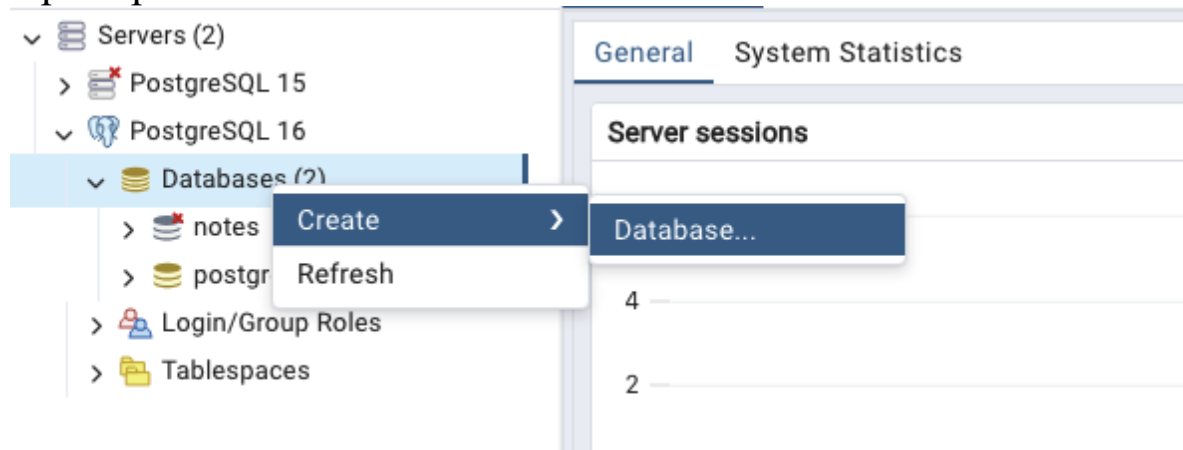


Node js

Шаг 1

Рассмотрим работу с базой данных PostgreSQL. Для начала запускаем PostgreSQL и создаем новую базу с названием posts.

Пример:



Работа продолжается в проекте, который создавался в первой работе.

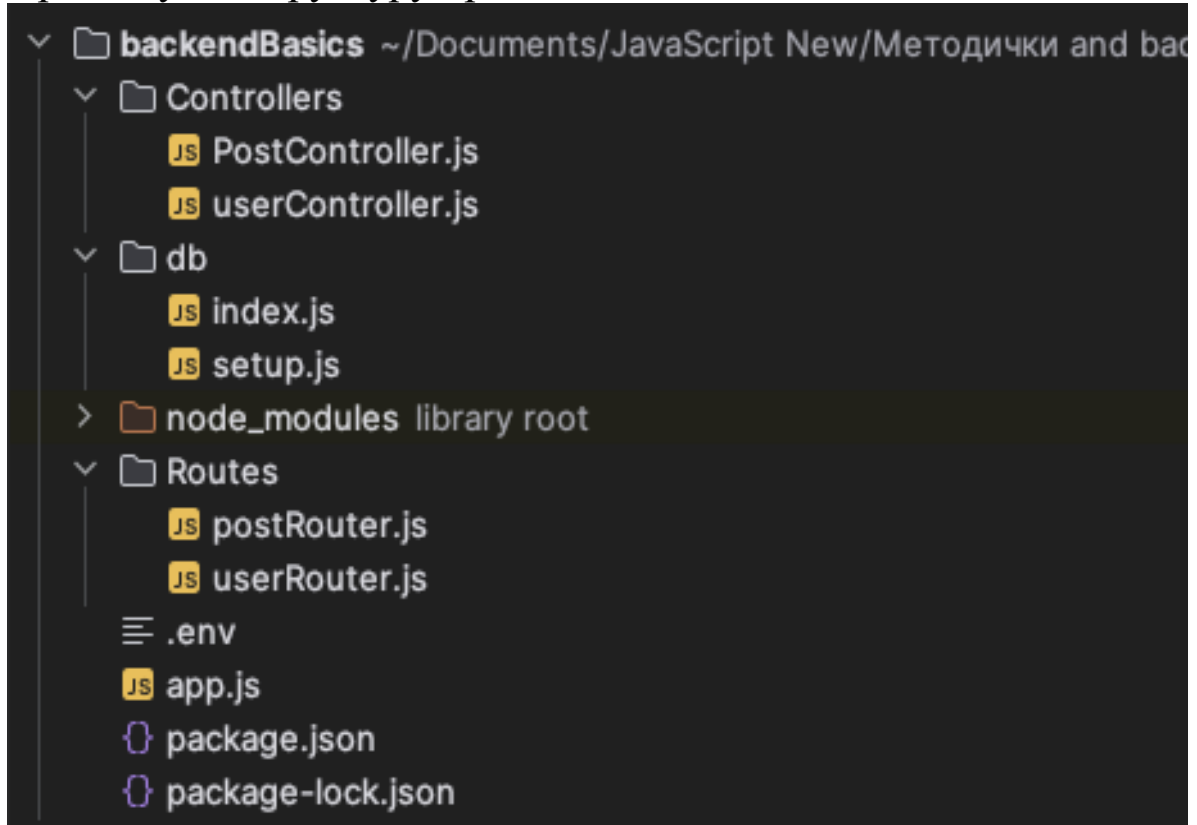
Далее установим пакет командой **npm i dotenv** это популярный модуль в экосистеме Node.js, который используется для загрузки переменных окружения из файла. env в process.env. Это позволяет безопасно хранить конфиденциальные данные, такие как пароли, ключи API, строки подключения к базам данных и другие настройки конфигурации, вне исходного кода.

