|  |
| --- |
|  |
| МИНОБРНАУКИ РОССИИ |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования  **«МИРЭА − Российский технологический университет»**  **РТУ МИРЭА** |
| Институт кибербезопасности и цифровых технологий |
| Кафедра КБ-14 «Цифровые технологии обработки данных» |

**ОТЧЕТ**

**по практической работе**

**«№3. РАЗРАБОТКА БАЗЫ ДАННЫХ»**

**по дисциплине «Обработка данных с веб-ресурсов и поисковых машин»**

|  |  |  |  |
| --- | --- | --- | --- |
| Выполнил |  |  | Парамонов П.К.  *фамилия, имя, отчество* |
| шифр | 22Б0723 | группа | БСБО-05-22 |
|  |  |  |  |
| Проверил |  | к.т.н., доцент  *ученая степень, должность* | Ильин Д.Ю.  *фамилия, имя, отчество* |

**Москва 2024г.**

**Цель практической работы**

Цель настоящей практической работы — научиться проектировать базы данных для хранения данных веб-ресурсов и использовать ORM.

**Задачи практической работы**

Для достижения поставленной цели необходимо решить следующие

задачи:

1. Спроектировать базу данных исходя из их характера и особенностей.
2. Сформировать проект на Node.js на основе результатов работы №2.
3. Установить программные библиотеки, реализующие ORM.
4. Разработать программный код ORM-моделей для взаимодействия с БД.
5. Доработать программный код из работы №2 для использования моделей.
6. Выполнить сбор данных.
7. Удостовериться в корректном хранении результатов в БД.
8. Подготовить ответы на контрольные вопросы.
9. Составить отчет о проведенной работе.

**Ход выполнения практической работы**

В ходе выполнения практической работы был использован Sequelize для ORM, благодаря чему происходит взаимодействие с PostgreSQL.

В работе будут использоваться 3 скрипта:

1. Подключение к базе данных;
2. Модель данных;
3. Скрипт переноса данных.

Листинг кода каждого скрипта будет представлен в приложении А. Ниже представлены результаты работы каждого из скриптов



Рис 1 – результат работы скрипта database.js

Скрипт database.js является основным и используется во всех последующих скриптах.

Скрипт modelCreation.js является вспомогательным для скрипта import.js. В нём определяется ORM модель данных для дальнейшего использования.

