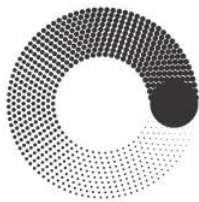


ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ



МОСКОВСКИЙ ПОЛИТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ

Факультет Информационных технологий

Кафедра Информатики и информационных технологий

направление подготовки

09.03.02 «Информационные системы и технологии»

КУРСОВОЙ ПРОЕКТ

Дисциплина: Базы Данных

Тема: Интернет-магазин с использованием базы данных

Выполнил: студент группы 221-3711

Костоваров Александр Сергеевич

(Фамилия И.О.)

Дата, подпись _____

(Дата)

(Подпись)

Проверил: _____

(Фамилия И.О., степень, звание)

Дата, подпись _____

(Дата)

(Подпись)

Замечания: _____

Москва

Содержание

| | |
|---------------------------------------|----|
| Введение | 3 |
| Глава 1 Проектирование..... | 5 |
| 1.1 Описание предметной области | 5 |
| 1.2 Выбор стека технологий | 5 |
| Глава 2 Разработка | 7 |
| 2.1 База данных | 7 |
| 2.2 Серверная часть (Back-End) | 11 |
| 2.3 Клиентская часть (Frontend) | 18 |
| Заключение | 36 |
| Список Литературы | 37 |
| Приложение А | 38 |
| Приложение Б..... | 41 |
| Приложение В | 44 |
| Приложение Г | 49 |
| Приложение Д | 53 |

Введение

В рамках курса "Базы данных" была выполнена курсовая работа на тему "Интернет-магазин с использованием базы данных". Целью данной работы стало создание прототипа интернет-магазина, который включает в себя ключевые функции современных онлайн-магазинов, такие как каталог товаров, корзина, регистрация и авторизация пользователей, а также инструменты для управления товарами для администратора. Актуальность проекта обусловлена тем, что интернет-магазины стали важной частью электронной коммерции, обеспечивая удобство покупок для клиентов и продавцов. Структура интернет-магазина является наглядным примером применения баз данных в разработке информационных систем. Создание такого прототипа способствует приобретению практических навыков работы с базами данных и основ веб-разработки.

Определим цели данной курсовой работы:

1. Создание учебного интернет-магазина, который включает в себя основные компоненты и функции современного онлайн-магазина.
2. Получение практических навыков работы с базами данных, вебразработки и управления данными.

Задачи:

1. Создание базы данных:
 - Спроектировать структуру базы данных для хранения информации о пользователях, товарах, категориях товаров и корзинах покупок.
 - Создать ER-диаграмму и определить связи между таблицами.
2. Создание серверной части:
 - Реализовать серверную платформу.
 - Обеспечить обработку запросов от клиента и взаимодействие с базой данных.

- Реализовать функции регистрации и авторизации пользователей.
- Обеспечить возможность добавления, редактирования и удаления товаров для администратора.

3. Разработка клиентской части приложения:

- Разработать удобный интерфейс для взаимодействия пользователя с магазином
- Реализовать отображение каталога товаров, страницу товара, корзину покупок и оформление заказа.
- Обеспечить возможность загрузки и отображения изображений товаров.

4. Тестирование и интеграция

- Провести интеграцию клиентской и серверной частей.
- Провести тестирование функциональности интернетмагазина.
- Исправить выявленные ошибки и улучшить производительность.

Глава 1 Проектирование

1.1 Описание предметной области

Создание интернет-магазина является одним из наиболее востребованных направлений в веб-разработке. Такие платформы позволяют пользователям совершать покупки онлайн, предоставляя широкий выбор товаров и услуг. В данной работе рассматривается процесс создания интернет-магазина с акцентом на изучение и применение современных технологий веб-разработки и проектирования баз данных.

Основные аспекты предметной области

Интернет-магазины представляют собой сложные информационные системы, включающие такие компоненты, как каталог товаров, управление пользователями, корзина покупок, система оплаты и другие. Все эти элементы связаны между собой и взаимодействуют для обеспечения удобства и безопасности онлайн-покупок.

Одной из ключевых задач при создании интернет-магазина является обеспечение надежного и корректного хранения данных. В проекте используется реляционная база данных для эффективного управления информацией о пользователях, товарах и заказах.

Пользователи системы

Проект включает две основные категории пользователей

- Покупатели – пользователи, которые выбирают и покупают товары. Для них предусмотрены функции регистрации, авторизации, просмотра каталога, добавления товаров в корзину и оформления заказов.
- Администраторы – пользователи с расширенными правами доступа. Они управляют ассортиментом товаров, обрабатывают заказы и обеспечивают функционирование интернет-магазина. Администраторы могут добавлять, редактировать и удалять товары, а также просматривать статистику продаж.

Структура системы

Основными компонентами интернет-магазина являются:

- Каталог товаров – отображает информацию о доступных товарах, включая их характеристики и цены. Для хранения данных о товарах используется база данных, где содержится информация о названии, описании, категории, цене и наличии на складе.
- Корзина покупок – временное хранилище для выбранных товаров перед оформлением заказа. Она позволяет пользователям добавлять, удалять и изменять количество товаров.
- Система управления пользователями – включает регистрацию, авторизацию и управление профилем. Особое внимание уделяется безопасности данных, особенно паролей, для чего используются методы хеширования.
- Административная панель – интерфейс для администраторов, который предоставляет возможности управления каталогом товаров, обработки заказов и просмотра статистики для принятия управленческих решений.

1.2 Выбор стека технологий

После того, как цели и задачи на работу были поставлены, был проведен выбор стека используемых технологий. Для разработки серверной части мной был выбран веб-фреймворк ASP.NET Core с версией .NET 8.0 данный фреймворк был изучен мной в ходе освоения курса Back-end разработки и представляет собой удобный инструмент для создания кросс платформенных веб приложений, работой с API и HTTP запросами. Так же для совмещения базы данных с работой серверной части бы установлен фреймворк Entity Framework Core - это ORM (Object-Relational Mapping) от Microsoft, который упрощает взаимодействие с базой данных путем автоматического сопоставления объектов в коде с таблицами в базе данных, что упрощает и систематизирует написания запросов к БД. Для управления базой данных мной была выбрана СУБД MySQL т.к она легко интегрируется с ASP NET

Core. Для разработки клиентской же части мой выбор был сделан в пользу библиотеки React, React - это JavaScript-библиотека для построения пользовательских интерфейсов, а в качестве языка программирования TypeScript - это язык программирования, расширяющий возможности JavaScript за счет введения статической типизации. Так же в проект были интегрированы 3 вспомогательные библиотеки. Material-UI: Библиотека компонентов для React, обеспечивающая простое создание интерфейсов для более удобного написания и читаемости кода. Axios: Библиотека для выполнения HTTP-запросов из клиентской части к серверу. React Router: Библиотека для управления маршрутизацией в React-приложениях. В качестве среды разработки были выбраны Microsoft Visual Studio (серверная часть) и Visual Studio Code, а также создан репозиторий на платформе GitHub.

Глава 2 Разработка

2.1 База данных

В данном разделе разберем созданную базу данных, таблицы, их атрибуты и связи между сущностями. Как уже было описано в качестве СУБД была выбрана MySQL, в MySQL Workbench была создана EER диаграмма для упрощения проектированной базы данных и настройки связей.

