**Поволжский Государственный Университет Телекоммуникаций и Информатики**Кафедра «Программная инженерия»

|  |  |
| --- | --- |
| Сдана на проверку  «\_\_» \_\_\_\_2024 г. | Допустить к защите  «\_\_» \_\_\_\_\_ 2024 г.  Защищена с оценкой  «\_\_\_» \_\_\_\_\_\_\_\_ 2024 г. |

**КУРСОВАЯ РАБОТА**По дисциплине: «Прикладное программирование»  
На тему: «Микросервисная архитектура для e-commerce платформы»  
  
Пояснительная записка

|  |  |
| --- | --- |
| Студент группы ИВТ-22\_\_\_\_\_\_\_ Романов М.А  (роспись) (ФИО)  Руководитель \_\_\_\_\_\_\_\_ к.т.н., доц. Ахметшина Э. Г.  (роспись) (ФИО) | 0 220085 0  (№ зачетной книжки) |

Самара 2024

Оглавление

[Оглавление 1](#_Toc187782484)

[Рецензия 2](#_Toc187782485)

[Введение 3](#_Toc187782486)

[Цель и задачи работы 4](#_Toc187782487)

[Описание методов и технологий 5](#_Toc187782488)

[Реализация 7](#_Toc187782489)

[Дизайн 7](#_Toc187782490)

[База данных 8](#_Toc187782491)

[Программный код 9](#_Toc187782492)

[Аутентификация: 10](#_Toc187782493)

[Каталог товаров 23](#_Toc187782494)

[Корзина и оформление заказа 28](#_Toc187782495)

[Админ панель 35](#_Toc187782496)

[Docker и файлы работы 70](#_Toc187782497)

[ПРИЛОЖЕНИЕ – А 75](#_Toc187782498)

[Список используемых источников 82](#_Toc187782499)

Рецензия

Введение

Современные технологии и стремительное развитие электронной коммерции (e-commerce) предъявляют высокие требования к архитектуре веб-приложений. Для обеспечения гибкости, масштабируемости и надёжности, разработчики всё чаще обращаются к микросервисной архитектуре, которая предоставляет уникальные возможности для решения задач электронной коммерции.

Микросервисная архитектура представляет собой подход к разработке программного обеспечения, при котором приложение разделяется на множество мелких сервисов, каждый из которых выполняет свою конкретную задачу и взаимодействует с другими сервисами через стандартизированные интерфейсы. Такой подход позволяет упростить управление системой, повысить её отказоустойчивость и сократить время на внедрение новых функций.

Применение микросервисной архитектуры в контексте e-commerce платформы открывает широкий спектр возможностей. Каждый микросервис может быть ответственен за определённую область функционала, будь то управление товарами, обработка заказов, учёт платежей или работа с клиентскими данными. Это позволяет улучшить производительность системы, обеспечивая независимость и параллельность выполнения операций.

Целью данной работы является исследование особенностей применения микросервисной архитектуры в e-commerce платформах, анализ её преимуществ и недостатков, а также разработка рекомендаций по внедрению данной архитектуры для повышения эффективности работы системы. В процессе работы будут рассмотрены основные принципы проектирования микросервисов, используемые технологии и инструменты, а также примеры успешных реализаций в реальных проектах.

Результаты исследования могут быть полезны разработчикам, архитекторам и компаниям, планирующим переход на микросервисную архитектуру или её использование для создания высоконагруженных и надёжных e-commerce систем.

Цель и задачи работы

Целью данной работы является обоснование преимуществ микросервисной архитектуры для платформ электронной коммерции (e-commerce). В рамках исследования будут рассмотрены ключевые аспекты, влияющие на эффективность e-commerce систем, такие как масштабируемость, надёжность, производительность и гибкость. Особое внимание будет уделено анализу влияния микросервисного подхода на возможности адаптации платформ к изменениям требований рынка и запросам пользователей.

Для достижения поставленной цели необходимо:

1. Изучить основные принципы микросервисной архитектуры и их применение в контексте e-commerce.
2. Выявить преимущества микросервисного подхода по сравнению с монолитной архитектурой для решения задач электронной коммерции.
3. Рассмотреть реальные примеры использования микросервисов в известных e-commerce платформах.
4. Сверстать сайт интернет магазина, для внедрения в него микросервисной архитектуры.
5. Разработать рекомендации по внедрению микросервисной архитектуры для повышения эффективности и конкурентоспособности e-commerce систем.

Описание методов и технологий

Для реализации платформы электронной коммерции на основе микросервисной архитектуры используются современные технологии и подходы, обеспечивающие высокую производительность, гибкость и удобство разработки. В рамках данной работы применяются следующие инструменты и технологии:

React: Популярная библиотека для разработки пользовательских интерфейсов. React обеспечивает модульный подход к созданию компонентов, что идеально сочетается с принципами микросервисной архитектуры.

Next.js: Фреймворк на базе React, предлагающий серверный рендеринг (SSR) и статическую генерацию (SSG). Эти возможности позволяют улучшить производительность и SEO, что критично для e-commerce платформ.

Tailwind CSS: Утилитарный CSS-фреймворк, обеспечивающий гибкость и высокую скорость стилизации. Tailwind позволяет легко создавать адаптивные интерфейсы, используя классы напрямую в разметке.

ShadCN: Современный инструмент для управления темами и стилями, позволяющий настраивать пользовательский интерфейс в соответствии с уникальными требованиями бренда. Он интегрируется с Tailwind для улучшения визуального опыта пользователей.

JWT-аутентификация: Технология JSON Web Token используется для авторизации пользователей. Этот подход обеспечивает безопасность и масштабируемость, позволяя эффективно управлять доступом к различным частям системы.

MySQL: Реляционная база данных, выбранная для хранения и управления данными. MySQL обеспечивает высокую производительность, отказоустойчивость и гибкость при работе с большими объемами данных.

REST API: Подход к взаимодействию между микросервисами и фронтендом. REST API позволяет стандартизировать обмен данными, упрощая интеграцию и масштабирование системы.

Docker: Для удобного разворачивания сайта на любом десктопном устройстве

Вышеуказанные технологии обеспечивают эффективное управление стилями и упрощают разработку адаптивных, интерактивных и высокопроизводительных интерфейсов. Их совместное использование позволяет создавать платформы, отвечающие высоким требованиям рынка электронной коммерции.

