Министерство науки и высшего образования Российской Федерации

федеральное государственное автономное   
образовательное учреждение высшего образования   
«Самарский национальный исследовательский университет   
имени академика С.П. Королева»

Институт информатики и кибернетики

Кафедра технической кибернетики

Отчет по лабораторным работам

Дисциплина: «Технологии сетевого программирования»

Выполнил: Кузьмин Д. О.

Студент группы:

6302-010302D

Самара, 2025

**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 3](#_Toc197191635)

[1 Архитектура приложения 4](#_Toc197191636)

[2 Описание API в формате таблицы 5](#_Toc197191637)

[3 Crud методы 6](#_Toc197191638)

[4 Тестирование API 7](#_Toc197191639)

[5 Аутентификация и авторизация 10](#_Toc197191640)

[6 Структура интерфейса 11](#_Toc197191641)

[7 Примеры рендеринга данных и передачи параметров 12](#_Toc197191642)

[8 Описание структуры контейнеризации и настройки окружения 13](#_Toc197191643)

[9 Скриншоты работающего приложения в контейнерах 15](#_Toc197191644)

[Приложение А 16](#_Toc197191645)

# ВВЕДЕНИЕ

SamaraCar — это веб-приложение для интернет-магазина автозапчастей, разработанное с использованием стека PERN (PostgreSQL, Express, React, Node.js). Проект реализует полнофункциональный RESTful API и включает как серверную, так и клиентскую части. Приложение контейнеризировано с помощью Docker, что обеспечивает лёгкий запуск и масштабирование

SamaraCar разработан как демонстрационный проект, иллюстрирующий практические навыки в бэкенд- и фронтенд-разработке, проектировании API, работе с реляционными базами данных, а также настройке и контейнеризации современного веб-приложения.

# Архитектура приложения

**Общая структура**

Приложение реализовано по клиент-серверной архитектуре и разделено на три основных компонента, каждый из которых размещён в отдельном Docker-контейнере:

1. Frontend — интерфейс пользователя, построенный на React с использованием фреймворка Bootstrap.
2. Backend — серверная часть, реализованная на Node.js с использованием Express. API обеспечивает обработку запросов и взаимодействие с базой данных через SQL-запросы или ORM.
3. PostgreSQL — реляционная база данных, в которой хранится информация о пользователях, товарах и других сущностях.

Взаимодействие между компонентами обеспечивается с помощью Docker Compose, что упрощает процесс сборки, запуска и конфигурации приложения. Такой подход обеспечивает изоляцию окружения, удобное масштабирование и позволяет без труда разворачивать проект как в локальной среде разработчика, так и на сервере в продакшене.

**Компоненты**

**1. Frontend (React)**

Пользовательский интерфейс реализован на React с применением библиотеки Bootstrap для стилизации. Проект организован по компонентной структуре и использует следующие инструменты:

* axios — для отправки HTTP-запросов к backend API;
* react-router-dom — маршрутизация внутри Single Page Application;
* mobx и mobx-react-lite — управление глобальным состоянием через хранилища (UserStore.js, CartStore.js, и др.).

**2. Backend (Node.js + Express)**

Серверная часть приложения реализована с использованием Node.js и Express:

* Взаимодействие с базой данных осуществляется через ORM Sequelize (в отличие от MongoDB/Mongoose).
* Реализовано JWT-аутентификация (используется только access token, без refresh).
* Система ролей: доступ к защищённым маршрутам ограничен middleware (например, только для admin).
* Конфигурация (ключи, порты, строки подключения) — через .env и docker-compose.yml.

Функции backend включают:

* обработку HTTP-запросов (CRUD для товаров, пользователей, заказов и пр.);
* валидацию данных;
* авторизацию и защиту маршрутов;
* генерацию токенов и работу с ними.

**3. База данных — PostgreSQL**

В качестве основного хранилища данных используется PostgreSQL, развернутая в отдельном контейнере.

Особенности реализации:

* Данные структурированы в таблицы, соответствующие сущностям приложения: users, products, orders, categories, cart\_items и др.
* В проект включена папка db-init, содержащая дамп данных — SQL-скрипты для автоматического создания схемы и начального наполнения базы при первом запуске контейнера.
* Доступ к БД осуществляется через Sequelize, позволяющий работать с БД на уровне моделей и связей.
* Контейнер PostgreSQL использует volume postgres\_data для постоянного хранения данных между перезапусками.

**Интеграция через Docker Compose**

Все три компонента объединены в одну систему с помощью docker-compose.yml:

* Настроены сети, переменные окружения и порты.
* Используются зависимости depends\_on для контроля порядка запуска (сначала БД, затем backend, потом frontend).
* Один файл .env управляет переменными как для backend, так и для БД.

Такой подход гарантирует:

* Быстрый запуск на любой машине с Docker;
* Простое масштабирование и сопровождение;
* Изоляцию окружения для разработки и продакшена.

# Описание API в формате таблицы

|  |  |  |
| --- | --- | --- |
| Метод | URL | Описание |
| GET | /user/auth | Проверка авторизации пользователя |
| GET | /user | Получение данных о всех пользователях (для админа) |
| GET | /product | Получить все товары (продукты) |
| GET | /product/:id | Возвращает продукт с заданным ID |
| GET | /order | Получить заказы пользователя |
| GET | /order/:id | Возвращает заказ с заданным ID |
| GET | /order/admin/all | Получить все заказы всех пользователей |
| GET | /category | Возвращает все категории |
| GET | /category/:id | Возвращает категорию с заданным ID |
| GET | /cart | Получить корзину пользователя |
| POST | /user/registration | Регистрация пользователя |
| POST | / user/login | Авторизация пользователя |
| POST | /product | Создание товара (продукта) |
| POST | /order | Создание заказа |
| POST | /category | Создание категории |
| POST | /cartitem | Добавление продукта в корзину |
| DELETE | /user | Удаление аккаунта пользователя |
| DELETE | /product/:id | Удаление продукта с заданным ID |
| DELETE | /category/:id | Удаление категории с заданным ID |
| DELETE | /cart | Очистка корзины от продуктов |
| DELETE | /cartitem/:cart\_item\_id | Удаление конкретного продукта из корзины |
| PATCH | /cartitem/:cart\_item\_id | Уменьшение кол-ва конкретного продукта на 1 из корзины |
| PUT | /product/:id | Изменить информацию о продукте с заданным ID |
| PUT | /order/:order\_id | Изменить статус заказа с заданным ID |
| PUT | /category/:id | Изменить название категории с заданным ID |

# Crud методы

Для каждой модели были реализованы следующие CRUD-методы:

Create: Создание новых записей в базе данных.