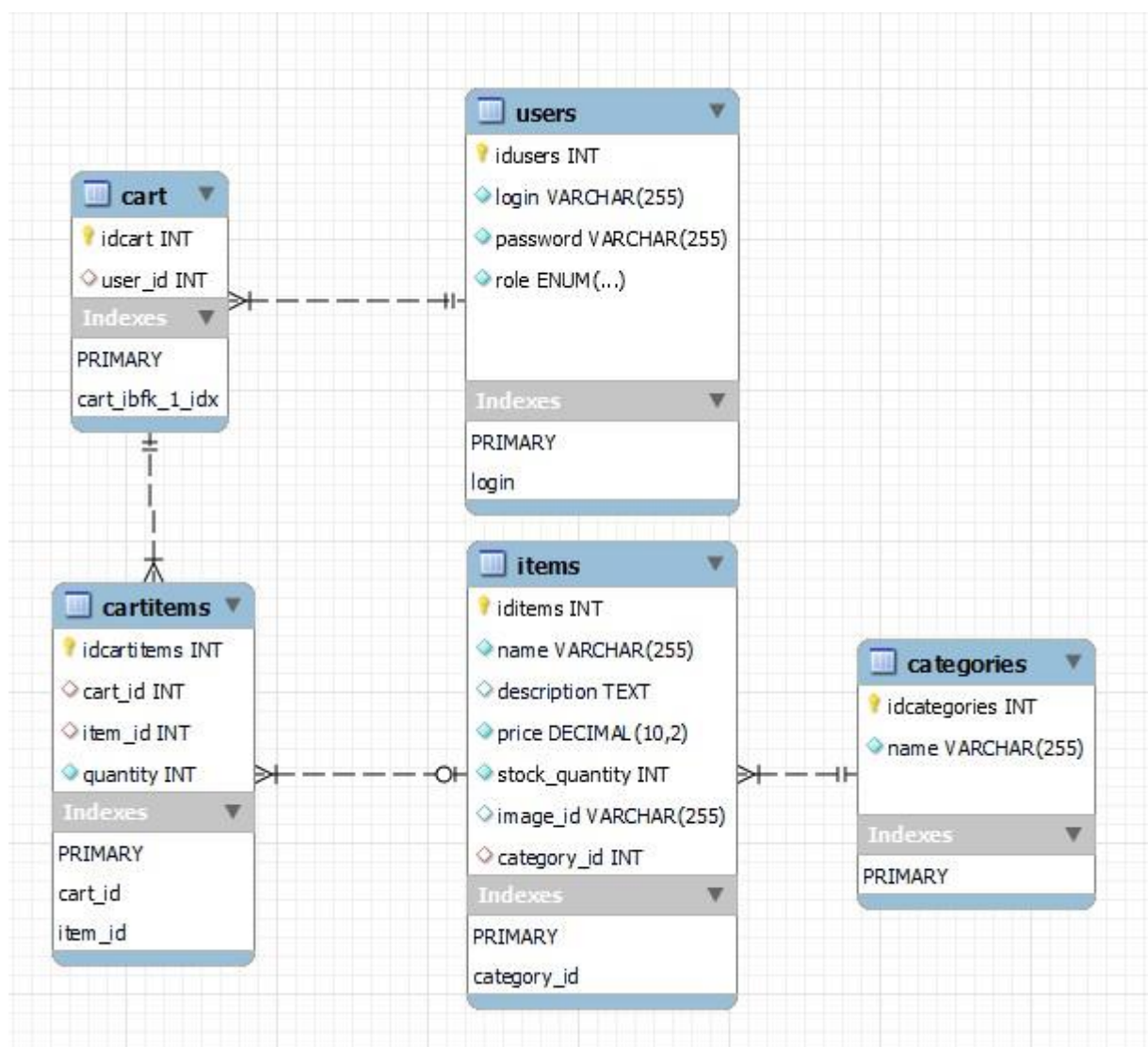


Рисунок 2.1 EER диаграмма базы данных

Данная диаграмма наглядно иллюстрирует структуру базы данных для хранения всех необходимых в разработке интернет магазина данных. Разберем подробнее каждый элемент.

Таблица users хранит информацию о пользователях интернет-магазина. Она содержит следующие атрибуты:

1. idusers (INT): Первичный ключ, уникальный идентификатор пользователя.
2. login (VARCHAR(255)): Логин пользователя, используется для аутентификации.

3. password (VARCHAR(255)): Пароль пользователя, используется для аутентификации.
4. role (ENUM): Роль пользователя, может принимать значения 'admin' или 'user'.

Таблица categories хранит информацию о категориях товаров. Она содержит следующие атрибуты:

1. idcategories (INT): Первичный ключ, уникальный идентификатор категории.
2. name (VARCHAR(255)): Название категории.

Таблица items хранит информацию о товарах. Она содержит следующие атрибуты:

1. iditems (INT): Первичный ключ, уникальный идентификатор товара.
2. name (VARCHAR(255)): Название товара.
3. description (TEXT): Описание товара.
4. price (DECIMAL(10,2)): Цена товара.
5. stock_quantity (INT): Количество товара на складе.
6. image_id (VARCHAR(255)): Путь к изображению товара.
7. category_id (INT): Внешний ключ, указывающий на категорию товара.

Таблица cart хранит информацию о корзинах пользователей. Она содержит следующие атрибуты:

1. idcart (INT): Первичный ключ, уникальный идентификатор корзины.
2. user_id (INT): Внешний ключ, указывающий на пользователя, которому принадлежит корзина.

Таблица cartitems хранит информацию о товарах в корзинах. Она содержит следующие атрибуты:

1. `idcartitems` (INT): Первичный ключ, уникальный идентификатор записи в корзине.
2. `cart_id` (INT): Внешний ключ, указывающий на корзину.
3. `item_id` (INT): Внешний ключ, указывающий на товар.
4. `quantity` (INT): Количество товара в корзине.

Связи между таблицами определены следующий образом:

- `Users` и `Carts`: Один пользователь может иметь одну корзину. Связь осуществляется через атрибут `user_id` в таблице `cart`, который является внешним ключом, ссылающимся на атрибут `idusers` в таблице `users`.
- `Categories` и `Items`: Одна категория может включать множество товаров. Связь осуществляется через атрибут `category_id` в таблице `items`, который является внешним ключом, ссылающимся на атрибут `idcategories` в таблице `categories`.
- `Carts` и `CartItems`: Одна корзина может содержать множество записей о товарах. Связь осуществляется через атрибут `cart_id` в таблице `cartitems`, который является внешним ключом, ссылающимся на атрибут `idcart` в таблице `cart`.
- `Items` и `CartItems`: Один товар может быть включен во множество записей корзины. Связь осуществляется через атрибут `item_id` в таблице `cartitems`, который является внешним ключом, ссылающимся на атрибут `iditems` в таблице `items`.

Данная структура предоставляет собой хорошую основу для разработки прототипа интернет магазина, имеющего все типичные для такой системы функции.

2.2 Серверная часть (Back-End)

После создания базы данных началась разработка серверной части приложения. Первым шагом, после настройки среды разработки и установки необходимых пакетов через NuGet, стало создание контекста базы данных. В приложении ASP.NET Core с использованием Entity Framework Core контекст базы данных играет ключевую роль, управляя подключением к базе данных и предоставляя API для работы с данными. Он представляет собой класс, производный от DbContext, и включает наборы сущностей, которые соответствуют таблицам базы данных. Для этого создаются модели данных, которые повторяют структуру таблиц в базе данных. Рассмотрим структуру на примере модели данных для таблицы items.

```

namespace coursework.Models
{
    [Table("Items")]
    Ссылка: 6
    public class Item
    {
        [Key]
        [Column("iditems")]
        Ссылка: 3
        public int Id { get; set; }

        [Required]
        [Column("name")]
        Ссылка: 0
        public string Name { get; set; }

        [Column("description")]
        Ссылка: 0
        public string Description { get; set; }

        [Column("price")]
        Ссылка: 0
        public decimal Price { get; set; }

        [Column("stock_quantity")]
        Ссылка: 2
        public int Stock { get; set; }

        [Column("image_id")]
        Ссылка: 0
        public string ImagePath { get; set; }

        [Required]
        [Column("category_id")]
        Ссылка: 0
        public int CategoryId { get; set; }
    }
}

```

Листинг 2.1 модель данных items

Данный фрагмент демонстрирует общую структуру реализации моделей данных для создания контекста БД. Поля класса, их типы данных и название класса должны в точности совпадать с названиями и типами в таблице для автоматической связи, но в нашем случае если использовать названия отличные от тех, что находятся в БД, как в нашем случае для удобства написания некоторых из них, нужно использовать атрибуты Table и Column для точного указания к какой таблице и атрибуту относится то или иное поле, при этом типы данных должны все так же совпадать. Общая структура каталога Models выглядит следующим образом:

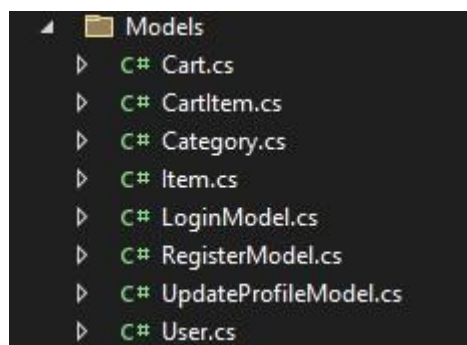


Рисунок 2.1 Структура каталога для хранения моделей данных

В нем содержатся модели для каждой сущности, а также 2 вспомогательные, используемые в контроллерах, для регистрации и обновления информации в профиле, чтобы ограничить прямое взаимодействие с моделью, содержащей полную информацию о пользователе. Теперь создаем сам контекст и подключение к базе данных, контекст создается в отдельном скрипте, с классом ShopContext, который наследуется от класса DbContext.

Подключение же к базе данных можно настроить разными способами, но в нашем случае добавим его в классе Program, для этого указывается сервер, порт, имя пользователя, имя БД и пароль подключения.

```
builder.Services.AddDbContext<ShopContext>(options =>
    options.UseMySQL("Server=127.0.0.1;Port=3306;Database=shop;Uid=root;Pwd=pugpug12;",
        new MySqlServerVersion(new Version(8, 0, 36))));
```

Листинг 2.2 подключение к базе данных

```
using coursework.Models;
using Microsoft.EntityFrameworkCore;

Ссылка: 11
public class ShopContext : DbContext
{
    Ссылка: 5
    public DbSet<User> Users { get; set; }
    Ссылка: 1
    public DbSet<Category> Categories { get; set; }
    Ссылка: 9
    public DbSet<Item> Items { get; set; }
    Ссылка: 6
    public DbSet<Cart> Carts { get; set; }
    Ссылка: 9
    public DbSet<CartItem> CartItems { get; set; }

    Ссылка: 0
    public ShopContext(DbContextOptions<ShopContext> options) : base(options) { }

    Ссылка: 0
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);
    }
}
```

Листинг 2.3 контекст базы данных

В контексте указываются все модели, относящиеся к базе данных для добавления в него, а в конструкторе указываются параметры, так как мы настроили их в файле Program (а именно подключение и версию MySQL), то здесь указываем их как options. После создания основы для работы с базой данных, были созданы контроллеры. Созданные API контроллеры будут служить для получения и обработки запросов с клиентской стороны приложения, всего их 5, каждый имеет свою четкую функцию: Работа с данными пользователя, данными товаров, корзиной, изображениями и категориями товаров. Рассмотрим их функционал.

```
[ApiController]
[Route("api/[controller]")]
Ссылка 1
public class CategoriesController : ControllerBase
{
    private readonly ShopContext _context;

    Ссылка 0
    public CategoriesController(ShopContext context)
    {
        _context = context;
    }

    // GET: api/Categories
    [HttpGet]
    Ссылка 0
    public async Task<ActionResult<IEnumerable<Category>>> GetCategories()
    {
        return await _context.Categories.ToListAsync();
    }
}
```

Листинг 2.4 Контроллер категорий

Самый мало функциональный из всех, это контроллер отвечающий за передачу списка всех категорий товаров, чтобы использовать его для на стороне клиента для отображения правильной категории товара или присвоение во время создания нового. Данные Get запрос выведен в отдельный контроллер, чтобы разгрузить довольно объемный ItemsController от еще большего количества методов, его мы и разберем следующим. Так как данный контроллер содержит большое количество запросов, полный код содержится в приложении А. Рассмотрим все типы запросов, которые способен обработать данный котроллер и их принцип работы.