Реализация

Дизайн

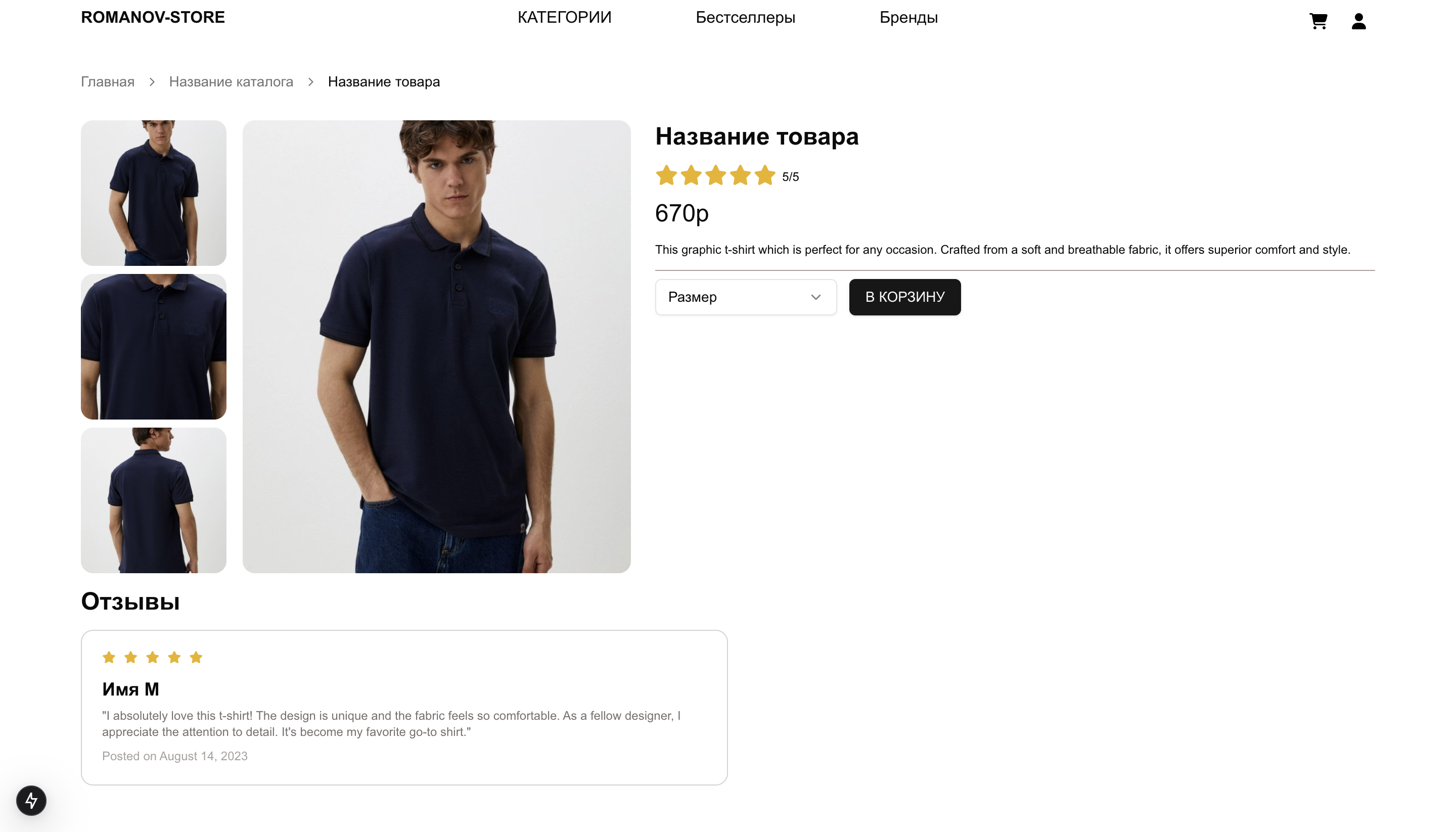


Рис Страница товара

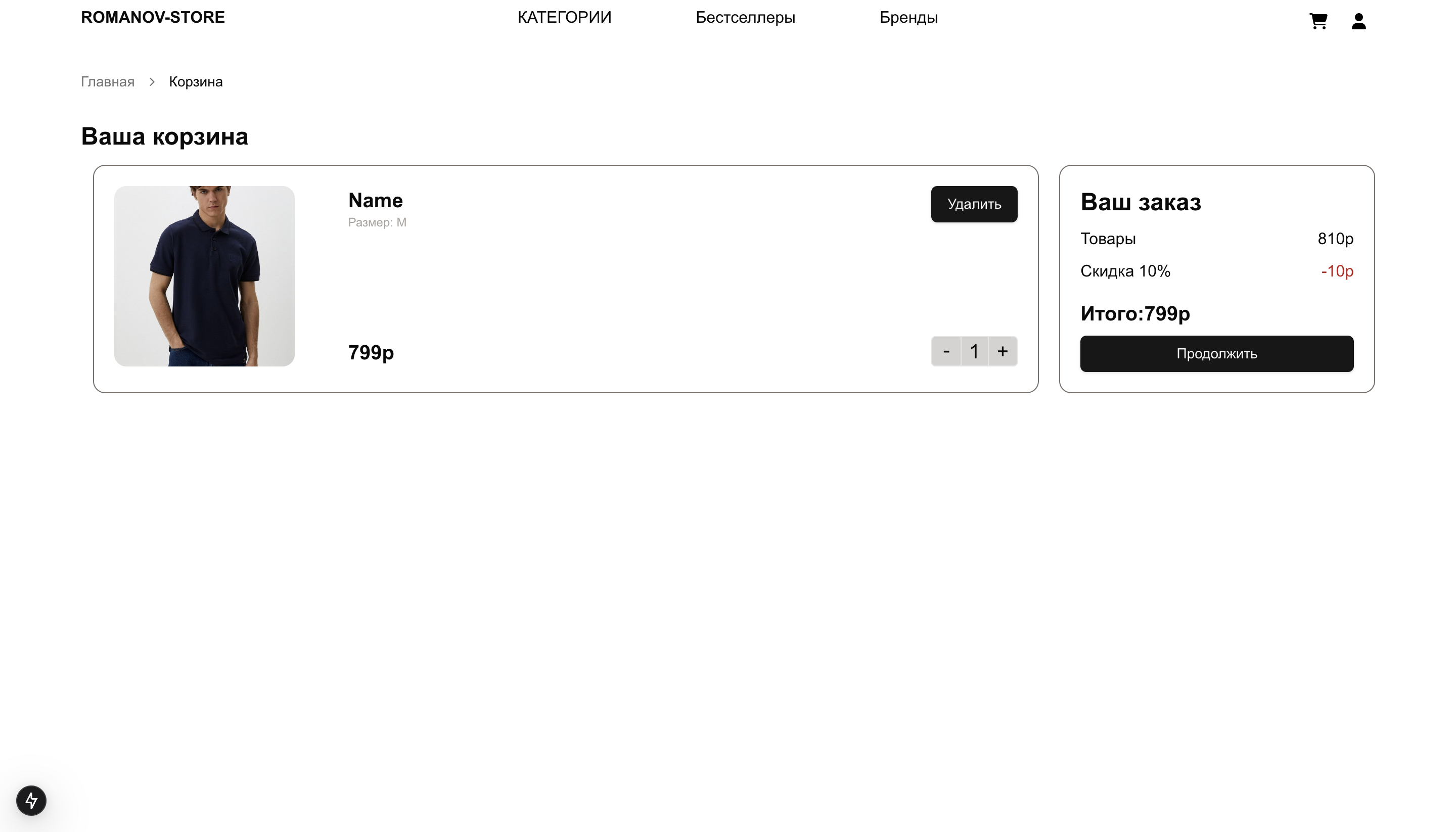


Рис Корзина

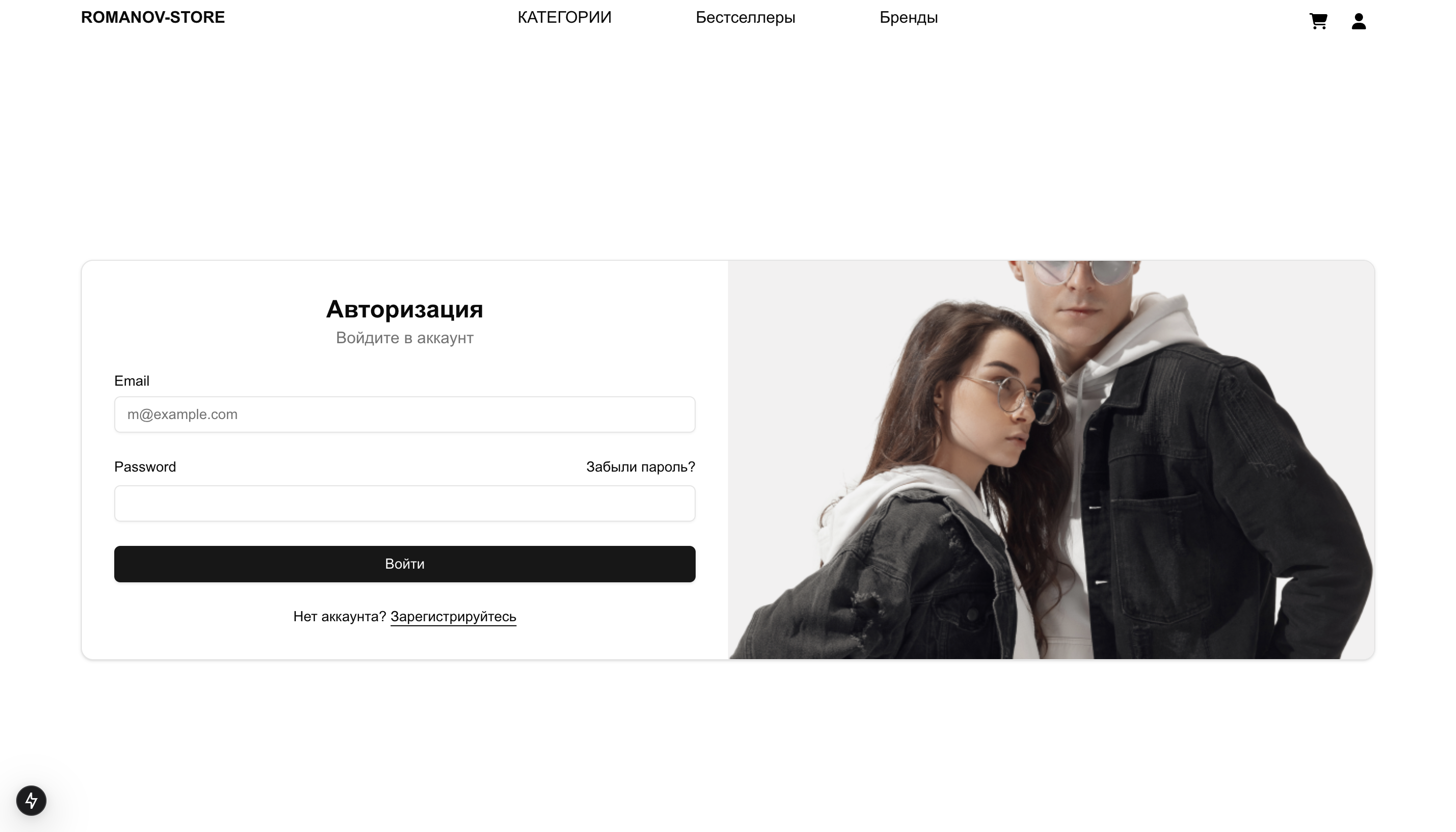


Рис Авторизация Рис Регистрация

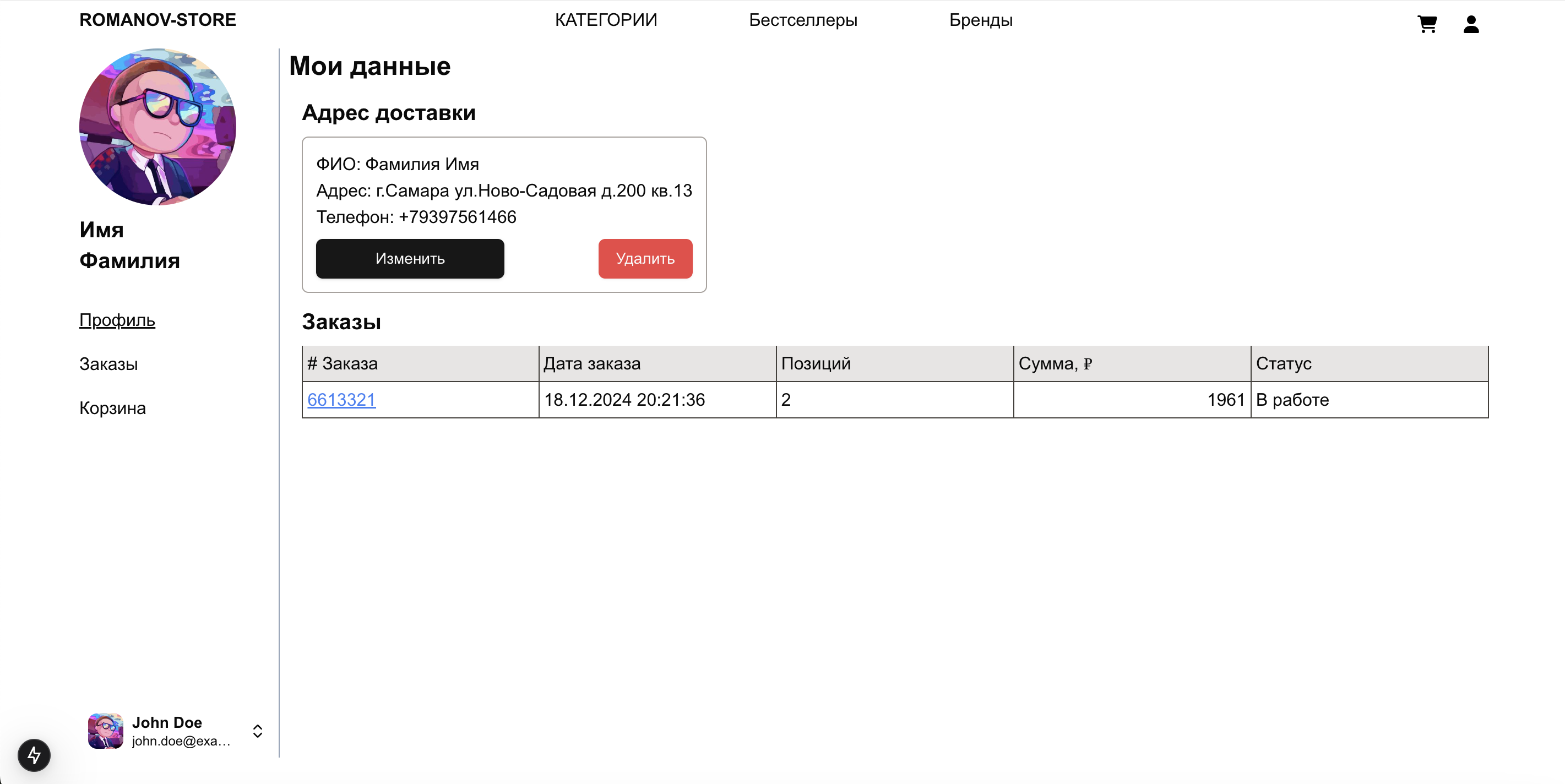


Рис Личный кабинет

База данных

В качестве базы данных, в работе будет использована mySQL. Потому что я имею достаточный опыт работы с этой БД.

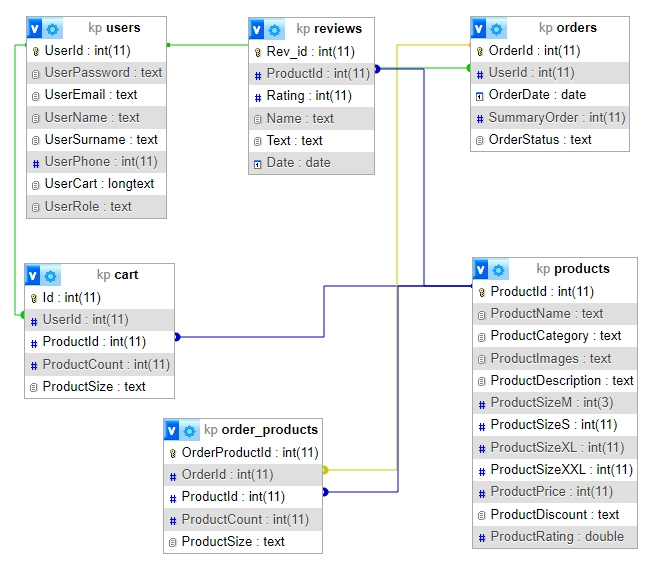


Рис. 20 Структура базы данных

Программный код

Подключение базы данных:

Server.js:

import mysql from 'mysql2/promise';

const { DB\_HOST, DB\_USER, DB\_PASSWORD, DB\_NAME } = process.env;

const pool = mysql.createPool({

  host: DB\_HOST,

  user: DB\_USER,

  password: DB\_PASSWORD,

  database: DB\_NAME,

  connectionLimit: 6,

});

export default pool;

import pool from "../../server"; - Файл подключения базы данных

import jwt from 'jsonwebtoken'; - Библиотека для работы с jwt

const SECRET\_KEY = process.env.JWT\_SECRET\_KEY; - Закрытый ключ

Аутентификация:

REST API **авторизации**:

import bcrypt from 'bcrypt';

export async function POST(req) {

    try {

        const { email, password } = await req.json();

        // Проверка пользователя

        const [rows] = await pool.query('SELECT \* FROM users WHERE UserEmail = ?', [email]);

        if (rows.length === 0) {

            return new Response(