Read: Получение информации о существующих записях.

Update: Обновление информации о существующих записях.

Delete: Удаление существующих записей в базе данных.

Модели и методы:

1. Пользователь (User)

* Create: регистрация пользователя, логин
* Read: получение всех пользователей, проверка авторизации
* Delete: удаление аккаунта

1. Продукт (Product)

* Read: получение всех продуктов, получение продукта по ID
* Create: создание нового продукта
* Update: изменение информации о товаре
* Delete: удаление конкретного товара

1. Заказ (Order)

* Create: создание нового заказа
* Read: получение всех заказов у конретного пользователя, получение всех существующих заказов всех пользователей, получение заказа по ID
* Update: изменение статуса заказа

1. Категория (Category)

* Create: создание категории
* Read: получение всех категорий, получение категории по ID
* Update: изменение категории по ID
* Delete: удаление категории по ID

1. Корзина (Cart)

* Read: получение корзины
* Delete: очистка корзины от всех товаров

1. Предмет корзины (CartItem)

* Create: добавление товара в корзину
* Update: изменение количества товара
* Delete: удаление товара из корзины

**Настройка маршрутов и обработка запросов**

Маршруты для работы с API настроены следующим образом:

router.use("/user", userRouter);

router.use("/product", productRouter);

router.use("/category", categoryRouter);

router.use("/order", orderRouter);

router.use("/cart", cartRouter);

router.use("/cartitem", cartitem);

Каждый маршрут обрабатывает соответствующие запросы, используя контроллеры, которые взаимодействуют с сервисами и моделями.

# Тестирование API

1. Использование Postman

Для тестирования API использовался **Postman**, с помощью которого каждый маршрут проверялся на корректную работу с различными HTTP-методами (GET, POST, PUT, DELETE) и разнообразными сценариями: как с корректными данными, так и с ошибочными или неполными запросами.

Обработка ошибок в приложении реализована централизованно с использованием кастомного класса ApiError. Это позволяет возвращать понятные и структурированные ответы при возникновении исключений. В зависимости от типа ошибки клиент получает соответствующий HTTP-статус и сообщение:

return res.status(err.status).json({ message: err.message });

Если ошибка не является экземпляром ApiError, возвращается стандартный ответ с кодом 500:

return res.status(500).json({ message: "Непредвиденная и неизвестная ошибка!" });

Такой подход повышает надёжность API и упрощает отладку при разработке и тестировании.

1. Несколько примеров запросов и ответов

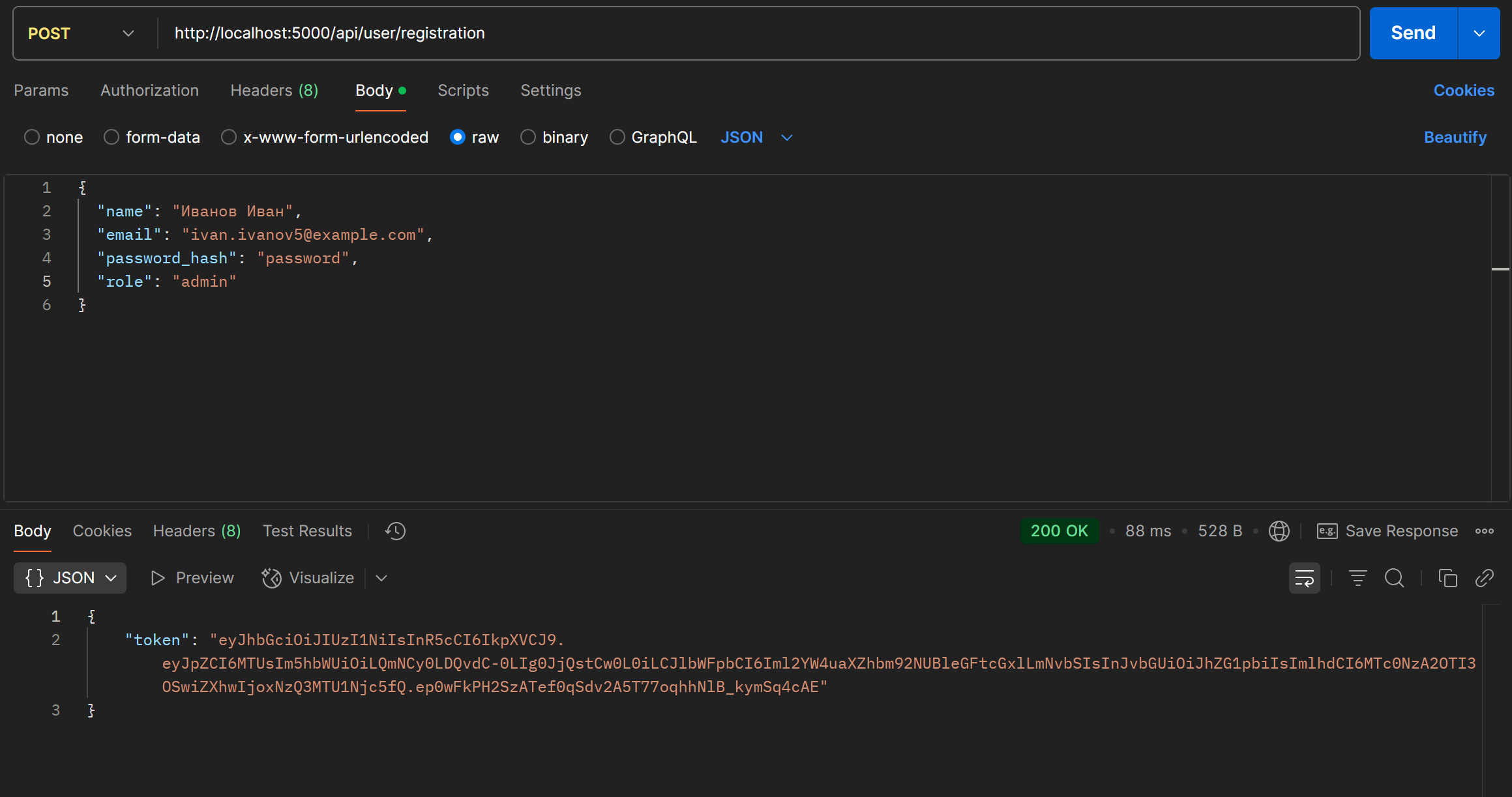


Рисунок 1 – Пример запроса и ответ на регистрацию пользователя

Поскольку был создан пользователь с ролью администратора, то воспользуемся данным токеном для последующих тестов, требующих данную роль.