Создадим файл в корне проекта с названием. env и занесем переменные для подключения к базе данных.

Пример файла:

```
DB_USER=postgres
DB_HOST=localhost
DB_NAME=posts
DB_PASSWORD=root
DB_PORT=5432
```

Организируйте структуру проекта как показано ниже:



Шаг 2

В папке db откроем файл index.js и настроим подключение к базе данных.

Пример подключения:

```
const { Pool } = require('pg');
require('dotenv').config(); // Подключаем переменные окружения

// Создаем клиент PostgreSQL с настройками из .env
const pool = new Pool({
  user: process.env.DB_USER,
  host: process.env.DB_HOST,
  database: process.env.DB_NAME,
  password: process.env.DB_PASSWORD,
  port: process.env.DB_PORT,
});

module.exports = pool;
```

Pool представляет собой пул соединений, который управляет группой соединений с базой данных.

Далее настроим файл `setup.js` в папке **db** в нем будет находиться наш SQL код для создания таблиц в базе данных.

Пример кода:

```
async function createTables(pool) { Show usages
  try {

    // SQL-запросы для создания таблиц
    const createUsersTable = `
      CREATE TABLE IF NOT EXISTS users (
        id SERIAL PRIMARY KEY,
        name VARCHAR(100) NOT NULL,
        email VARCHAR(100) UNIQUE NOT NULL,
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
      )
    `;

    const createPostsTable = `
      CREATE TABLE IF NOT EXISTS posts (
        id SERIAL PRIMARY KEY,
        user_id INTEGER REFERENCES users(id) ON DELETE CASCADE,
        title VARCHAR(255) NOT NULL,
        body TEXT,
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
      )
    `;

    // Выполняем SQL-запросы
    await pool.query(createUsersTable);
    console.log('Users table created.');

await pool.query(createPostsTable);  
console.log('Posts table created.');



} catch (error) {  
  console.error('Error creating tables:', error.message);  
}



}



module.exports = createTables;


```

Передача pool в createTables: предполагается, что функция `createTables` принимает пул соединений и использует его для выполнения запросов. Это позволяет использовать уже настроенный пул для создания таблиц.

pool.query(): Каждый запрос к базе данных будет брать соединение из пула, использовать его и возвращать в пул автоматически.

Шаг 3

Напишем код для первого контроллера `UserController`, в нем будет описана вся основная логики работы с Базой данных. Импортируем нашу модель БД и используем `pool` соединение.