                JSON.stringify({ message: 'Не найден' }),

                { status: 404 }

            );

        }

        const user = rows[0];

        // Проверка пароля

        const isMatch = await bcrypt.compare(password, user.UserPassword)

        if (!isMatch) {

            return new Response(

                JSON.stringify({ message: 'Неверный пароль' }),

                { status: 401 }

            );

        }

        // Генерация JWT

        const token = jwt.sign(

            {

                id: user.UserId,

                name: user.UserName,

                email: user.UserEmail,

                role: user.UserRole,

            },

            SECRET\_KEY,

            { expiresIn: '7d' }

        );

        return new Response(

            JSON.stringify({ token }),

            { status: 200 }

        );

    } catch (err) {

        return new Response(

            JSON.stringify({ message: 'Internal server error' }),

            { status: 500 }

        );

    }

}

Клиентская сторона авторизации:

function setCookie(name, value, days) {

const expires = new Date();

expires.setTime(expires.getTime() + (days \* 24 \* 60 \* 60 \* 1000));

const expiresStr = "expires=" + expires.toUTCString();

document.cookie = `${name}=${value}; ${expiresStr}; path=/`;

}

const [email, setEmail] = useState('')

const [password, setPassword] = useState('')

async function login(e) {

e.preventDefault();

try {

const response = await fetch('/api/login', {

method: 'POST',

headers: {

'Content-Type': 'application/json',

},

body: JSON.stringify({ email, password }),

});

const res = await response.json();

if (response.ok) {

alert('Успешная авторизация');

setCookie('token', res.token, 7);

router.push('/dashboard')

} else {

alert(res.message);

}

} catch (error) {

alert('Ошибка: Сервер недоступен.');

}

}

Функция setCookie записывает в куки токен пользователя, токен обновляется при каждом входе пользователя на страницу через дополнительную функцию. В переменных email и password хранятся логин и пароль пользователя, e.PreventDefault() отключает стандартное поведение формы. Т.е отключает перезагрузку страницы после отправки формы. Далее идет попытка POST запроса на сервер через метод fetch. На сервере происходит проверка введенных данных. const isMatch = await bcrypt.compare(password, user.UserPassword) - Метод, который используется для сравнения введённого пользователем пароля с ранее захэшированным при регистрации паролем, хранящимся в базе данных.

password: Обычный текстовый пароль, введённый пользователем, который нужно проверить.

user.UserPassword: Захэшированный пароль, сохранённый в базе данных для данного пользователя.