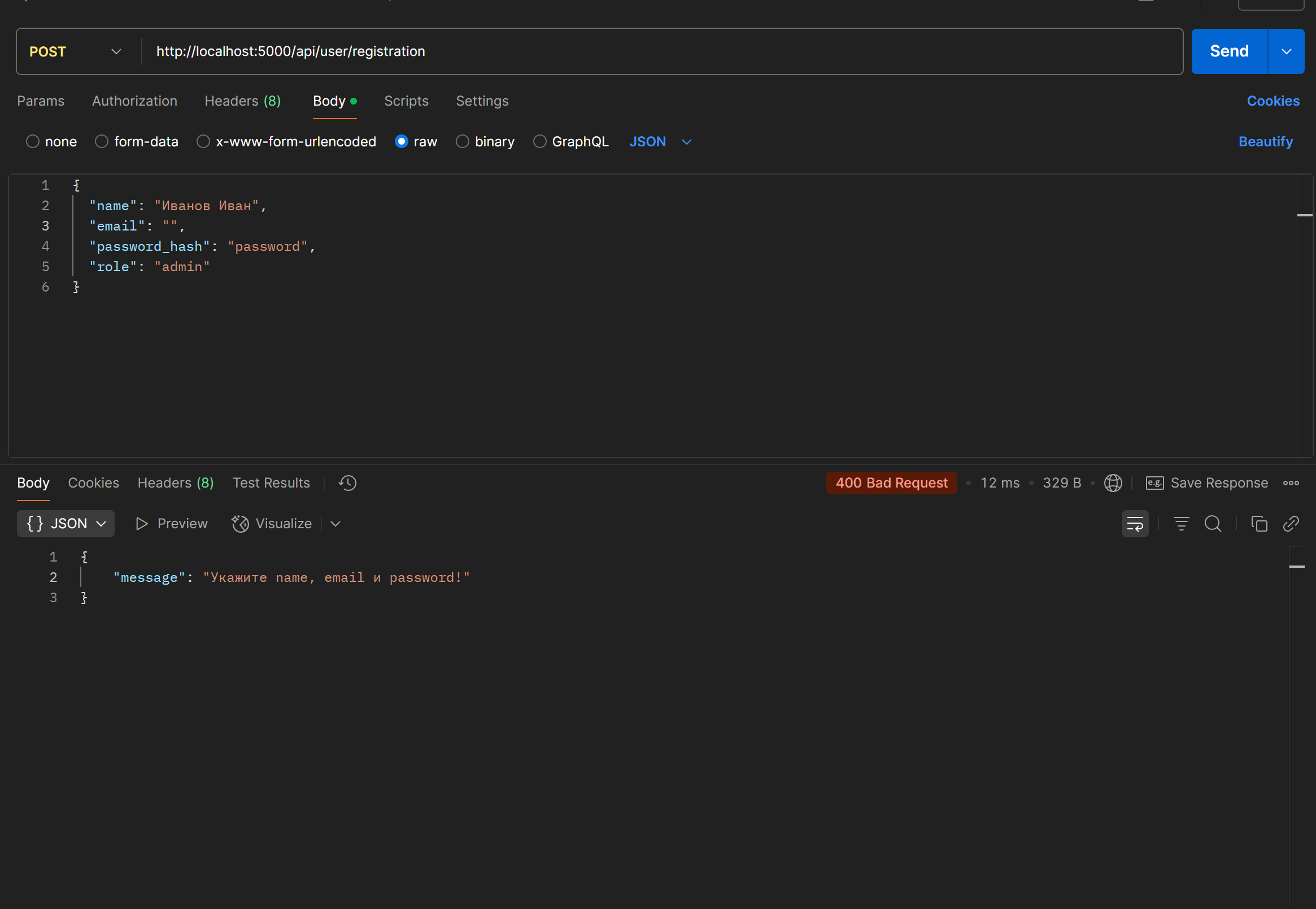


Рисунок 2 – Запрос с некорректными данными

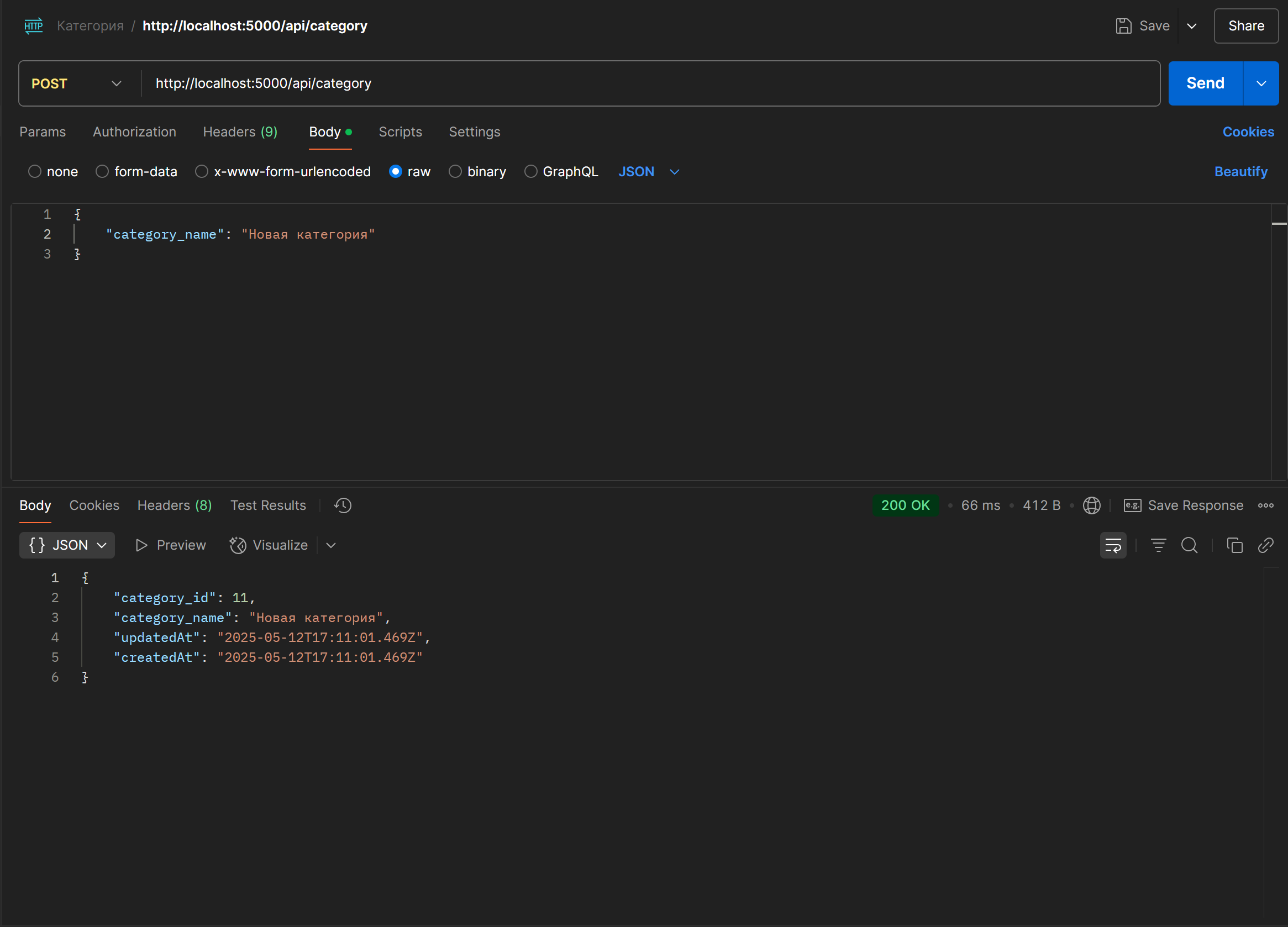


Рисунок 3 – Создание новой категории

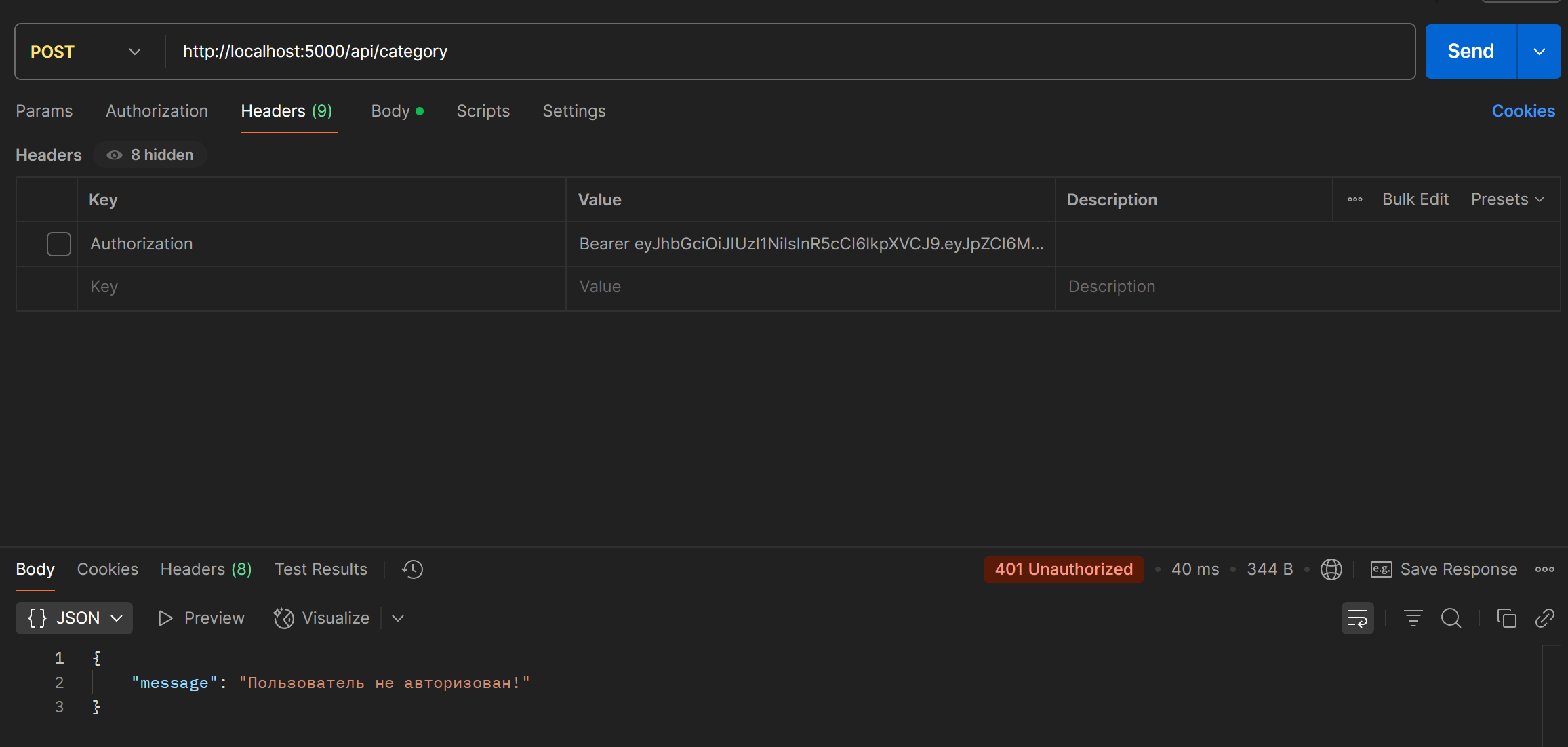