Код для получения всех пользователей:

```
const pool = require('../db')

class UserController { Show usages

  async getAllUsers(req, res) { Show usages
    try {
      const user = await pool.query(`SELECT * FROM users`)
      res.json(user.rows)
    } catch (error) {
      console.log(error)
    }
  }
}
```

Код для создания нового пользователя:

```
async createUser(req, res) { Show usages
  const {name, email} = req.body
  try {
    const user = await pool.query(`INSERT INTO users (name, email) VALUES ($1, $2) RETURNING *`, [name, email]);
    res.json(user.rows);
  } catch (error) {
    console.error('error', error)
  }
}
```

Код для обновления данных пользователя:

```
async updateUser(req, res) {
  const id = parseInt(req.params.id, 10);
  const {name, email} = req.body
  try {
    const user = await pool.query(`UPDATE users SET name = $1, email = $2 WHERE id = $3 RETURNING *`, [name, email, id]);
    res.json(user.rows);
  } catch (error) {
    console.error('error', error)
  }
}
```

Код для удаления пользователя:

```
async deleteUser(req, res) { Show usages
  const id = req.params.id;
  const user = await pool.query(`DELETE FROM users WHERE id = $1`, [id])
  res.json(user.rows[0]);
}
```

После написания всех методов экспортируем класс для дальнейшего использования.

```
module.exports = new UserController();
```

Шаг 4

Далее заходим в папку Routes и напишем маршруты в файле userRoutes для контроллера.

Пример кода:

```
const Router = require('express')
const router = new Router()
const userController = require('../Controllers/userController')

router.get('/user', userController.getAllUsers)
router.post(path: '/user', userController.createUser )
router.put(path: '/user/:id', userController.updateUser)
router.delete(path: '/user/:id', userController.deleteUser)

module.exports = router
```

Шаг 5

Дописываем корневой файл app.js и производим тест через Insomnia.

Пример кода:

```
const express = require('express');
const createTables = require('./db/setup');
const pool = require('./db/index');

const userRouter = require('./routes/userRouter');

// Создаем приложение express
const app = express();
const PORT = process.env.PORT || 5001;

// Middleware для парсинга JSON в теле запроса
app.use(express.json());

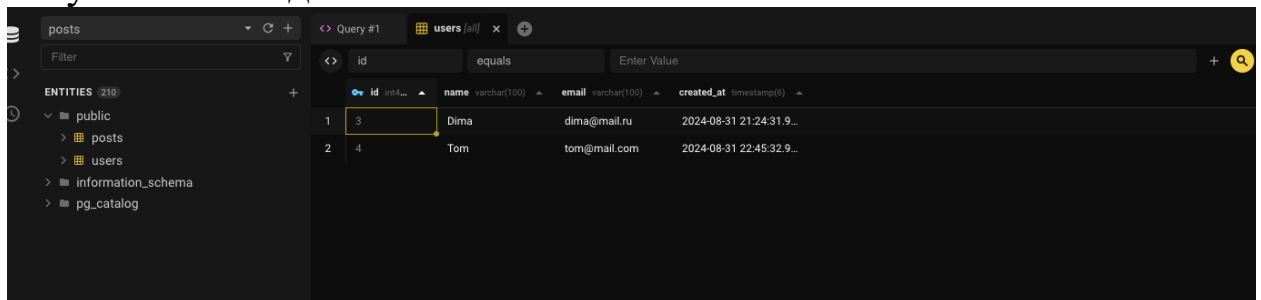
app.use('/api', userRouter)

async function initializeApp() { Show usages
  try {
    // Создаем таблицы
    await createTables(pool); // Передаем pool как параметр для использования в createTables

    // Запускаем сервер и выводим сообщение о том, что сервер запущен
    app.listen(PORT, { hostname: () => {
      console.log(`Server is running on port ${PORT}`);
    }});
  } catch (error) {
    console.error('Error initializing app:', error.message);
  }
}

// Инициализируем приложение
initializeApp();
```

Результат базы данных:



	id	name	email	created_at
1	3	Dima	dima@gmail.ru	2024-08-31 21:24:31.9...
2	4	Tom	tom@gmail.com	2024-08-31 22:45:32.9...

Задачи

Реализовать до конца контроллер с постами и прописать маршруты для данного контроллера.

Вариант 1

Создайте API для управления коллекцией книг в библиотеке. API должен позволять пользователям добавлять, редактировать, удалять и просматривать книги. Также реализовать функционал для управления авторами, жанрами и отслеживания статуса книги (в наличии, взята в аренду и т.д.).

Вариант 2

Разработайте API для онлайн-магазина, позволяющее пользователям просматривать товары, добавлять их в корзину, оформлять заказы и отслеживать их выполнение. Администраторы должны иметь возможность управлять каталогом товаров и обрабатывать заказы.

Вариант 3

Создайте API для системы управления обучением, где преподаватели могут создавать курсы, добавлять уроки и управлять студентами, а студенты могут записываться на курсы, просматривать материалы и отслеживать свой прогресс.