При успешном результате, генерируется токен доступа который отправляется на клиент и записывается в Cookie.

Запрос в базу данных и обновление информации о пользователе при открытии страницы:

DataContext.js:

const router = useRouter()

const userPath = usePathname()

const [userData, setUserData] = useState(null);

function getCookie(name) {

const value = `; ${document.cookie}`;

const parts = value.split(`; ${name}=`);

if (parts.length === 2) return parts.pop().split(';').shift();

return null; // Если куки нет

}

function setCookie(name, value, days) {

const expires = new Date();

expires.setTime(expires.getTime() + (days \* 24 \* 60 \* 60 \* 1000));

const expiresStr = "expires=" + expires.toUTCString();

document.cookie = `${name}=${value}; ${expiresStr}; path=/`;

}

// Функция для получения данных о пользователе

function fetchUserData() {

const token = getCookie('token'); // Получаем токен из куки

return fetch('/api/login', {

method: 'GET',

headers: {

'Content-Type': 'application/json',

'Authorization': `Bearer ${token}`,

},

})

.then((response) => {

if (!response.ok) {

if (userPath.startsWith('/dashboard')) {

if (response.status === 401) {

router.push('/login');

}

}

}

else{

return response.json()

}

}).then((res)=>{

setCookie('token', res.newToken, 7)

setUserData(res.rows[0])

})

.catch((error) => {

console.error('Fetch error:', error);

});

}

useEffect(() => {

fetchUserData();

}, [userPath]);

Код проверяет токен авторизации пользователя, хранящийся в куках.

Если токен действителен, получает данные пользователя и обновляет токен.

Если токен недействителен или отсутствует, перенаправляет пользователя на страницу авторизации при попытке доступа к защищённым маршрутам

Rest api GET запроса:

async function getUserFromToken(token) {

try {

const decoded = jwt.verify(token, SECRET\_KEY);

const userId = decoded.id;

const [rows] = await pool.query('SELECT UserId, UserName, UserEmail, UserRole FROM users WHERE UserId = ?', [userId]);

const newToken = jwt.sign(

{

id: rows[0].UserId,

name: rows[0].UserName,

email: rows[0].UserEmail,

role: rows[0].UserRole,

},

SECRET\_KEY,

{ expiresIn: '7d' }

);

if (rows.length === 0) {

throw new Error('User not found');

}

return {newToken, rows}; // Возвращаем данные пользователя

} catch (error) {

throw new Error('Invalid token or user not found');

}

}

export async function GET(req) {

const token = req.headers.get('authorization')?.split(' ')[1];

if (!token) {

return new Response(JSON.stringify({ message: 'Tокен не предоставлен' }), { status: 401 });

}

try {

const user = await getUserFromToken(token);

return new Response(JSON.stringify(user), { status: 200 });

} catch (error) {

return new Response(JSON.stringify(), { status: 401 });

}

}

Функция GET(req):

const decoded = jwt.verify(token, SECRET\_KEY);

const userId = decoded.id;

Декодирование токена и извлечение из него id пользователя

Если токен отсутствует, возвращает ответ с кодом 401

Вызов getUserFromToken(token): Пытается получить данные пользователя, используя токен. Если всё успешно, возвращает данные пользователя и новый токен с кодом ответа 200.

Обработка ошибок: Если токен недействителен или пользователь не найден, возвращает ответ с кодом 401.

Основная логика: Код проводит аутентификацию пользователя с помощью JWT токена.

Если токен действителен, данные пользователя извлекаются из базы данных, и создаётся новый токен.

Если токен отсутствует или недействителен, возвращается ошибка аутентификации.

Такой запрос вызывается при каждом входе на страницу. И обновляет токен в Cookie пользователя, а так-же записывает полученные данные в переменную userData

DataContext.js - Контекст

export function DataProvider({ children }) {

const [userData, setUserData] = useState(null);

// Остальной код

return (

<DataContext.Provider value={{ userData }}>

{children}

</DataContext.Provider>

);

}

export function useData() {

return useContext(DataContext);

}

Layout.js

<DataProvider>

          <Header/>

          {children}

 </DataProvider>

DataProvider оборачивает дочерние компоненты чтобы они могли получить доступ к данным через контекст

**Регистрация нового пользователя**

Rest API:

const SECRET\_KEY = process.env.JWT\_SECRET\_KEY;

export async function POST(req) {

try {

const { name, email, password } = await req.json();

// Проверка, существует ли пользователь

const [rows] = await pool.query('SELECT \* FROM users WHERE UserEmail = ?', [email]);

if (rows.length > 0) {

return new Response(

JSON.stringify({ message: 'Пользователь уже существует' }),

{ status: 400 }

);

} else {

// Сохранение пользователя в базе данных

const hashedPassword = await bcrypt.hash(password, 10)

await pool.query(

'INSERT INTO users (UserEmail, UserPassword, UserName) VALUES (?, ?, ?)',

[email, hashedPassword, name]

);

// Получение сохраненного пользователя

const [newRows] = await pool.query('SELECT \* FROM users WHERE UserEmail = ?', [email]);

const user = newRows[0];

// Генерация токена

const token = jwt.sign(

{

id: user.UserId,

name: user.UserName,

email: user.UserEmail,

role: user.UserRole,

},

SECRET\_KEY,

{ expiresIn: '7d' }

);

return new Response(

JSON.stringify({ message: 'Готово', token }),

{ status: 201 }

);

}

} catch (err) {

return new Response(

JSON.stringify({ message: 'Ошибка' }),

{ status: 500 }

);

}

}

Описание работы запроса: От пользователя приходит Имя, Email и пароль. Далее, происходит проверка зарегистрированных пользователей по введенному Email. Если пользователь существует, сервер возвращает ошибку о том что пользователь уже существует. Если же совпадений не найдено, то происходит хеширование пароля и запись данных в базу данных. После чего, данные записываются в переменную user и происходит генерация jwt токена, в котором будет зашифрована информация: UserId, UserName, UserEmail, UserRole. JWT токен отправляется на клиент.