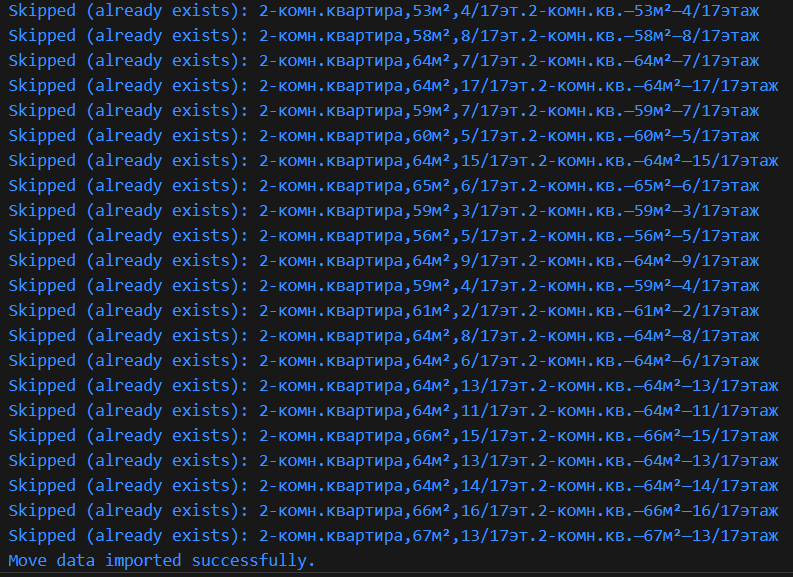


Рис 2 – пример работы скрипта import.js

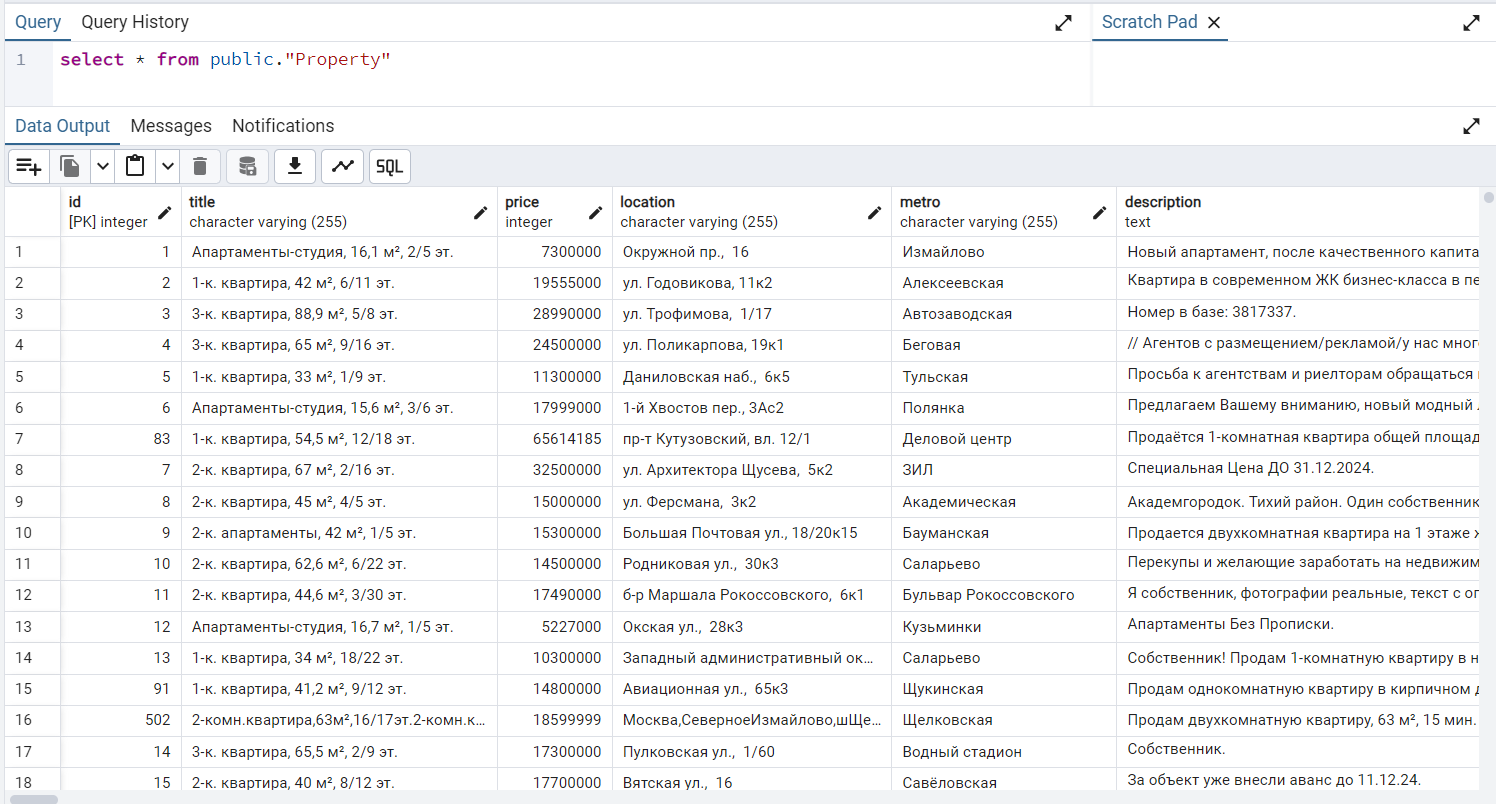


Рис 3 – пример заполненных данных в БД

**Трудности в ходе выполнения практической работы**

Никаких трудностей не возникло.