Рисунок 4 – Ответ при создании категории, когда пользователь не авторизован

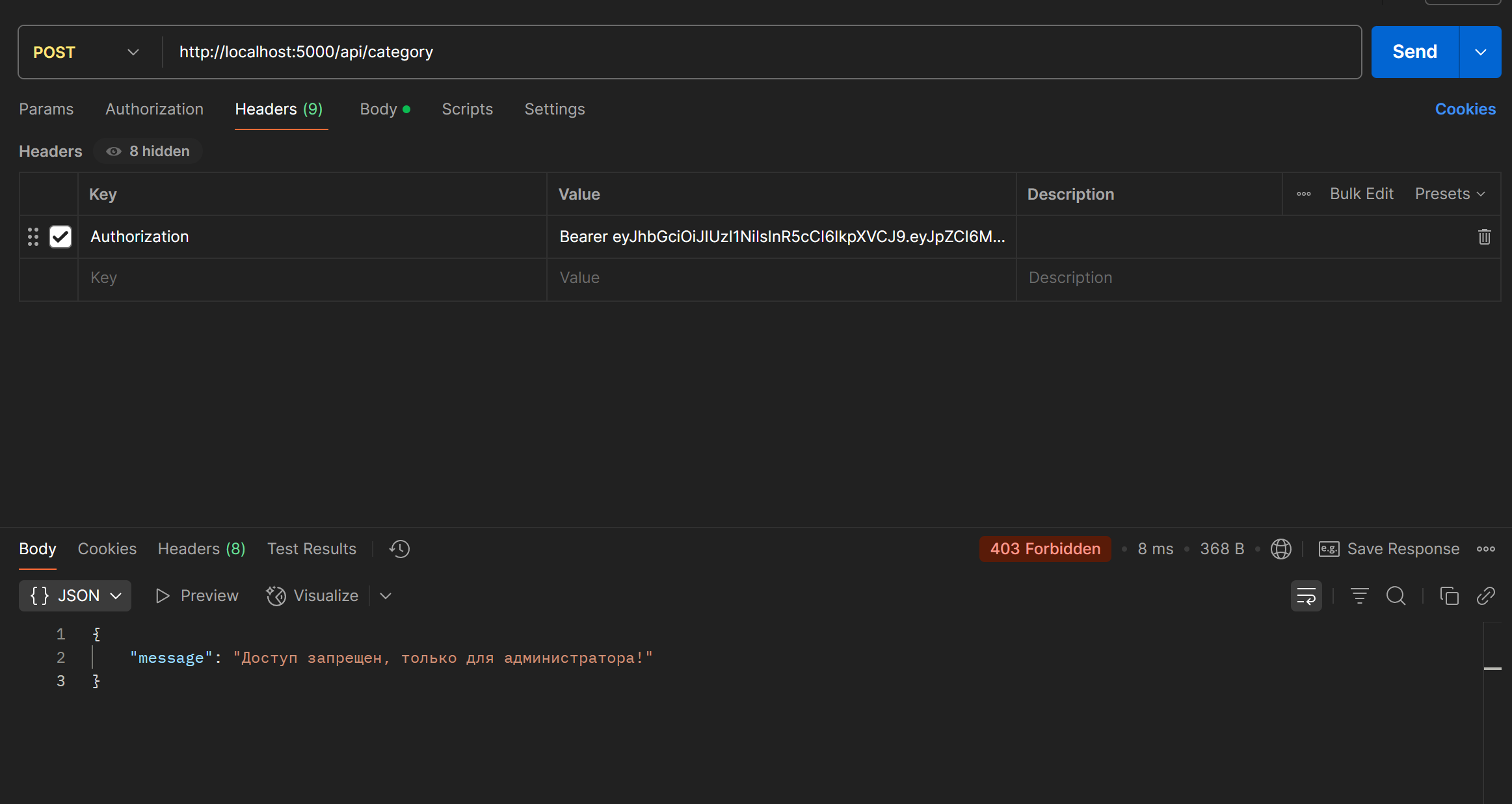


Рисунок 5 – Ответ при создании категории пользователем, который не является администратором