Клиент:

const router = useRouter()

const [name, setName] = useState('')

const [email, setEmail] = useState('')

const [password, setPassword] = useState('')

const [confirmPassword, setConfirmPassword] = useState('')

function setCookie(name, value, days) {

const expires = new Date();

expires.setTime(expires.getTime() + (days \* 24 \* 60 \* 60 \* 1000));

const expiresStr = "expires=" + expires.toUTCString();

document.cookie = `${name}=${value}; ${expiresStr}; path=/`;

}

async function register(e) {

e.preventDefault();

if (password === confirmPassword) {

try {

const response = await fetch('/api/sign-up', {

method: 'POST',

headers: {

'Content-Type': 'application/json',

},

body: JSON.stringify({ name, email, password }),

});

const res = await response.json();

if (response.ok) {

alert('Успешная регистрация');

setCookie('token', res.token, 7);

router.push('/dashboard')

} else {

alert(res.message);

}

} catch (error) {

alert('Ошибка: Сервер недоступен.');

}

} else {

alert('Пароли не совпадают');

}

}

Переменные name, email, password, confirmPassword приходят из инпута на фронтенде клиента, и отправляются на сервер. В случае успешного ответа от сервера (Ответ: 200), пользователю демонстрируется уведомление об успешной регистрации, и происходит запись токена в куки. После чего через метод router=useRouter происходит перенаправление на страницу ‘/dashboard’

Данные в профиле появляются благодаря контексту, который оборачивает все дочерние элементы на странице и не требует вызова сервера для каждого отдельного элемента использующего данные. Что позволяет снизить нагрузку на сервер.

import { useData } from "../components/DataContext";

export default function Account() {

    const { userData } = useData()

// остальной код

}

Каталог товаров

App/components/Catalog/Catalog.js

Страница товара, добавление товара в корзину

import ProductCard from "../ProductCard/ProductCard";

export default function Catalog() {

const[products, setProducts]=useState([])

useEffect(()=>{

fetch('/api/products/')

.then((res)=>{

return res.json()

}).then((results)=>{

setProducts(results)

console.log(results)

})

},[])

return (

<div className="container mx-auto flex flex-wrap gap-3">

{products && products.length !== undefined ? products.map((product)=>(

<ProductCard key={product.ProductId} name={product.ProductName}/>

)) : <div className="mx-auto font-bold text-xl mb-32">Ошибка сервера. Но мы уже работаем над этим...</div>}

</div>

);

}

При открытии страницы, 1 раз выполняется запрос на сервер. Полученные данные записываются переменную products. Через метод map() массив перебирается и для каждого объекта рендерится карточка продукта (ProductCard.js), в которую, с помощью пропсов передаются данные из item.

REST API: ‘/api/products.js’

export async function GET() {

try {

const [rows] = await pool.query('SELECT \* FROM products');

// Обрабатываем каждый продукт, преобразуя строку с картинками в массив

const products = rows.map(product => {

const images = product.ProductImages.replace(/"/g, '').split(','); // Преобразуем строку в массив

return {

...product,

ProductImagesArray: images, // Добавляем массив изображений

};

});

return new Response(

JSON.stringify(products),

{ status: 200, headers: { "Content-Type": "application/json" } }

);

} catch (err) {

console.error('Ошибка на сервере:', err);

return new Response(

JSON.stringify({ message: "Ошибка сервера" }),

{ status: 500, headers: { "Content-Type": "application/json" } }

);

}

}

Из интересного, стоит отметить преобразования строки с изображениями, разделенных запятыми – в массив, и добавление этого массива в обьект с товаром

App/[type]/[id]/page.js

export default function ProductPage() {

  const[productInfo, setProductInfo] = useState('')

  const pathname = usePathname()

  const lastSegment = pathname.split('/').filter(Boolean).pop()

  useEffect(()=>{

    fetch(`/api/products/${lastSegment}`)

    .then((res)=>{

      return res.json()

    })

    .then((result)=>{

      setProductInfo(result[0])

    })

  },[lastSegment])

  useEffect(()=>{

    console.log("DL",productInfo?.ProductImagesArray)

  },[productInfo])

  return (

    <div className="container mx-auto">

        <div className="flex justify-between">

            <ProductImages images={productInfo?.ProductImagesArray}/>

            <ProductInfo id={productInfo?.ProductId} name={productInfo?.ProductName} price={productInfo?.ProductPrice} descr={productInfo?.ProductDescription} sizeS={productInfo?.ProductSizeS} sizeM={productInfo?.ProductSizeM} sizeXL={productInfo?.ProductSizeXL} sizeXXL={productInfo?.ProductSizeXXL}/>

        </div>

    </div>

  );

}

lastSegment приходит из последнего элемента URL. Это ID товара. Далее происходит GET запрос на сервер с получением и записью данных в переменную productInfo.

REST API:

/api/products/[id]/route.js

export async function GET(request, { params }) {

try {

const pid = (await params).id

console.log(pid)

const [rows] = await pool.query('SELECT \* FROM products WHERE ProductId = ?',[pid]);

const product\_info = rows.map(product => {

const images = product.ProductImages.replace(/"/g, '').split(','); // Преобразуем строку в массив

return {

...product,

ProductImagesArray: images, // Добавляем массив изображений

};

});

return new Response(

JSON.stringify(product\_info),

{ status: 200, headers: { "Content-Type": "application/json" } }

);

} catch (error) {

console.error(error);

return NextResponse.json({ message: error.message }, { status: 500 });

}

}

params: Объект с параметрами маршрута. Здесь используется для получения id продукта.

Код получает идентификатор продукта из параметров запроса.

Выполняет запрос к базе данных, чтобы получить информацию о продукте.

Преобразует строку с изображениями в массив.

Возвращает JSON-ответ с информацией о продукте.