**Ответы на контрольные вопросы**

**1. В чем отличия подходов с использованием «чистого» SQL, Query Builder’а и ORM?**

* **Чистый SQL**: Написание запросов вручную. Гибкость, полный контроль, но больше кода и сложность сопровождения.
* **Query Builder**: Генерация SQL-запросов через программный интерфейс (например, Knex.js). Упрощает создание запросов, но требует знания синтаксиса SQL.
* **ORM**: Работа с базой данных через объекты в коде (например, Sequelize). Минимум SQL, быстрый старт, но иногда менее эффективен и гибок.

**2. Что такое миграция базы данных?**

Процесс изменения структуры базы данных (создание, изменение таблиц, индексов и т.д.) с сохранением истории изменений. Используется для синхронизации схемы базы данных с кодом проекта.

**3. Что такое гетерогенные источники данных?**

Источники данных разнородных типов, например, базы данных, файлы, API, стримы. Они имеют разные форматы, структуры и могут требовать преобразований для совместного использования.

**4. Чем отличается SQL-синтаксис реляционных СУБД?**

* Разные СУБД (PostgreSQL, MySQL, Oracle, SQLite) поддерживают базовый SQL, но могут отличаться:
  + Типы данных (например, SERIAL в PostgreSQL и AUTO\_INCREMENT в MySQL).
  + Функции (например, STRING\_AGG в PostgreSQL и GROUP\_CONCAT в MySQL).
  + Механизмы работы с транзакциями и индексацией.

**5. В чем отличия файл-серверной, клиент-серверной и встраиваемой СУБД?**

* **Файл-серверная СУБД**: Данные хранятся в файлах, доступ к ним идёт напрямую (например, Microsoft Access). Простая, но не подходит для больших нагрузок.
* **Клиент-серверная СУБД**: Клиенты обращаются к серверу, который обрабатывает запросы (например, PostgreSQL, MySQL). Подходит для многопользовательских систем.
* **Встраиваемая СУБД**: Интегрируется в приложение, работает локально (например, SQLite). Легковесная, не требует установки сервера.

**Вывод**

В ходе выполнения практической работы «Разработка базы данных» были рассмотрены особенности работы с библиотекой Sequelize, благодаря которой было успешно перенесены данные в PostgreSQL.

**Приложение А. Листинг кода**

Database.js

const { Sequelize } = require('sequelize');

// Настройка подключения к базе данных

const sequelize = new Sequelize('parse\_project', 'parse\_project\_user', '00000000', {

    host: 'localhost',       // Адрес сервера базы данных

    dialect: 'postgres',     // Указываем, что используем PostgreSQL

    logging: false,

    define: {

      freezeTableName: true,

      timestamps: true       // Добавлять поля createdAt и updatedAt

    }

  });

  (async () => {

        try {

            await sequelize.authenticate();

            console.log('Connection to PostgreSQL has been established successfully.');

        } catch (error) {

            console.error('Error syncing database:', error);

        }

  })();

  module.exports = sequelize;

modelCreation.js

const { DataTypes} = require('sequelize');

const sequelize = require('./database');

const Property = sequelize.define('Property', {

    title: {

      type: DataTypes.STRING,

      allowNull: false

    },

    price: {

        type: DataTypes.INTEGER,

        allowNull: false

    },

    location: {

        type: DataTypes.STRING

    },

    metro: {

        type: DataTypes.STRING

    },

    description: {

        type: DataTypes.TEXT

    },

    link: {

        type: DataTypes.TEXT

    },

    source: {

      type: DataTypes.STRING

    }

  }, {

    timestamps: false

  });

  module.exports = Property;

import.js

const Property = require('./modelCreation');

const avitoJsonData = require('../Jsons/data-avito.json');

const cianJsonData = require('../Jsons/data-cian.json');

const mirkvJsonData = require('../Jsons/data-mirkv.json');

const moveJsonData = require('../Jsons/data-move.json');