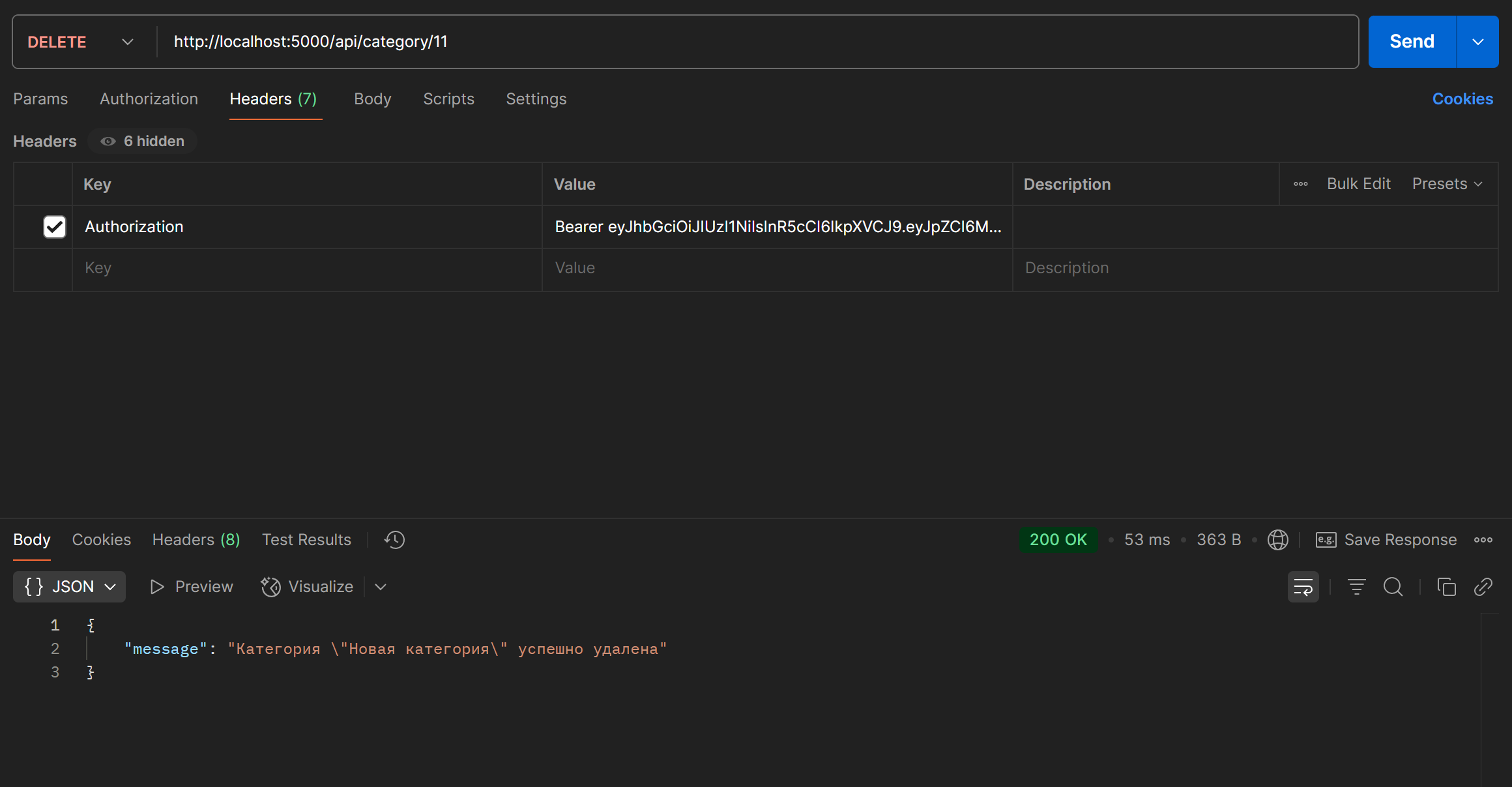


Рисунок 6 – Удаление созданной категории по ID (только для админа)

# Аутентификация и авторизация

В приложении реализована система регистрации и входа пользователей с использованием JWT-аутентификации на основе access token. При успешной авторизации пользователь получает access token, содержащий его ID, email и роль, с сроком действия 24 часа. Этот токен используется для доступа к защищённым маршрутам.

Основные элементы реализации:

1. Middleware:

* Проверка токена (аутентификация) — middleware извлекает токен из заголовка Authorization, проверяет его на валидность через jwt.verify и прикрепляет расшифрованные данные к req.user. В случае отсутствия или недействительного токена возвращается ошибка 401.
* Ролевая проверка доступа — отдельный middleware проверяет роль пользователя (например, admin) и ограничивает доступ к маршрутам, где это необходимо. При несоответствии — ошибка 403.
* Обработка ошибок — централизованно через ApiError, с понятными сообщениями и соответствующими статус-кодами.
* CORS — настроен для работы с клиентской частью из разных источников.

1. Сервисный слой

* Работа с базой данных
* Генерация токенов
* Валидация данных

1. Технологии и инструменты

* Node.js — среда выполнения
* Express.js — фреймворк для построения API
* PostgreSQL — СУБД (по стеку PERN)
* Sequelize — ORM для работы с базой данных
* JWT (jsonwebtoken) — генерация и проверка токена
* dotenv — управление переменными окружения
* cors — вспомогательные инструменты

# Структура интерфейса

Проект построен по компонентному принципу с использованием библиотеки React. Структура папок отражает разделение логики, API-запросов, состояния и маршрутизации.

**components/**

Содержит переиспользуемые компоненты интерфейса:

* AppRouter.js — настройка маршрутов приложения.
* CategoryBar.js, PaginationBar.js, ProductItem.js и тд. — элементы интерфейса магазина.
* modals/ — модальные окна (всплывающие компоненты) для взаимодействия с пользователем (например, подтверждения или уведомления).

**http/**

Здесь находятся модули для работы с API:

* cart\_cartItemAPI.js, orderAPI.js и тд. — функции для обращения к backend через axios.
* index.js — конфигурация HTTP-клиента (axios), возможно, с токеном авторизации.

**pages/**

Хранятся основные страницы, отображаемые по маршрутам:

* admin.js, auth.js, cart.js, notFound.js, order.js, shop.js.

**store/**

Состояние приложения управляется через MobX, здесь определены хранилища:

* CartStore.js, OrderStore.js, ProductStore.js, UserStore.js.

**utils/**

Служебные утилиты и константы:

* constants.js — вероятно, содержит базовые URL, значения и т.п.

**App.js**

Главная точка входа в приложение:

* Настраивается маршрутизация с помощью react-router-dom.
* Подключается логика авторизации.
* Отображаются основные и защищённые маршруты.
* Производится рендер основных компонентов.

**index.js**

Точка монтирования React-приложения в DOM.

**routes.js**

Определяет маршруты приложения (пути и связанные с ними страницы).

# Описание структуры контейнеризации и настройки окружения

Проект использует Docker и Docker Compose для упрощённого запуска и управления средой, объединяя frontend, backend и базу данных PostgreSQL в изолированные контейнеры.

**Общая архитектура контейнеров:**

**Frontend**:

* + Собирается из Node.js-образа node:22.14.0.
  + Рабочая директория: /app.
  + Зависимости устанавливаются через npm install.
  + Приложение запускается командой npm start после сборки npm run build.
  + Доступен на порту 3000.

**Backend**:

* + Также использует Node.js-образ node:22.14.0.
  + Запускается с командой npm run dev.
  + Проброшен порт 5000.

**База данных**:

* Используется официальный образ **PostgreSQL 17**.
* Переменные окружения задают имя БД, пользователя и пароль.
* Применяется volume для сохранения данных между перезапусками.
* Также предусмотрена инициализация через папку db-init, содержащую дамп данных с локального хранилища (.sql-файл в моем случае).