Получение списка всех товаров (GET: `api/Items`): Возвращает список всех товаров из таблицы `Items` базы данных. Метод асинхронно выполняет запрос к базе данных и возвращает все записи в виде списка.

Получение товара по идентификатору (GET: `api/Items/{id}`): Возвращает данные о конкретном товаре, используя его идентификатор. Если товар не найден, возвращается статус 404 (Not Found). Метод асинхронно ищет товар в базе данных по идентификатору.

Добавление нового товара (POST: `api/Items`): Принимает данные нового товара и добавляет его в таблицу `Items` базы данных. Проверяет валидность модели, сохраняет изменения в базе данных и возвращает созданный товар с HTTP статусом 201 (Created) и ссылкой на созданный товар.

Обновление существующего товара (PUT: `api/Items/{id}`): Принимает идентификатор товара и обновленные данные товара. Проверяет, что идентификатор в запросе совпадает с идентификатором в данных товара. Обновляет данные в базе данных, сохраняет изменения. Если товар не найден, возвращает статус 404 (Not Found). В случае успешного обновления возвращает статус 204 (No Content).

Удаление товара (DELETE: `api/Items/{id}`): Принимает идентификатор товара и удаляет его из таблицы `Items` базы данных. Если товар не найден, возвращает статус 404 (Not Found). Удаляет товар из базы данных, сохраняет изменения и возвращает статус 204 (No Content).

Проверка существования товара (приватный метод): Проверяет наличие товара с заданным идентификатором в таблице `Items`. Возвращает `true`, если товар существует, и `false` в противном случае. Используется в методе `UpdateItem` для проверки существования товара перед обновлением данных.

Таким образом данный контроллер позволяет осуществлять CRUD (create, read, upload, delete) операции с товарами из базы данных. Следующим идет контроллер, осуществляющий контроль над учетной записью пользователя, авторизация, редактирование данных, вход и т.д, полный код представлен в приложении Б, а ниже рассмотрим все возможные действия:

Регистрация нового пользователя (POST: api/Account/Register): Принимает данные для регистрации нового пользователя. Проверяет валидность модели и наличие пользователя с таким же логином. Если пользователь уже существует, возвращает статус 409 (Conflict). Если данные валидны, создает нового пользователя с ролью "user", сохраняет его в базе данных и возвращает сообщение об успешной регистрации и идентификатор пользователя.

Вход пользователя (POST: api/Account/Login): Принимает данные для входа пользователя. Проверяет валидность модели и наличие пользователя с указанным логином и паролем. Если пользователь не найден или данные неверны, возвращает статус 401 (Unauthorized) и логирует предупреждение. Если данные валидны, возвращает роль и идентификатор пользователя.

Выход пользователя (POST: api/Account/Logout): Обработывает запрос на выход пользователя. Логирует успешный выход и возвращает статус 200 (OK) с сообщением об успешном выходе.

Обновление профиля пользователя (PUT: api/Account/{id}): Принимает идентификатор пользователя и данные для обновления профиля. Проверяет валидность модели и наличие пользователя с указанным идентификатором. Если пользователь не найден, возвращает статус 404 (Not Found). Если данные валидны, обновляет логин и пароль пользователя в базе данных, сохраняет изменения и возвращает сообщение об успешном обновлении профиля.

Далее рассмотрим контроллер для работы с изображениями, изображения для товаров, хранятся в папке wwwroot, откуда легко можно получить

изображения для фронтэнд части, при обновлении или добавлении товара, данный контроллер сохраняет путь к изображению для дальнейшего внесения его в базу данных, откуда его сможет использовать React приложение.

```
[ApiController]
[Route("api/[controller]")]
Ссылка: 0
public class ImageUploadController : ControllerBase
{
    [HttpPost]
    Ссылка: 0
    public async Task<IActionResult> Upload(IFormFile file)
    {
        if (file == null || file.Length == 0)
            return BadRequest("No file uploaded");

        var filePath = Path.Combine("wwwroot/images", file.FileName);

        using (var stream = new FileStream(filePath, FileMode.Create))
        {
            await file.CopyToAsync(stream);
        }

        var relativePath = $"/images/{file.FileName}";
        return Ok(new { filePath = relativePath });
    }
}
```

Листинг 2.5 контроллер загрузки изображений

Последним контроллером, завершающим функциональность бэкэнда, является контроллер управления корзиной пользователя, его код представлен в приложении В, а функциональность каждого запроса разберем ниже:

Получение всех товаров в корзине пользователя (GET: api/Cart/{userId}): Возвращает список всех товаров, добавленных в корзину указанного пользователя. Асинхронно выполняет запрос к базе данных для получения корзины пользователя и всех товаров в этой корзине. Если корзина не найдена, возвращает пустой список.

Добавление товара в корзину (POST: api/Cart/{itemId}): Добавляет указанный товар в корзину пользователя. Идентификатор пользователя извлекается из заголовка авторизации. Если корзины у пользователя нет, она создается. Если товар уже есть в корзине, его количество увеличивается на 1, иначе товар добавляется в корзину с количеством 1.

Удаление товара из корзины (DELETE: api/Cart/{itemId}): Удаляет указанный товар из корзины пользователя. Идентификатор пользователя

извлекается из заголовка авторизации. Если корзина или товар в корзине не найдены, возвращается соответствующий статус ошибки.

Очистка корзины (DELETE: `api/Cart/clear/{userId}`): Удаляет все товары из корзины указанного пользователя. Если корзина не найдена, возвращает статус 404 (Not Found). Если корзина найдена, все товары из неё удаляются, и корзина очищается.

Оформление заказа (POST: `api/Cart/checkout`): Обработывает оформление заказа для пользователя. Идентификатор пользователя извлекается из заголовка авторизации. Проверяет наличие корзины и достаточность каждого товара в корзине. Если товара недостаточно на складе, возвращает сообщение об ошибке. Если все проверки пройдены, уменьшает количество каждого товара на складе на количество, указанное в корзине, очищает корзину и возвращает сообщение об успешном оформлении заказа.

Таким образом были выполнены все задачи, относящиеся к бэкэнд части, приложение теперь имеет действия для обработки всех необходимых для функционирования запросов.

2.3 Клиентская часть (Frontend)

Завершающей частью разработки, стало создание react приложения для клиентской части интернет магазина, в ней были реализованы все необходимые страницы, запросы к серверу, а также пользовательский интерфейс. Клиентская часть взаимодействует с серверной частью через APIзапросы, выполненные с использованием библиотеки `axios`. Это позволяет эффективно управлять данными и обеспечивать обновление интерфейса в реальном времени. Структура приложения выглядит следующим образом:

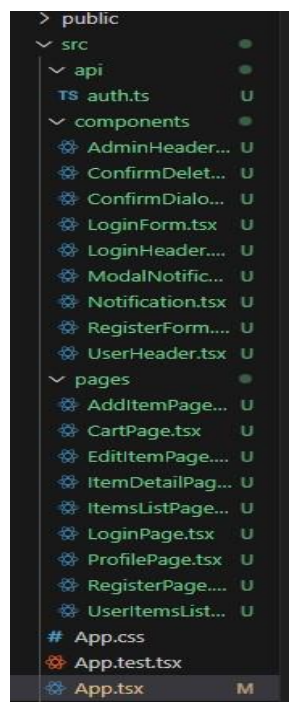


Рисунок 2.1 Структура приложения

Приложение разделено на модули, включающие в себя страницы и отдельные компоненты, такие как формы для ввода, хэдеры и компоненты уведомлений.

Компонент App представляет собой основную точку входа в React-приложение. Он использует библиотеку react-router-dom для управления маршрутизацией и отображения различных страниц в зависимости от роли пользователя. В него импортируются необходимые компоненты страниц и заголовков, которые будут использоваться для отображения содержимого и навигации.

```
App.tsx > ...
import React, { useEffect, useState } from 'react';
import { BrowserRouter as Router, Route, Routes, Navigate } from 'react-router-dom';
import LoginPage from './pages/LoginPage';
import RegisterPage from './pages/RegisterPage';
import ItemsListPage from './pages/ItemsListPage';
import AddItemPage from './pages/AddItemPage';
import EditItemPage from './pages/EditItemPage';
import UserItemsListPage from './pages/UserItemsListPage';
import ItemDetailPage from './pages/ItemDetailPage';
import CartPage from './pages/CartPage';
import ProfilePage from './pages/ProfilePage';
import AdminHeader from './components/AdminHeader';
import UserHeader from './components/UserHeader';
import LoginHeader from './components/LoginHeader';
```

Листинг 2.1 Импорт компонентов в app.tsx

Далее используется хук `useState` для хранения роли пользователя и хук `useEffect` для установки роли пользователя при загрузке приложения. Роль пользователя так же сохранена в локальном хранилище. После чего идет настройка маршрутизации, которая в зависимости от роли пользователя позволяет заходить на определенные страницы и отображает соответствующий хэдер для навигации, хэдеров всего 3 типа, для неавторизованного пользователя (только название сайта), для пользователя (каталог, корзина, профиль, выход) и для администратора (редактирование, добавление, выход).

```
const App: React.FC = () => {
  const [userRole, setUserRole] = useState<string | null>(localStorage.getItem('userRole'));

  useEffect(() => {
    const role = localStorage.getItem('userRole');
    if (role) {
      setUserRole(role);
    }
  }, []);
};
```