(async () => {

    try {

        // Перебираем массив из JSON и добавляем данные в таблицу

        for (const item of avitoJsonData) {

          try {

            // Проверяем, существует ли запись с таким же "link" (или другим уникальным полем)

            const existingRecord = await Property.findOne({ where: { link: item.link } });

            if (!existingRecord) {

              // Если запись не существует, добавляем её

              await Property.create({

                title: item.title,

                price: item.price,

                location: item.location,

                metro: item.metro,

                description: item.description,

                link: item.link,

                source: "Авито"

              });

              console.log(`Added: ${item.title}`);

            } else {

              console.log(`Skipped (already exists): ${item.title}`);

            }

          } catch (innerError) {

            // Логируем ошибку для конкретной строки, но продолжаем обработку остальных

            console.error(`Error processing item: ${item.title}, Error: ${innerError.message}`);

          }

        }

        console.log('Avito data imported successfully.');

    } catch (error) {

        // Ловим общие ошибки

        console.error('Error importing Avito data:', error);

    }

    try {

        // Перебираем массив из JSON и добавляем данные в таблицу

        for (const item of cianJsonData) {

          try {

            // Проверяем, существует ли запись с таким же "link" (или другим уникальным полем)

            const existingRecord = await Property.findOne({ where: { link: item.link } });

            if (!existingRecord) {

              // Если запись не существует, добавляем её

              await Property.create({

                title: item.title,

                price: item.price,

                location: item.location,

                metro: item.metro,

                description: item.description,

                link: item.link,

                source: "Циан"

              });

              console.log(`Added: ${item.title}`);

            } else {

              console.log(`Skipped (already exists): ${item.title}`);

            }

          } catch (innerError) {

            // Логируем ошибку для конкретной строки, но продолжаем обработку остальных

            console.error(`Error processing item: ${item.title}, Error: ${innerError.message}`);

          }

        }

        console.log('Cian data imported successfully.');

    } catch (error) {

        // Ловим общие ошибки

        console.error('Error importing Cian data:', error);

    }

    try {

        // Перебираем массив из JSON и добавляем данные в таблицу

        for (const item of mirkvJsonData) {

          try {

            // Проверяем, существует ли запись с таким же "link" (или другим уникальным полем)

            const existingRecord = await Property.findOne({ where: { link: item.link } });

            if (!existingRecord) {

              // Если запись не существует, добавляем её

              await Property.create({

                title: item.title,

                price: item.price,

                location: item.location,

                metro: item.metro,

                description: item.description,

                link: item.link,

                source: "Мир квартир"

              });

              console.log(`Added: ${item.title}`);

            } else {

              console.log(`Skipped (already exists): ${item.title}`);

            }

          } catch (innerError) {

            // Логируем ошибку для конкретной строки, но продолжаем обработку остальных

            console.error(`Error processing item: ${item.title}, Error: ${innerError.message}`);

          }

        }

        console.log('Mirkv data imported successfully.');

    } catch (error) {

        // Ловим общие ошибки

        console.error('Error importing Mirkv data:', error);

    }

    try {

      // Перебираем массив из JSON и добавляем данные в таблицу

      for (const item of moveJsonData) {

        try {

          // Проверяем, существует ли запись с таким же "link" (или другим уникальным полем)

          const existingRecord = await Property.findOne({ where: { link: item.link } });

          if (!existingRecord) {

            // Если запись не существует, добавляем её

            await Property.create({

              title: item.title,

              price: item.price,

              location: item.location,

              metro: item.metro,

              description: item.description,

              link: item.link,

              source: "Move"

            });

            console.log(`Added: ${item.title}`);

          } else {

            console.log(`Skipped (already exists): ${item.title}`);

          }

        } catch (innerError) {

          // Логируем ошибку для конкретной строки, но продолжаем обработку остальных

          console.error(`Error processing item: ${item.title}, Error: ${innerError.message}`);

        }

      }

      console.log('Move data imported successfully.');

  } catch (error) {

      // Ловим общие ошибки

      console.error('Error importing Move data:', error);

  }

})();