**Docker Compose**

docker-compose.yml оркестрирует все сервисы:

* Гарантирует порядок запуска через depends\_on.
* Объединяет сервисы в одну виртуальную сеть app\_network.

version: "3.8"

services:

  # Контейнер для backend

  backend:

    build:

      context: .

      dockerfile: backend/Dockerfile

    container\_name: backend

    environment:

      - DB\_NAME=autoparts\_shop

      - DB\_USER=postgres

      - DB\_PASSWORD=ps5694rus56

      - DB\_HOST=db

      - DB\_PORT=5432

    ports:

      - "5000:5000"

    depends\_on:

      - db

    networks:

      - app\_network

    command: npm run dev

  # Контейнер для frontend

  frontend:

    build:

      context: .

      dockerfile: frontend/Dockerfile

    container\_name: frontend

    ports:

      - "3000:3004"

    networks:

      - app\_network

    depends\_on:

      - backend

    command: npm start

  # Контейнер для базы данных PostgreSQL

  db:

    image: postgres:17

    container\_name: db

    environment:

      - POSTGRES\_DB=autoparts\_shop

      - POSTGRES\_USER=postgres

      - POSTGRES\_PASSWORD=ps5694rus56

    volumes:

      - postgres\_data:/var/lib/postgresql/data

      - ./db-init:/docker-entrypoint-initdb.d

    ports:

      - "5432:5432"

    networks:

      - app\_network

networks:

  app\_network:

    driver: bridge

volumes:

  postgres\_data:

# Работа приложения в контейнерах

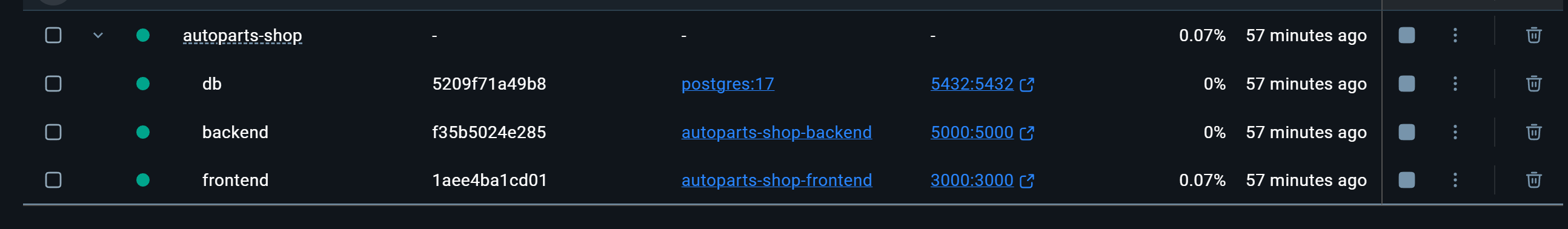


Рисунок 7 – Запущены все контейнеры в Docker Desktop

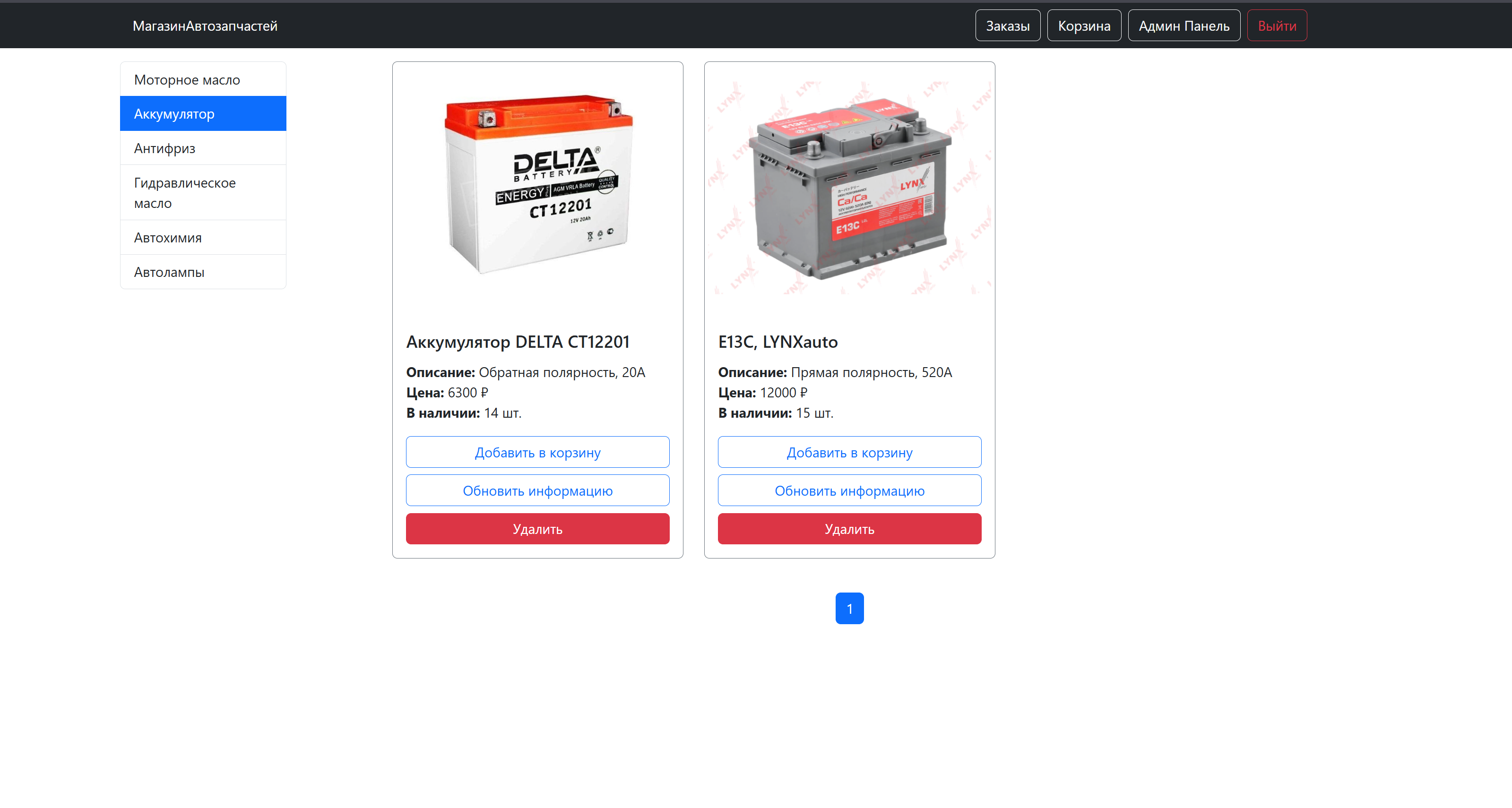
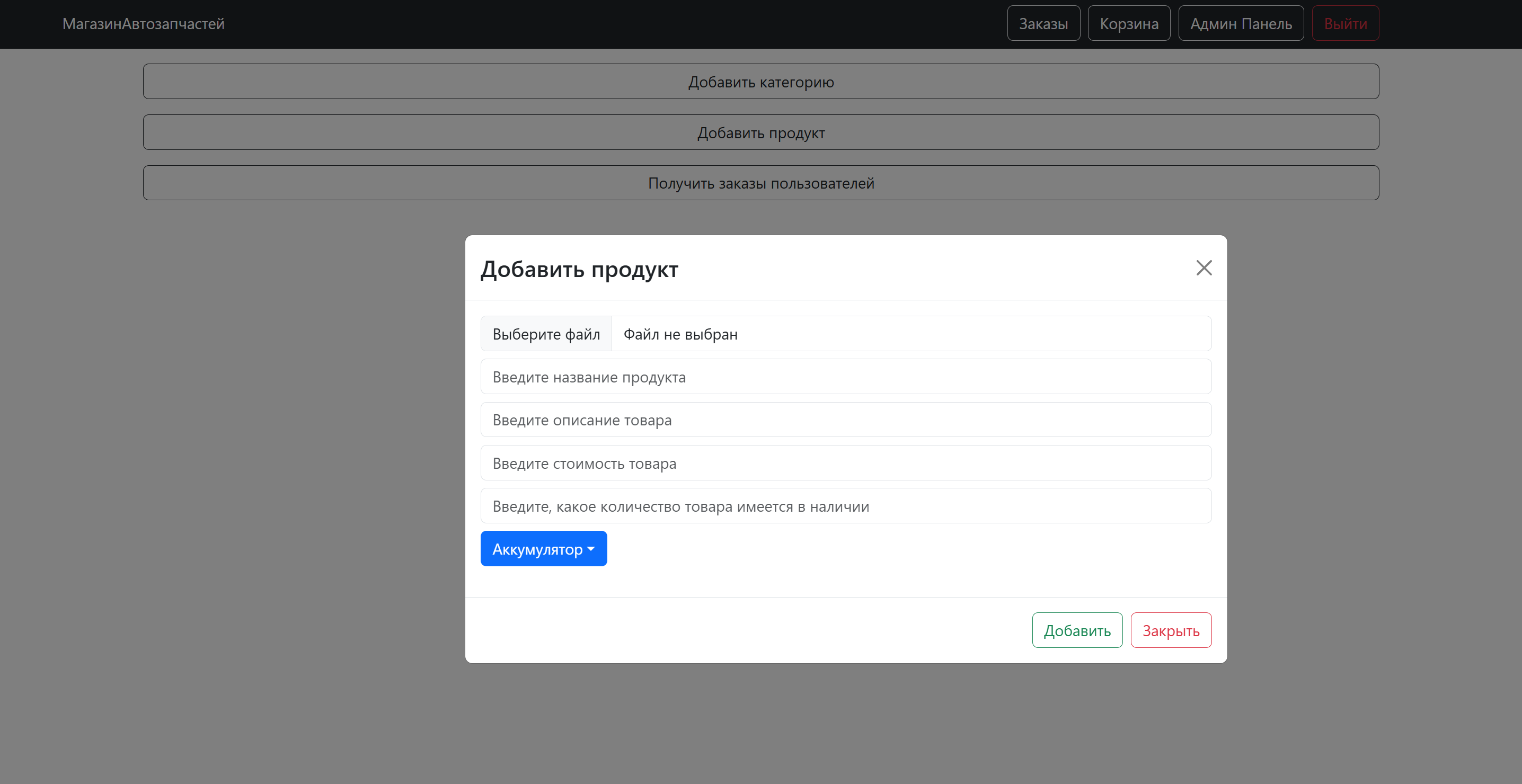