Листинг 2.2 хуки для управление ролями

Перейдем к авторизации, за данный процесс отвечают несколько компонентов, первый из них это модуль `auth.ts` он содержит 2 функции POST для регистрации и входа с использованием `axios`.

```
interface AuthResponse {
  message: string;
  role?: string;
  userId?: string; // Добавляем userId
}

export const register = async (login: string, password: string): Promise<AuthResponse> => {
  const response = await axios.post<AuthResponse>(`${API_URL}/Register`, { login, password });
  return response.data;
};

export const login = async (login: string, password: string): Promise<AuthResponse> => {
  const response = await axios.post<AuthResponse>(`${API_URL}/Login`, { login, password });
  return response.data;
};
```

Листинг 2.3 модуль отправки запросов регистрации и авторизации

Далее рассмотрим компонент `LoginForm` представляющей форму для входа, этот компонент отображает форму для ввода данных и вызывает функцию входа из `auth` для аутентификации при отправке формы (`handleSubmit`). В случае положительного ответа от сервера, айди пользователя сохраняется в локальном хранилище, чтобы затем корректно синхронизировать

пользователя и его корзину. Так же в зависимости от роли, используется перенаправление на страницы администратора, где дается инструментарий управления товарами или пользователя (интерфейс покупок и просмотра).

```
const LoginForm: React.FC = () => {
  const [loginValue, setLoginValue] = useState('');
  const [password, setPassword] = useState('');
  const [notification, setNotification] = useState({ open: false, message: '', severity: 'success' as 'success' | 'error' | 'warning' | 'info' });
  const navigate = useNavigate();

  const handleSubmit = async (event: React.FormEvent) => {
    event.preventDefault();
    try {
      const response = await login(loginValue, password);
      console.log('Login response:', response); // Логируем ответ сервера для проверки
      if (response.userId) {
        localStorage.setItem('userId', response.userId.toString());
      }
      localStorage.setItem('userRole', response.role || '');
      if (response.role === 'admin') {
        navigate('/items');
      } else {
        navigate('/user-items');
      }
      setNotification({ open: true, message: `Logged in as ${response.role}`, severity: 'success' });
    } catch (err) {
      console.error('Login error:', err);
      setNotification({ open: true, message: 'Invalid login or password', severity: 'error' });
    }
  };

  const handleNotificationClose = () => {
    setNotification({ ...notification, open: false });
  };
};
```

Листинг 2.4 Функции формы входа

Визуальное отображение данной формы в свою очередь выполнено с использованием Material-UI, рассмотрим, как выглядит подобная форма, так как похожая структура используется на многих страницах, таких как регистрация, добавление или редактирование товара, но со своими специфическими полями ввода.

```

return [
  <Container component="main" maxWidth="xs">
    <Box component="form" onSubmit={handleSubmit} sx={{ mt: 1 }}>
      <Typography component="h1" variant="h5">
        Login
      </Typography>
      <TextField
        margin="normal"
        required
        fullWidth
        label="Login"
        name="login"
        value={loginValue}
        onChange={(e) => setLoginValue(e.target.value)}
        autoFocus
      />
      <TextField
        margin="normal"
        required
        fullWidth
        name="password"
        label="Password"
        type="password"
        value={password}
        onChange={(e) => setPassword(e.target.value)}
      />
      <Button type="submit" fullWidth variant="contained" sx={{ mt: 3, mb: 2 }}>
        Login
      </Button>
      <Notification
        open={notification.open}
        message={notification.message}
        severity={notification.severity}
        onClose={handleNotificationClose}
      />
    </Box>
  </Container>
];

```

Листинг 2.5 визуальное оформление формы входа

Последним же звеном становится страница для входа, она отображает

Loginform для полей ввода данных и сохраняет в локальном хранилище

```

const LoginPage: React.FC<LoginPageProps> = ({ onLogin }) => {
  const [login, setLogin] = useState('');
  const [password, setPassword] = useState('');
  const navigate = useNavigate();

  const handleLogin = async () => {
    try {
      const response = await axios.post('https://localhost:7009/api/Account/Login', { login, password });
      const role = response.data.role;
      const userId = response.data.userId; // Получаем userId из ответа

      if (role && userId) {
        localStorage.setItem('userRole', role);
        localStorage.setItem('userId', userId); // Сохраняем userId в localStorage
        onLogin(role); // Вызываем onLogin с ролью пользователя
        if (role === 'admin') {
          navigate('/items');
        } else if (role === 'user') {
          navigate('/user-items');
        }
      } else {
        console.error('No role or userId received from the server');
      }
    } catch (error) {
      console.error('Login failed', error);
    }
  };
};

```

Листинг 2.6 страница входа

Для регистрации же форма и структура похожа, с несколькими отличиями, данные введенные пользователем так же отправляются на сервер через auth после чего если процесс регистрации успешен, пользователь перенаправляется обратно на страницу входа, здесь представлены функции формы регистрации:

```
const RegisterForm: React.FC = () => {
  const [loginValue, setLoginValue] = useState('');
  const [password, setPassword] = useState('');
  const [error, setError] = useState('');
  const navigate = useNavigate();

  const handleSubmit = async (event: React.FormEvent) => {
    event.preventDefault();
    try {
      const hashedPassword = await bcrypt.hash(password, 10);
      const response = await register(loginValue, hashedPassword);
      alert(response.message);
      navigate('/login');
    } catch (err) {
      setError('Registration failed');
    }
  };
};
```

Листинг 2.7 форма регистрации

Рассмотрим визуал страницы входа и регистрации у запущенного приложения.

Рисунок 2.2 страница входа

Рисунок 2.3 страница регистрации

Затем пользователь или администратор попадают на страницу со списком всех товаров, пользователь может перейти на более подробную страницу товара или добавить в корзину, а администратор может удалять или редактировать товары, а также добавить новый, рассмотрим то, как работает отображение списка товаров путем запроса на получение данных о них, на всех подобных страницах, алгоритм схож, поэтому для того, чтобы не дублировать похожий код

несколько раз а сосредоточиться на уникальных функциях, рассмотрим получение информации о товарах со страницы каталога у пользователя, полный код данного компонента содержится в приложении Г. Для начала, используем хук `useState` для создания состояний, необходимых для хранения данных о товарах, категориях и уведомлениях. Состояния `items` и `categories` будут хранить данные о товарах и категориях соответственно, а состояние `notification` будет использоваться для управления уведомлениями, которые отображаются пользователю.

Для получения информации о товарах и категориях используются два хука `useEffect`. Первый хук выполняется при монтировании компонента и вызывает асинхронную функцию `fetchItems`, которая отправляет GET-запрос к API `https://localhost:7009/api/Items`. После успешного выполнения запроса данные о товарах сохраняются в состояние `items`. Второй хук также выполняется при монтировании компонента и вызывает функцию `fetchCategories`, которая отправляет GET-запрос к API `https://localhost:7009/api/Categories`. Полученные данные о категориях сохраняются в состояние `categories`.

Функция `fetchItems` отвечает за выполнение асинхронного запроса к серверу для получения списка товаров. Аналогично, функция `fetchCategories` выполняет запрос для получения списка категорий. Оба этих запроса выполняются один раз при загрузке компонента благодаря хукам `useEffect`.

После получения данных о товарах и категориях, компонент `UserItemsListPage` отображает информацию с использованием компонентов из библиотеки `Material-UI`. Товары отображаются в виде сетки (`Grid`), где каждый товар представлен в карточке (`Card`). Карточка товара включает в себя изображение товара (`CardMedia`), название, описание, категорию и цену (`CardContent`), а также кнопку для добавления товара в корзину. Теперь

рассмотрим функцию добавления в корзину, так как она одинакова, что для общей страницы так и конкретного товара. Для добавления товара в корзину используется функция `addToCart`. При нажатии на кнопку "Купить", функция проверяет наличие `userId` в `localStorage`, чтобы убедиться, что пользователь авторизован. Если пользователь не авторизован, отображается уведомление с просьбой войти в систему. Если пользователь авторизован, отправляется POST-запрос к API `https://localhost:7009/api/Cart/{itemId}`, чтобы добавить товар в корзину. В случае успешного добавления товара в корзину, отображается уведомление о том, что товар был успешно добавлен. Функция `handleNotificationClose` используется для закрытия уведомлений. Она обновляет состояние `notification`, чтобы закрыть компонент `Snackbar`, который отображает уведомления пользователю.

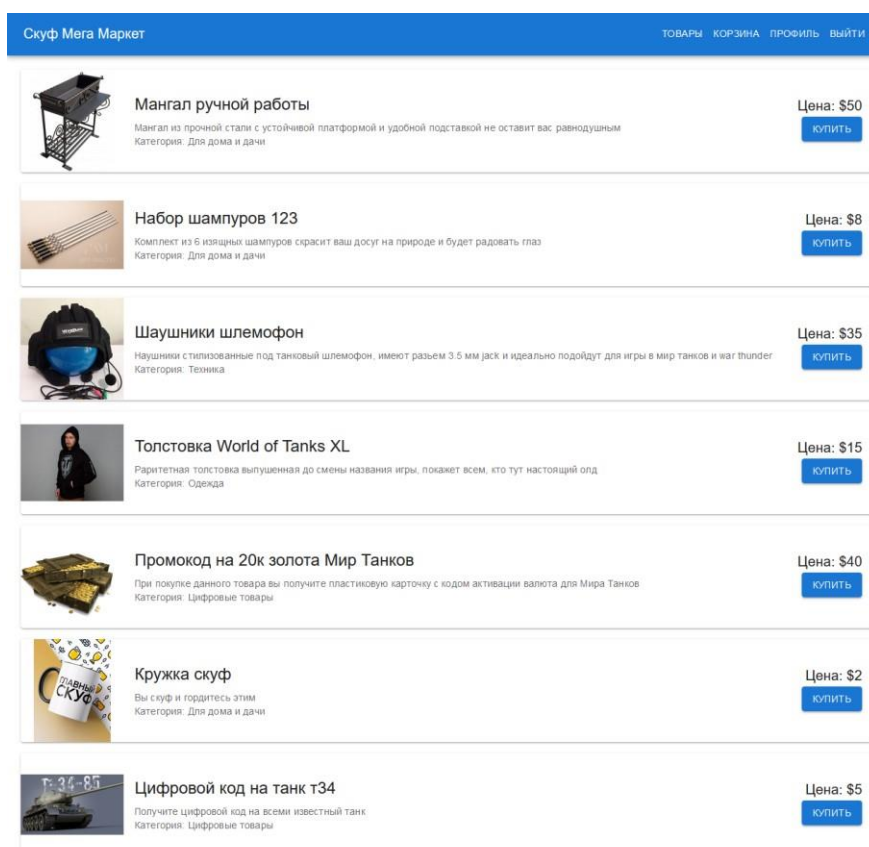


Рисунок 2.4 каталог товаров

Так же с данной страницы можно перейти на детальную страницу конкретного товара, на которой будет указано еще и количество выбранного товара, переход осуществляется по клику на карточку товара с помощью Link: Компонент Link оборачивает информацию о товаре, включая название, описание и категорию. Он создает ссылку на страницу детального просмотра товара, используя путь `/item/${item.id}`.

