение

Сегодня мы сделали критически важный шаг — подключили нашему приложению "долговременную память". Мы поняли, почему профессиональные разработчики избегают

"сырых" SQL-запросов и используют ORM. Мы на практике познакомились с современным инструментом Prisma, научились описывать модели данных, управлять структурой базы с помощью миграций и выполнять все базовые CRUD-операции.

Наше приложение теперь не просто "игрушка", а система, способная надежно хранить и обрабатывать данные. На следующей лекции мы продолжим его совершенствовать и добавим "замки и ключи" — систему **аутентификации и авторизации**.