Рисунок 8 – Открыта страница для категории аккумуляторы, для пользователя с ролью администратора

  
Рисунок 9 – Пример модального окна для создания продукта

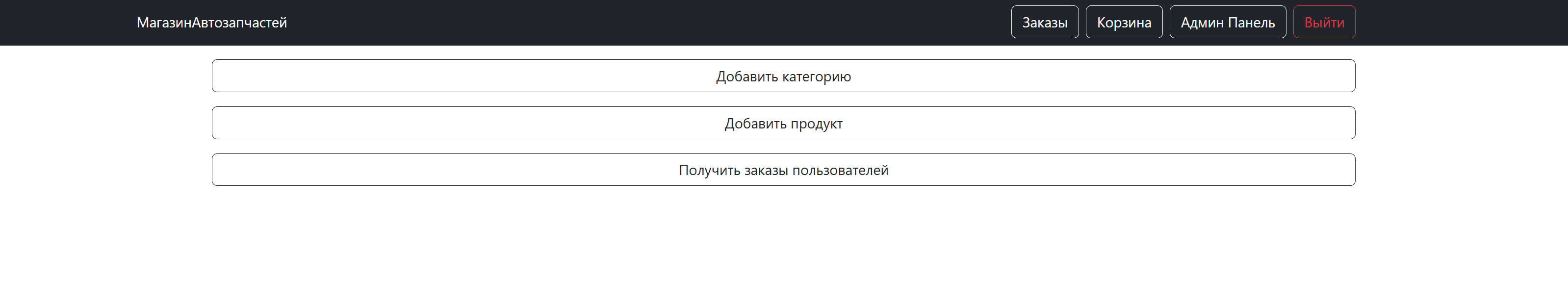


Рисунок 10 – Админ панель

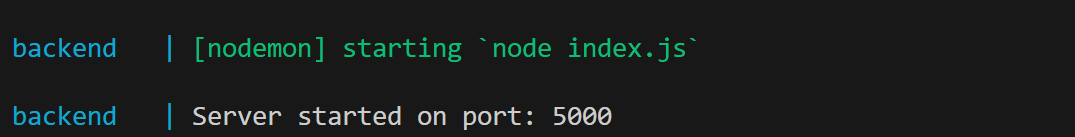


Рисунок 11 – Показывает, что backend успешно запустился на порту 3000 и подключился к БД

# Приложение А

Ссылка на GitHub со всем кодом:

<https://github.com/titonufim/Autoshop_project.git>