Путь `/item/${item.id}` включает идентификатор товара (`item.id`). Это позволяет передать уникальный идентификатор товара в URL, что затем используется для получения данных о конкретном товаре на странице детального просмотра.

На странице детального просмотра товара используется компонент `useParams` из `react-router-dom` для получения параметра `id` из URL. Этот идентификатор используется для выполнения запроса к серверу и получения данных о конкретном товаре.

```
const ItemDetailPage: React.FC = () => {
  const { id } = useParams<{ id: string }>();
  const [item, setItem] = useState<Item | null>(null);
  const [categories, setCategories] = useState<Category[]>([]);
  const [notification, setNotification] = useState({ open: false, message: '', severity: 'success' as 'success' | 'error' });

  useEffect(() => {
    const fetchItem = async () => {
      const response = await axios.get(`https://localhost:7009/api/Items/${id}`);
      setItem(response.data);
    };

    const fetchCategories = async () => {
      const response = await axios.get('https://localhost:7009/api/Categories');
      setCategories(response.data);
    };

    fetchItem();
    fetchCategories();
  }, [id]);
```

Листинг 2.8 получение информации о конкретном товаре

Данный фрагмент работает по следующей схеме:

- `useParams` извлекает `id` из параметров URL.
- `useEffect` выполняет запрос к серверу, используя идентификатор товара для получения данных о конкретном товаре.
- Данные о товаре сохраняются в состояние `item`, которое затем используется для отображения информации на странице.

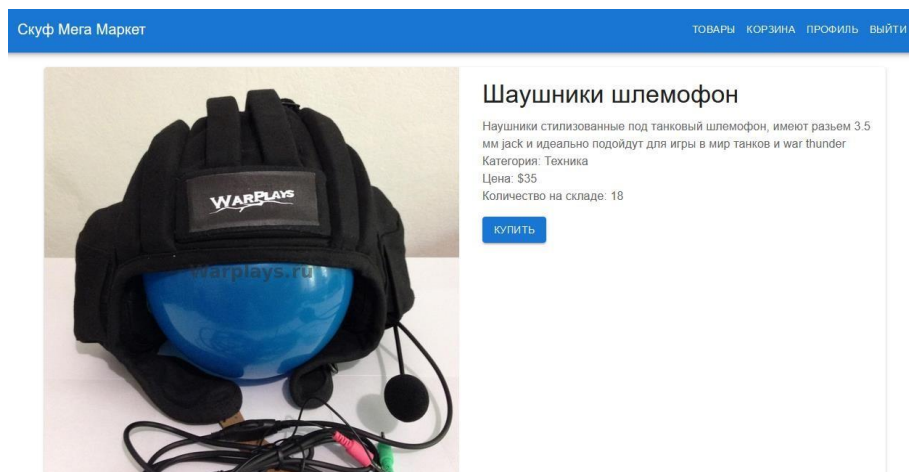


Рисунок 2.5 пример полной страницы товара

Теперь перейдем к корзине, она является из завершающих компонентов пользовательской функциональности интернет магазина. При загрузке компонента с помощью `useEffect` вызывается функция `fetchCart`, которая выполняет запрос к серверу для получения товаров в корзине текущего пользователя по его `userId`.

функция `fetchCart`:

- Запрашивает идентификатор пользователя из `localStorage`.
- Выполняет GET-запрос к серверу по адресу `https://localhost:7009/api/Cart/${userId}`.
- Если запрос успешен, обновляет состояние `cartItems`.

```
const CartPage: React.FC = () => {
  const [cartItems, setCartItems] = useState<any[]>([]);
  const [isDialogOpen, setIsDialogOpen] = useState(false);
  const [dialogTitle, setDialogTitle] = useState('');
  const [dialogDescription, setDialogDescription] = useState('');

  const fetchCart = async () => {
    try {
      const userId = localStorage.getItem('userId');
      if (!userId) {
        throw new Error('User ID not found');
      }

      const response = await axios.get('https://localhost:7009/api/Cart/${userId}');
      setCartItems(response.data);
    } catch (error) {
      console.error('Error fetching cart items:', error);
    }
  };
};
```

Листинг 2.9 получение списка товаров в корзине

Далее идут 2 функции, удаление одного типа товара из корзины или полная очистка товаров, работают они сходим образом отправляя DELETE запрос к

контроллеру управления корзиной либо передавая сначала айди пользователя а затем айди предмета, либо сразу удаляет объекты по айди пользователя.

```
const removeItem = async (itemId: number) => {
  try {
    const userId = localStorage.getItem('userId');
    if (!userId) {
      throw new Error('User ID not found');
    }

    await axios.delete(`https://localhost:7009/api/Cart/${itemId}`, {
      headers: {
        'Authorization': `Bearer ${userId}`
      }
    });
    fetchCart();
  } catch (error) {
    console.error('Error removing item from cart:', error);
  }
};

const clearCart = async () => {
  try {
    const userId = localStorage.getItem('userId');
    if (!userId) {
      throw new Error('User ID not found');
    }

    await axios.delete(`https://localhost:7009/api/Cart/clear/${userId}`);
    fetchCart();
  } catch (error) {
    console.error('Error clearing cart:', error);
  }
};
```

Листинг 2.10 удаление и очистка корзины

Последняя функция корзины — это оформление заказа. `handleOrder` проверяет наличие товаров в корзине и выполняет POST-запрос на сервер для оформления заказа.

- Если товары успешно заказаны, корзина очищается и выводится сообщение об успешном оформлении.
- В случае недостаточного количества товаров на складе или других ошибок выводится соответствующее сообщение. Сами товары же расположены по сетке в виде своих карточек с кнопками для удаления их из корзины.

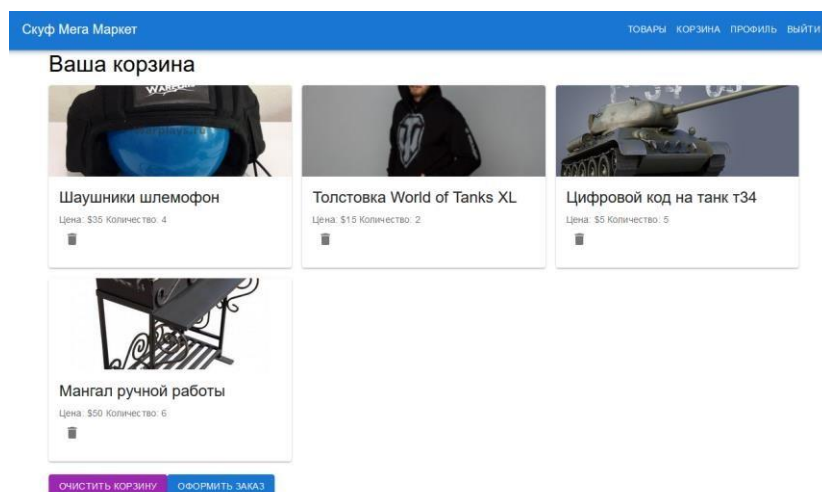


Рисунок 2.6 вид корзины

Последним компонентом является страница профиля, на ней можно обновить данные об аккаунте, при вводе новых данных и нажатии на кнопку обновить происходит PUT запрос к контроллеру аккаунтов, который обновляет данные для юзера, айди которого все так же передается из локального хранилища.

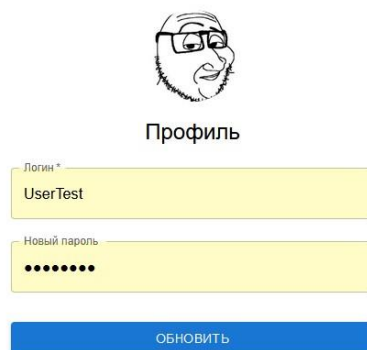


Рисунок 2.7 страница обновления данных профиля

Помимо пользовательской зоны, приложение содержит и панель администратора с возможностью редактирования, и удаления товаров, аккаунт администратора фиксированный, и предопределен в базе данных (всем, кто регистрируется присваивается user), при входе в аккаунт администратора он попадает на страницу, где отображаются все товары.