Корзина и оформление заказа

Добавление товара в корзину

App/components/ProductInfo/ProductInfo.js

function getCookie(name) {

const value = `; ${document.cookie}`;

const parts = value.split(`; ${name}=`);

if (parts.length === 2) return parts.pop().split(';').shift();

return null;

}

const [selectedSize, setSelectedSize] = useState('');

const handleChange = (value) => {

setSelectedSize(value);

}

function addCartItem(){

const token = getCookie('token');

fetch(`/api/cart`,{

method: 'POST',

headers: {

'Content-Type': 'application/json',

'Authorization': `Bearer ${token}`,

},

body: JSON.stringify({

productId: props.id,

productCount: 1,

productSize: selectedSize

})

}

)

}

С помощью пропсов, на страницу приходят данные с информацией о товаре, на сервер отправляется POST запрос с идентификатором продукта, количеством в размере 1шт и размером, который выбрал пользователь на фронтенде. В заголовке приходит токен. Который декодируется на сервере, получает id пользователя, и создает запись в таблице «cart» с идентификатором пользователя, товара, количеством и размером. Для каждого товара, отдельная запись

Rest api:

export async function POST(req) {

const token = req.headers.get('authorization')?.split(' ')[1];

if (!token) {

return new Response(JSON.stringify({ message: 'Tокен не предоставлен' }), { status: 401 });

}

try {

// Извлекаем данные из тела запроса

const { productId, productCount, productSize } = await req.json();

if (!productId) {

return new Response(JSON.stringify({ message: 'Не указан productId' }), { status: 400 });

}

const product = await addCartItem(token, productId, productCount, productSize);

return new Response(JSON.stringify(product), { status: 200 });

} catch (error) {

return new Response(JSON.stringify({ message: error.message }), { status: 500 });

}

}

Код проверяет наличие токена и извлекает данные о товаре из тела запроса.

Проверяет наличие обязательного параметра productId.

Вызывает функцию addCartItem для добавления товара в корзину.

Возвращает JSON-ответ с данными о добавленном товаре или ошибкой.

async function addCartItem(token, productId, productCount, productSize) {

    try {

        // Расшифровка токена

        const decoded = jwt.verify(token, SECRET\_KEY);

        const userId = decoded.id;

        // Добавляем позицию в корзину

        await pool.query(

            "INSERT INTO cart (UserId, ProductId, ProductCount, ProductSize) VALUES (?, ?, ?, ?)",

            [userId, productId, productCount, productSize]

        );

        return new Response(JSON.stringify({ message: "Позиция успешно добавлена" }), {

            status: 200,

        });

    } catch (error) {

        return new Response(

            JSON.stringify({ error: "Ошибка сервера" }),

            { status: 500 }

        );

    }

}

По аналогичному принципу работает и удаление товара из корзины

Создание заказа.

Фронтенд:

const router = useRouter()

const [cart, setCart] = useState([])

function getCookie(name) {

const value = `; ${document.cookie}`;

const parts = value.split(`; ${name}=`);

if (parts.length === 2) return parts.pop().split(';').shift();

return null; // Если куки нет

}

function getCart(){

const token = getCookie('token');

fetch('/api/cart', {

method: 'GET',

headers: {

'Content-Type': 'application/json',

'Authorization': `Bearer ${token}`,

}

}

)

.then((res)=>{

return res.json()

})

.then((items)=>{

if(items.length >= 1){

setCart(items)

}

})

}

useEffect(()=>{

getCart()

},[])

const [isSuccessDialogOpen, setSuccessDialogOpen] = useState(false)

const [isErrorDialogOpen, setErrorDialogOpen] = useState(false)

async function createOrder() {

const token = getCookie('token');

const orderData = {

products: cart.map(item => ({

productId: item.ProductId,

productCount: item.ProductCount,

productSize: item.ProductSize

}))

};

try {

const response = await fetch('/api/orders/get-orders', {

method: 'POST',

headers: {

'Content-Type': 'application/json',

'Authorization': `Bearer ${token}`

},

body: JSON.stringify(orderData)

});

if (!response.ok) {

const errorData = await response.json();

throw new Error(errorData.message || 'Failed to create order');

}

const result = await response.json();

setSuccessDialogOpen(true)

} catch (error) {

setErrorDialogOpen(true)

}

}

Функция getCart выполняет получение добавленных в корзину товаров и записывает их в переменную cart. Далее, выполняется функция отправки заказа в базу данных. По аналогии с предыдущими функциями, в headers отправляется токен с закодированными данными. В объекте orderData, функции createOrder хранится массив products с товаром записанным в переменной cart.

REST API:

async function createOrder(token, orderData) {

try {

const decoded = jwt.verify(token, SECRET\_KEY);

const userId = decoded.id;

// Вставка нового заказа в таблицу orders

const [orderResult] = await pool.query('INSERT INTO orders (UserId, OrderDate, OrderStatus) VALUES (?, NOW(), ?)', [userId, 'Создан']);

const orderId = orderResult.insertId;

// Вставка продуктов в таблицу order\_products

const orderProducts = orderData.products.map(product => [orderId, product.productId, product.productCount, product.productSize]);

// Очистка корзины

await pool.query('INSERT INTO order\_products (OrderId, ProductId, ProductCount, ProductSize) VALUES ?', [orderProducts]);

await pool.query('DELETE FROM `cart` WHERE UserId = ?', userId)

return { orderId, message: 'Order created successfully'};

} catch (error) {

throw new Error(`Error creating order: ${error.message}`);

}

}

Проходит верификация токена и извлечение userId из декодированного токена.

Вставка нового заказа в таблицу orders через запрос

pool.query('INSERT INTO orders (UserId, OrderDate, OrderStatus) VALUES (?, NOW(), ?)', [userId, 'Создан']):

Добавляет новый заказ в таблицу orders с текущей датой (NOW()) и статусом "Создан".

orderResult.insertId: Получает идентификатор нового заказа.

Вставка продуктов в таблицу order\_products:

Создаёт массив orderProducts, содержащий информацию о продуктах (идентификатор заказа, идентификатор продукта, количество, размер).

Вставляет эти данные в таблицу order\_products с помощью await pool.query('INSERT INTO order\_products (OrderId, ProductId, ProductCount, ProductSize) VALUES ?', [orderProducts]);

await pool.query('DELETE FROM `cart` WHERE UserId = ?', userId) - Удаляет все товары из таблицы cart для текущего пользователя (userId).

Админ панель

Реализация доступа администратора

App/dashboard/(admin)/layout.js

const router = useRouter()

const { userData } = useData()

useEffect(()=>{

userData?.UserRole != 'admin' ? router.push('/dashboard/account') : null

},[userData])

Если роль пользователя не равна admin. Происходит перенаправление на страницу обычного пользователя.