Список товаров


ДОБАВИТЬ НОВЫЙ ТОВАР

Мангал ручной работы

Категория: Для дома и дачи

Цена: \$50

Количество на складе: 94



РЕДАКТИРОВАТЬ


УДАЛИТЬ

Набор шампуров 123

Категория: Для дома и дачи

Цена: \$8

Количество на складе: 58



РЕДАКТИРОВАТЬ


УДАЛИТЬ

Шаушники шлемофон

Категория: Техника

Цена: \$35

Количество на складе: 18



РЕДАКТИРОВАТЬ


УДАЛИТЬ

Толстовка World of Tanks XL

Категория: Одежда

Цена: \$15

Количество на складе: 143



РЕДАКТИРОВАТЬ


УДАЛИТЬ

Промокод на 20к золота Мир Танков

Категория: Цифровые товары

Цена: \$40

Количество на складе: 20000



РЕДАКТИРОВАТЬ

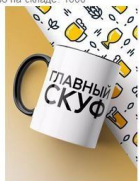
УДАЛИТЬ

Кружка скуф

Категория: Для дома и дачи

Цена: \$2

Количество на складе: 1000



РЕДАКТИРОВАТЬ

УДАЛИТЬ

Рисунок 2.8 каталог администратора

по своим функциям она практически идентична той, что мы разобрали из приложения Г, но есть отличия, в том, что она содержит кнопку, ведущую на добавление товара, редактирование и удаление выбранной позиции.

30

```

const ItemListPage: React.FC = () => {
  const [items, setItems] = useState<Item[]>([]);
  const [categories, setCategories] = useState<Category[]>([]);
  const [modalOpen, setModalOpen] = useState(false);
  const [modalMessage, setModalMessage] = useState('');
  const [confirmDialogOpen, setConfirmDialogOpen] = useState(false);
  const [itemToDelete, setItemToDelete] = useState<number | null>(null);
  const navigate = useNavigate();

  useEffect(() => {
    const fetchItems = async () => {
      try {
        const response = await axios.get('https://localhost:7009/api/Items');
        setItems(response.data);
      } catch (error) {
        console.error('Error fetching items:', error);
      }
    };

    const fetchCategories = async () => {
      try {
        const response = await axios.get('https://localhost:7009/api/Categories');
        setCategories(response.data);
      } catch (error) {
        console.error('Error fetching categories:', error);
      }
    };

    fetchItems();
    fetchCategories();
  }, []);

  const handleDelete = async () => {
    if (itemToDelete !== null) {
      try {
        await axios.delete(`https://localhost:7009/api/Items/${itemToDelete}`);
        setModalMessage('Товар успешно удален');
        setModalOpen(true);
        setItems(items.filter(item => item.id !== itemToDelete));
      } catch (error) {
        console.error('Error deleting item:', error);
        setModalMessage('Ошибка при удалении товара');
        setModalOpen(true);
      } finally {
        setItemToDelete(null);
        setConfirmDialogOpen(false);
      }
    }
  };
};

```

Листинг 2.11 функция удаление

Данный фрагмент содержит знакомые функции по запросу всех товаров списком, но при это есть функция, которой еще не было, это удаление при нажатии на кнопку получаем айди нажатого товара для удаления и открывается окно с подтверждением, после чего отправляется DELETE запрос. Кнопка редактировать так же формирует ссылку на редактирования товара, после чего хук Params вытягивает переданный в айди параметр и запрашивает данные товара чтобы поместить их в форму для редактирования.

```

<CardActions>
  <Button
    variant="outlined"
    color="primary"
    onClick={() => navigate(`/edit-item/${item.id}`)}
    fullWidth
  >
    Редактировать
  </Button>
  <Button
    variant="outlined"
    color="secondary"
    onClick={() => openConfirmDialog(item.id)}
    fullWidth
  >
    Удалить
  </Button>
</CardActions>

```

Листинг 2.12 формирования ссылок на удаление и переход

Страницы редактирования и добавления нового товара похожи и имеют одинаковые формы и поля для ввода, единственное отличие в том, что добавление отправляет POST запрос и данные нужно вводить все, страница редактирования же, как было сказано из параметров ссылки получает айди и заполняет все поля для выбранного предмета, после чего их можно изменять и отправлять PUT запросы на обновление. Разберем работу данных систем на примере добавления нового товара, полный код компонента содержится в приложении Д.

инициализация состояния:

Компонент использует хуки `useState` для управления состояниями таких полей, как название, описание, цена, количество на складе, категория, изображение, предпросмотр изображения, список категорий, а также состояния модального окна и сообщения в нем.

Получение списка категорий:

При загрузке компонента с помощью `useEffect` вызывается функция `fetchCategories`, которая выполняет запрос к серверу для получения списка категорий. Полученные категории сохраняются в состоянии `categories`.

Обработка отправки формы:

Функция `handleSubmit` обрабатывает отправку формы. При отправке формы создается объект `FormData`, в который добавляется изображение. Затем выполняется POST-запрос к серверу для загрузки изображения.

После успешной загрузки изображения создается объект нового товара, включающий название, описание, цену, количество, категорию и путь к изображению. Этот объект отправляется на сервер с помощью POST-запроса.

В случае успешного добавления товара выводится сообщение об успехе, иначе - сообщение об ошибке.

Обработка изменения изображения:

Функция `handleImageChange` обрабатывает изменение изображения. Когда пользователь выбирает файл изображения, он сохраняется в состоянии `image`, а также создается его предварительный просмотр с помощью `FileReader`.

Удаление изображения:

Функция `handleRemoveImage` позволяет удалить выбранное изображение и сбросить состояние предпросмотра.

Закрытие модального окна:

Функция `handleModalClose` закрывает модальное окно и перенаправляет пользователя на страницу со списком товаров.

Отображение интерфейса

Компонент рендерит форму для ввода данных о товаре. Форма включает поля для ввода названия, описания, цены, количества на складе и выбора категории из выпадающего списка. Также предусмотрена возможность загрузки изображения товара. Форма представлена с помощью компонентов `TextField`, `Button`, `Box`, `Typography` и `Container` из библиотеки `@mui/material`. Кнопка для загрузки изображения открывает диалог выбора файла, а предварительный просмотр изображения отображается под кнопкой. При успешном добавлении товара отображается модальное окно с соответствующим сообщением. После закрытия модального окна пользователь перенаправляется на страницу со списком товаров.

Добавить новый товар

Название *

Новый товар1

Описание *

это совершенно новый товар

Цена *

12


Количество на складе *

242

Категория *

Одежда

ЗАГРУЗИТЬ ИЗОБРАЖЕНИЕ



УДАЛИТЬ ИЗОБРАЖЕНИЕ

ДОБАВИТЬ ТОВАР

Рисунок 2.9 добавление товара

Последними же стоит разобрать служебные компоненты, такие как уведомления и хэдеры, как было упомянуто в начале данного раздела приложения содержит 3 различных хедера в зависимости от авторизации пользователя, что было так же заметно на демонстрационных изображениях.

Рассмотрим хэдэр для пользователя.

```
import React from 'react';
import { useNavigate } from 'react-router-dom';
import { AppBar, Toolbar, Typography, Button } from '@mui/material';

const UserHeader: React.FC = () => {
  const navigate = useNavigate();
  const userRole = localStorage.getItem('userRole');

  const handleLogout = () => {
    localStorage.removeItem('userRole');
    localStorage.removeItem('userId');
    navigate('/login');
  };

  return (
    <AppBar position="static">
      <Toolbar>
        <Typography variant="h6" component="div" sx={{ flexGrow: 1 }}>
          Скуп Мерд Маркер
        </Typography>
        <Button color="inherit" onClick={() => navigate('/user-items')}>Товары</Button>
        <Button color="inherit" onClick={() => navigate('/cart')}>Корзина</Button>
        <Button color="inherit" onClick={() => navigate('/profile')}>Профиль</Button>
        <Button color="inherit" onClick={handleLogout}>Выйти</Button>
      </Toolbar>
    </AppBar>
  );
};

export default UserHeader;
```

Листинг 2.13 хэдер пользователя

По своей сути, данный компонент — это панель с кнопками при нажатии на которые, пользователь перенаправляется на ту или иную страницу, а также при

нажатии на кнопку выйти вызывается функции очистки авторизации и перенаправления на страницу входа. Уведомления же представляют собой либо всплывающие окна с сообщением или окна с кнопками вызывающие то или иное действие, например, всплывающее уведомление внизу экрана.

```
interface NotificationProps {
  open: boolean;
  message: string;
  severity: 'success' | 'error' | 'warning' | 'info';
  onClose: () => void;
}

const Notification: React.FC<NotificationProps> = ({ open, message, severity, onClose }) => {
  return (
    <Snackbar open={open} autoHideDuration={6000} onClose={onClose}>
      <Alert onClose={onClose} severity={severity} sx={{ width: '100%' }}>
        {message}
      </Alert>
    </Snackbar>
  );
};

export default Notification;
```

Листинг 2.14 уведомление

Данное уведомление имеет только одну функцию для закрытия его и несет в себе только информативный характер, так же данное уведомление автоматически пропадает если его не закрыть. Но есть и более функциональный тип уведомлений, например, подтверждение удаление товара.

```
interface ConfirmDeleteDialogProps {
  open: boolean;
  title: string;
  description: string;
  onConfirm: () => void;
  onCancel: () => void;
}

const ConfirmDeleteDialog: React.FC<ConfirmDeleteDialogProps> = ({ open, title, description, onConfirm, onCancel }) => {
  return (
    <Dialog open={open} onClose={onCancel}>
      <DialogTitle>{title}</DialogTitle>
      <DialogContent>
        <Typography>{description}</Typography>
      </DialogContent>
      <DialogActions>
        <Button onClick={onCancel} color="primary">
          Отмена
        </Button>
        <Button onClick={onConfirm} color="secondary">
          Удалить
        </Button>
      </DialogActions>
    </Dialog>
  );
};
```

Листинг 2.15 окно подтверждения удаления

Функция удаления товара использует вызов onConFirm, как начало работы, таким образом мы рассмотрели все возможности клиентской части приложения из чего так же можно сделать вывод о том, что поставленные цели и задачи были достигнуты.

Заключение

Процесс разработки интернет-магазина с использованием базы данных стал важным этапом в освоении современных технологий веб-разработки. Этот проект охватил все ключевые аспекты создания веб-приложений, начиная с проектирования и реализации серверной части, заканчивая созданием функционального и удобного пользовательского интерфейса.

В ходе работы над проектом была проведена тщательная проработка структуры базы данных. Основное внимание уделялось проектированию таблиц и их взаимосвязям, что обеспечило целостность и согласованность данных. Были разработаны ключевые сущности, такие как пользователи, категории товаров, товары, корзины и элементы корзин.

Реализация серверной части на основе ASP.NET Core обеспечила создание надежного API для управления пользователями, товарами и корзинами покупок.

Фронтенд-часть проекта была разработана с использованием React и библиотеки компонентов Material-UI. Это позволило создать адаптивный пользовательский интерфейс, обеспечивающий удобное взаимодействие пользователей с системой. Управление состоянием приложения и маршрутизация стали ключевыми элементами разработки, обеспечивающими корректное функционирование интерфейса.

В результате проделанной работы был реализован интернет-магазин с необходимой функциональностью, обеспечивающей полноценное функционирование системы.

Список Литературы

- 1) Трофимов В.В. ASP.NET Core: разработка современных вебприложений. - М.: ДМК Пресс, 2020.
- 2) Зубков С.И. Введение в Entity Framework Core. - СПб.: БХВ-Петербург, 2021.
- 3) Соколов А.В. Архитектура веб-приложений на ASP.NET Core и React. - М.: ДМК Пресс, 2020.
- 4) Курс базы данных 3-4 семестр [Электронный ресурс]. – Московский Политехнический университет.
- 5) Курс Back-end разработка [Электронный ресурс]. – Московский Политехнический университет.
- 6) Курс Веб программирование и дизайн [Электронный ресурс]. – Московский Политехнический университет.

Приложение А

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic; using
System.Threading.Tasks;
using coursework.Models;

namespace coursework.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class ItemsController : ControllerBase
    {
        private readonly ShopContext _context;

        public ItemsController(ShopContext context)
        {
            _context = context;
        }

        // GET: api/Items
        [HttpGet]
        public async Task<ActionResult<IEnumerable<Item>>> GetItems()
        {
            return await _context.Items.ToListAsync();
        }

        // GET: api/Items/{id}
        [HttpGet("{id}")]
        public async Task<ActionResult<Item>> GetItem(int id)
        {
            var item = await _context.Items.FindAsync(id);

            if (item == null)
            {
                return NotFound();
            }

            return item;
        }

        // POST: api/Items
        [HttpPost]
        public async Task<ActionResult<Item>> AddItem(Item newItem)
        {
            if (!ModelState.IsValid)
                return BadRequest(ModelState);

            _context.Items.Add(newItem);
            await _context.SaveChangesAsync();

            return CreatedAtAction(nameof(GetItem), new { id = newItem.Id }, newItem);
        }
    }
}
```

```
// PUT: api/Items/{id}
[HttpPut("{id}")]
public async Task<IActionResult> UpdateItem(int id, Item updatedItem)
{
    if (id !=
updatedItem.Id)
    {
        return BadRequest();
    }
}
```

```

_context.Entry(updatedItem).State = EntityState.Modified;
try
{
    await _context.SaveChangesAsync();
}
catch (DbUpdateConcurrencyException)
{
    if (!ItemExists(id))
    {
        return NotFound();
    }
    else
    {
        throw;
    }
}

return NoContent();
}

// DELETE: api/Items/{id} [HttpDelete("{id}")]
public async Task<IActionResult> DeleteItem(int id)
{
    var item = await _context.Items.FindAsync(id);    if (item == null)
    {
        return NotFound();
    }

    _context.Items.Remove(item);    await _context.SaveChangesAsync();

    return NoContent();
}

private bool ItemExists(int id)
{
    return _context.Items.Any(e => e.Id == id);
}
}

```


Приложение Б

```
using Microsoft.AspNetCore.Mvc;
using
Microsoft.EntityFrameworkCore;
using System.Threading.Tasks; using
coursework.Models;
using Microsoft.Extensions.Logging;

namespace coursework.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class AccountController : ControllerBase
    {
        private readonly ShopContext _context;
        private readonly ILogger<AccountController> _logger;

        public AccountController(ShopContext context, ILogger<AccountController> logger)
        {
            _context = context;
            _logger = logger;
        }

        // POST: api/Account/Register
        [HttpPost("Register")] public async Task<IActionResult>
        Register(RegisterModel model)
        {
            if (!ModelState.IsValid)
                return BadRequest(ModelState);

            if (await _context.Users.AnyAsync(u => u.Login == model.Login))
                return Conflict("User already exists");

            var user = new User
            {
                Login = model.Login,
                Password = model.Password,
                Role = "user"
            };

            _context.Users.Add(user);
            await _context.SaveChangesAsync();

            return Ok(new { message = "User registered successfully", userId = user.Id });
        }

        // POST: api/Account/Login
        [HttpPost("Login")]
        public async Task<IActionResult> Login(LoginModel model)
        {
            if (!ModelState.IsValid)
                return BadRequest(ModelState);

            var user = await _context.Users
                .FirstOrDefaultAsync(u => u.Login == model.Login && u.Password == model.Password);

            if (user == null)
```

```
    {  
        _logger.LogWarning("Invalid login attempt for user: {Login}", model.Login);  
        return Unauthorized("Invalid login or password");  
    }  
  
    _logger.LogInformation("User {Login} logged in successfully with userId {UserId}", user.Login, user.Id);  
    return Ok(new { role = user.Role, userId = user.Id });  
}
```

```

// POST: api/Account/Logout
[HttpPost("Logout")]
public IActionResult Logout()
{
    // успешный ответ
    _logger.LogInformation("User logged out successfully");
return Ok("Logged out successfully");
}

// PUT: api/Account/{id}
[HttpPut("{id}")]
public async Task<IActionResult> UpdateProfile(int id, UpdateProfileModel model)
{
    if (!ModelState.IsValid)
        return BadRequest(ModelState);

    var user = await _context.Users.FindAsync(id);
if (user == null)
    return NotFound("User not found");

    user.Login = model.Login;
user.Password = model.Password;

    _context.Users.Update(user);
    await _context.SaveChangesAsync();

    return Ok(new { message = "Profile updated successfully" });
}
}
}

```

Приложение В

```
using Microsoft.AspNetCore.Mvc;
using
Microsoft.EntityFrameworkCore;
using System.Linq; using
System.Threading.Tasks; using
coursework.Models;
using System.Collections.Generic;

namespace coursework.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class CartController : ControllerBase
    {
        private readonly ShopContext _context;

        public CartController(ShopContext context)
        {
            _context = context;
        }

        [HttpGet("{userId}")]
        public async Task<ActionResult<IEnumerable<object>>> GetCartItems(int userId)
        {
            var cart = await _context.Carts.FirstOrDefaultAsync(c => c.UserId == userId);

            if (cart == null)
            {
                return Ok(new List<object>());
            }

            var cartItems = await _context.CartItems
                .Where(ci => ci.CartId == cart.Id)
                .ToListAsync();

            var result = new List<object>();

            foreach (var ci in cartItems)
            {
                var item = await _context.Items.FindAsync(ci.ItemId);
                result.Add(new
                {
                    ci.Id,
                    ci.CartId,
                    ci.ItemId,
                    ci.Quantity,
                    item
                });
            }

            return Ok(result);
        }

        [HttpPost("{itemId}")]
        public async Task<ActionResult> AddToCart(int itemId, [FromHeader] string Authorization)
        {

```

```
        if (string.IsNullOrEmpty(Authorization))  
return Unauthorized();  
  
        var userId = int.Parse(Authorization.Replace("Bearer ", ""));  
        var cart = await _context.Carts.FirstOrDefaultAsync(c => c.UserId == userId);  
        if (cart ==  
null)  
        {  
            cart = new Cart { UserId = userId };  
        }
```

```

        _context.Carts.Add(cart);
        await _context.SaveChangesAsync();
    }

    var cartItem = await _context.CartItems.FirstOrDefaultAsync(ci => ci.CartId == cart.Id && ci.ItemId == itemId);

    if (cartItem != null)
    {
        cartItem.Quantity++;
    }
else
    {
        cartItem = new CartItem { CartId = cart.Id, ItemId = itemId, Quantity = 1 };
        _context.CartItems.Add(cartItem);
    }

    await _context.SaveChangesAsync();
    return Ok("Товар добавлен в корзину");
}

[HttpDelete("{itemId}")]
public async Task<ActionResult> RemoveFromCart(int itemId, [FromHeader] string Authorization)
{
    if (string.IsNullOrEmpty(Authorization))
        return Unauthorized();

    var userId = int.Parse(Authorization.Replace("Bearer ", ""));
    var cart = await _context.Carts.FirstOrDefaultAsync(c => c.UserId == userId);

    if (cart == null)
        return NotFound("Корзина не найдена");

    var cartItem = await _context.CartItems.FirstOrDefaultAsync(ci => ci.CartId == cart.Id && ci.ItemId == itemId);

    if (cartItem == null)
        return NotFound("Товар не найден в корзине");

    _context.CartItems.Remove(cartItem);
    await _context.SaveChangesAsync();
    return Ok("Товар удален из корзины");
}

[HttpDelete("clear/{userId}")]
public async Task<ActionResult> ClearCart(int userId)
{
    var cart = await _context.Carts.FirstOrDefaultAsync(c => c.UserId == userId);

    if (cart == null)
        return NotFound("Корзина не найдена");

    var cartItems = _context.CartItems.Where(ci => ci.CartId == cart.Id);
    _context.CartItems.RemoveRange(cartItems);
    await _context.SaveChangesAsync();

    return Ok("Корзина очищена");
}

```

```
[HttpPost("checkout")]
public async Task<IActionResult> Checkout([FromHeader] string Authorization)
{
    if (string.IsNullOrEmpty(Authorization))
        return Unauthorized();

    var userId = int.Parse(Authorization.Replace("Bearer ", ""));
    var cart = await _context.Carts.FirstOrDefaultAsync(c => c.UserId == userId);

    if (cart == null)
```

```

    {
        return Ok(new { success = false, message = "Корзина не найдена" });
    }

    var cartItems = await _context.CartItems
        .Where(ci => ci.CartId == cart.Id)
        .ToListAsync();

    foreach (var cartItem in cartItems)
    {
        var item = await _context.Items.FindAsync(cartItem.ItemId);
        if (item == null || item.Stock < cartItem.Quantity)
        {
            return Ok(new { success = false, message = "Недостаточно товара на складе" });
        }
    }

    foreach (var cartItem in cartItems)
    {
        var item = await _context.Items.FindAsync(cartItem.ItemId);
        if (item != null)
        {
            item.Stock -= cartItem.Quantity;
        }
    }

    _context.CartItems.RemoveRange(cartItems);
    await _context.SaveChangesAsync();

    return Ok(new { success = true, message = "Заказ успешно оформлен" });
}
}

```


Приложение Г

```
import React, { useState, useEffect } from 'react'; import { Link } from 'react-router-dom'; import
axios from 'axios'; import { Card, CardContent, CardMedia, Typography, Button, Grid, Snackbar }
from '@mui/material'; import Alert from '@mui/material/Alert';

interface Item {  id:
number;  name:
string;  description:
string;  price:
number;  stock:
number;
categoryId: number;
imagePath: string;
}

interface Category {
id: number;
name: string;
}

const UserItemsListPage: React.FC = () => {  const [items, setItems] = useState<Item[]>([]);  const
[categories, setCategories] = useState<Category[]>([]);  const [notification, setNotification] = useState({ open: false,
message: "", severity: 'success' as 'success' | 'error' | 'warning' | 'info' });

  useEffect(() => {    const fetchItems = async () => {      const
response = await axios.get('https://localhost:7009/api/Items');
setItems(response.data);
    };

    const fetchCategories = async () => {      const response = await
axios.get('https://localhost:7009/api/Categories');
setCategories(response.data);
    };

    fetchItems();
    fetchCategories();
  }, []);

  const addToCart = async (itemId: number) => {
```



```

    try {
      const userId =
localStorage.getItem('userId');
      if (!userId) {
        setNotification({ open: true, message: 'Пожалуйста, войдите в систему, чтобы добавить товар в корзину.', severity: 'warning' });
        return;
      }
      await axios.post(`https://localhost:7009/api/Cart/${itemId}`, {}, {
headers: {
      'Authorization': `Bearer ${userId}`
    }
  });
      setNotification({ open: true, message: 'Товар добавлен в корзину', severity: 'success' });
    } catch (error) {
      console.error('Error adding item to cart:', error);
      if ((error as any).response && (error as any).response.status === 401) {
        setNotification({ open: true, message: 'Ваша сессия истекла. Пожалуйста, войдите в систему снова.', severity: 'error' });
      } else {
        setNotification({ open: true, message: 'Произошла ошибка при добавлении товара в корзину.', severity: 'error' });
      }
    }
  };

  const handleNotificationClose = () => {
setNotification({ ...notification, open: false });
  };

  return (
    <Grid container spacing={2} style={{ padding: '20px' }}>
      {items.map(item => (
        <Grid item xs={12} key={item.id}>
          <Card style={{ display: 'flex', flexDirection: 'row', alignItems: 'center' }}>
            <CardMedia
              component="img"
              style={{ width: 150, height: 150, objectFit: 'contain' }}
              image={`https://localhost:7009/${item.imagePath}`}
              alt={item.name}
            />
            <CardContent style={{ flex: 1 }}>
              <Link to={`/item/${item.id}`} style={{ textDecoration: 'none', color: 'inherit' }}>
                <Typography gutterBottom variant="h5" component="div">
                  {item.name}
                </Typography>
              </Link>
            </CardContent>
          </Card>
        </Grid>
      ))}
    </Grid>
  );

```

```

        <Typography variant="body2" color="text.secondary">
          {item.description}
        </Typography>
        <Typography variant="body2" color="text.secondary">
          Категория: {categories.find(category => category.id === item.categoryId)?.name}
        </Typography>
        </Link>
      </CardContent>
    <CardContent style={{ display: 'flex', flexDirection: 'column', alignItems: 'flex-end' }}>
      <Typography variant="h6" component="div">
        Цена: ${item.price}
      </Typography>
      <Button variant="contained" color="primary" style={{ marginTop: 'auto' }} onClick={() =>
addToCart(item.id)}>
        Купить
      </Button>
    </CardContent>
  </Card>
</Grid>
))}
<Snackbar open={notification.open} autoHideDuration={6000} onClose={handleNotificationClose}>
  <Alert onClose={handleNotificationClose} severity={notification.severity} sx={{ width: '100%' }}>
    {notification.message}
  </Alert>
</Snackbar>
</Grid>
);
};

export default UserItemsListPage;

```

Приложение Д

```
// AddItemPage.tsx import React, { useState, useEffect } from 'react'; import { TextField,
Button, Box, Typography, Container, MenuItem } from '@mui/material'; import axios from
'axios'; import { useNavigate } from 'react-router-dom'; import ModalNotification from
'../components/ModalNotification';

const AddItemPage: React.FC = () => {  const [name, setName] = useState("");
const [description, setDescription] = useState("");  const [price, setPrice] =
useState("");  const [stock, setStock] = useState("");  const [categoryId,
setCategoryId] = useState("");  const [image, setImage] = useState<File |
null>(null);  const [preview, setPreview] = useState<string | null>(null);  const
[categories, setCategories] = useState<{ id: number, name: string }[]>([]);
const navigate = useNavigate();  const [modalOpen, setModalOpen] =
useState(false);  const [modalMessage, setModalMessage] = useState("");

  useEffect(() => {      const
fetchCategories = async () => {      try
{
        const response = await axios.get('https://localhost:7009/api/Categories');
setCategories(response.data);
      } catch (error) {      console.error('Error
fetching categories:', error);
      }
    };
    fetchCategories();
  }, []);

  const handleSubmit = async (event: React.FormEvent) =>
{    event.preventDefault();    if (!image) return;

    const formData = new FormData();
formData.append('file', image);

    try {
      const imageResponse = await axios.post('https://localhost:7009/api/ImageUpload', formData, {
headers: {
        'Content-Type': 'multipart/form-data',
```

```

    },
  });

  const newItem = {
    name, description,
    price: parseFloat(price),
    stock: parseInt(stock, 10),
    categoryId: parseInt(categoryId, 10),
    imagePath: imageResponse.data.filePath,
  };

  await axios.post('https://localhost:7009/api/Items', newItem);
  setModalMessage('Товар успешно добавлен');
  setModalOpen(true); } catch (error) {
  console.error('Error adding item:', error);
  setModalMessage('Ошибка при добавлении товара');
  setModalOpen(true);
}
};

const handleImageChange = (e: React.ChangeEvent<HTMLInputElement>) => {
  if (e.target.files && e.target.files[0]) {
    const file = e.target.files[0];
    setImage(file); const reader = new
    FileReader(); reader.onloadend = ()
    => { setPreview(reader.result as
    string);
  };
};

```

```

        reader.readAsDataURL(file);
    }
};

const handleRemoveImage = () => {
    setImage(null);    setPreview(null);
};

const handleModalClose = () => {
    setModalOpen(false);
    navigate('/items'); // Redirect to items list page
};

return (
    <Container component="main" maxWidth="xs">
        <Box
            sx={{
marginTop: 8,                display:
'flex',                flexDirection:
'column',                alignItems:
'center',
            }}
        >
            <Typography component="h1" variant="h5">
                Добавить новый товар
            </Typography>
            <Box component="form" onSubmit={handleSubmit} sx={{ mt: 1 }}>

```

```

        <TextField                margin="normal"
required                fullWidth
label="Название"        name="name"
value={ name }          onChange={ (e) =>
setName(e.target.value)}          autoFocus
/>

        <TextField                margin="normal"
required                fullWidth
label="Описание"        name="description"
value={ description }    onChange={ (e) =>
setDescription(e.target.value)}
/>

        <TextField
margin="normal"
required                fullWidth
label="Цена"
name="price"
type="number"
value={ price }
        onChange={ (e) => setPrice(e.target.value)}
/>

        <TextField
margin="normal"
        required
        fullWidth
        label="Количество на складе"
        name="stock"                type="number"
value={ stock }                onChange={ (e) =>
setStock(e.target.value)}

```



```

        />
        <TextField
            margin="normal"
            required
            fullWidth
            label="Категория"
            name="category"
            select
            value={categoryId}
            onChange={(e) => setCategoryId(e.target.value)}
        >
            {categories.map((category) => (
                <MenuItem key={category.id} value={category.id}>
                    {category.name}
                </MenuItem>
            ))}
        </TextField>

        <Button
            variant="contained"
            component="label"
            fullWidth
            sx={{ mt:
            3, mb: 2 }}
        >
            Загрузить изображение
            <input
            type="file"
            hidden
            onChange={handleImageChange}
        />
        </Button>
        {preview && (
            <Box sx={{ mt: 2, mb: 2 }}>
                <img src={preview} alt="Превью" style={{ width: '100%' }} />
            </Box>
        )}
    </div>

```

```
        <Button variant="contained" color="secondary" fullWidth onClick={handleRemoveImage}>
            Удалить изображение
        </Button>
    </Box>
)}
<Button type="submit" fullWidth variant="contained" sx={{ mt: 3, mb: 2 }}>
```

```
        Добавить товар
        </Button>
    </Box>
    <ModalNotification
    open={modalOpen}
    message={modalMessage}
    onClose={handleModalClose}
    />
    </Box>
    </Container>
    );
};

export default AddItemPage;
```