На странице администратора есть 3 кнопки. «Пользователи» “Товары» «Заказы»

Каждая кнопка отвечает за рендеринг определенного контента на странице.

Управление пользователями:

При нажатии на кнопку «Пользователи» рендерится таблица с зарегистрированными в системе пользователями (рис: #)

const[usersData, setUsersData]=useState([])

useEffect(()=>{

const token = getCookie('token')

fetch('/api/users',{

method: "GET",

headers: {

'Content-Type': 'application/json',

'Authorization': `Bearer ${token}`,

}

}).then((res)=>{

return res.json()

}).then((result)=>{

setUsersData(result)

})

},[])

async function getUserFromToken(token) {

const decoded = jwt.verify(token, SECRET\_KEY);

const role = decoded.role;

if(role === 'admin'){

const [rows] = await pool.query('SELECT \* FROM users');

return rows;

}else{

return new Response(JSON.stringify([]), { status: 200 });

}

}

Классический REST API запрос, с небольшим отличием - проверкой роли пользователя из декодированного токена. Такая проверка присутсвует на всех серверных запросах, относящиеся к работе администратора.

Страница пользователя (Приложение-А: рис.12.1(б))

App/dashboard/admin/user/page.js

Получаем id пользователя из URL формата: «/dashboard/admin/user#ID»

const pathname = usePathname()

    const [hash, setHash] = useState('')

    useEffect(() => {

      const handleHashChange = () => {

        const urlSliced = window.location.hash.slice(1)

        setHash(Number(urlSliced));

      };

      window.addEventListener('hashchange', handleHashChange);

      handleHashChange();

      return () => {

      window.removeEventListener('hashchange', handleHashChange);

      };

    }, [pathname])

window.location.hash.slice(1): Получает текущий хэш из URL и удаляет символ # с помощью slice(1).

setHash(Number(urlSliced)) преобразует строку хэша в число с помощью Number() и обновляет состояние через функцию setHash

window.addEventListener('hashchange', handleHashChange): Добавляет обработчик события hashchange, который вызывается при изменении хэша в URL. При изменении хэша вызывается handleHashChange

Получение данных пользователя:

const[userData, setUserData]=useState([])

useEffect(() => {

const token = getCookie('token')

if(hash != 0){

fetch(`/api/users/${hash}`,{

method: "GET",

headers: {

'Content-Type': 'application/json',

'Authorization': `Bearer ${token}`,

}

}).then((res)=>{

return res.json()

}).then((result)=>{

setUserData(result)

})

}

}, [hash])

**REST API**

export async function GET(req, { params }) {

const token = req.headers.get('authorization')?.split(' ')[1];

const decoded = jwt.verify(token, SECRET\_KEY);

const reqId = decoded.id;

const [role] = await pool.query('SELECT UserRole FROM users WHERE UserId = ?', [reqId]);

if (role[0].UserRole != 'admin') {

return new Response(JSON.stringify({ message: 'ПОшОл отсюдова}), { status: 401 });

}

try {

const userId = (await params).id

const [user] = await pool.query(

"SELECT \* FROM users WHERE UserId = ?",

[userId]

);

// Запрос на данные о заказах

const [orders] = await pool.query(

`SELECT orders.OrderId, COUNT(order\_products.ProductId) AS TotalItems, SUM(products.ProductPrice \* order\_products.ProductCount) AS TotalPrice, orders.OrderDate, orders.OrderStatus FROM orders JOIN order\_products ON orders.OrderId = order\_products.OrderId JOIN products ON order\_products.ProductId = products.ProductId WHERE orders.UserId = ? GROUP BY orders.OrderId, orders.OrderDate, orders.OrderStatus;`,

[userId]

);

const grandTotal = orders.reduce((sum, order) => sum + parseFloat(order.TotalPrice || 0), 0).toFixed(2);

const responseData = {

user: user[0],

orders,

grandTotal

};

return new Response(

JSON.stringify(responseData),

{ status: 200, headers: { "Content-Type": "application/json" } }

);

} catch (error) {

console.error(error);

return NextResponse.json({ message: error.message }, { status: 500 });

}

}

Получение и проверка токена:

1. Токен извлекается из заголовков запроса authorization.
2. Токен декодируется с помощью jwt.verify для получения ID пользователя.
3. Выполняется запрос к базе данных, чтобы получить роль пользователя по его ID.
4. Проверка роли пользователя:

Если роль пользователя не admin, возвращается статусом 401. Эта часть кода, включая сообщение.

1. Получение данных о пользователе и его заказах:

Выполняется запрос на получение информации о пользователе по userId.

Далее, с помощью SQL-запроса извлекаются данные о заказах пользователя, включая количество товаров в каждом заказе, общую стоимость, дату заказа и его статус

Общая сумма всех заказов (grandTotal) рассчитывается с использованием метода reduce

Сформированный объект responseData содержит информацию о пользователе, его заказы и общую сумму. Данные сериализуются в JSON и отправляются клиенту

Доступ к открытым заказам:

Администратор имеет доступ к созданным заказам. (Табл. 1)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№ Заказа** | **Дата заказа** | **Позиций** | **Сумма, ₽** | **Статус** |
|  |  |  |  |  |
| Артикул | Наименование | Размер | Количество |
|  |  |  |  |
|  |  |  |  |

(Табл. 1)

А так-же возможность изменять статус заказа и выпадающего HTML списка <select></select > на фронтенде клиента (Приложение-А Рис. 12.3(б))

Программный код

useEffect(()=>{

const token = getCookie('token');

fetch('/api/orders/get-orders/admin', {

method: 'GET',

headers: {

'Content-Type': 'application/json',

'Authorization': `Bearer ${token}`,

}

}

)

.then((res)=>{

return res.json()

})

.then((orders)=>{

setOrders(orders)

})

},[])

const handleStatusChange = async (orderId, newStatus) => {

try {

const token = getCookie('token');

const response = await fetch(`/api/orders/get-orders/admin`, {

method: 'PATCH',

headers: {

'Content-Type': 'application/json',

'Authorization': `Bearer ${token}`,

},

body: JSON.stringify({ orderId, newStatus })

});

if (response.ok) {

// Обновление статуса в локальном состоянии

setOrders(prevOrders =>

prevOrders.map(order =>

order.OrderId === orderId ? { ...order, OrderStatus: newStatus } : order

)

);

toast(`Статус заказа №${orderId} обновлен на ${newStatus}`)

} else {

console.error('Ошибка обновления статуса заказа');

}

} catch (error) {

console.error('Ошибка при отправке запроса на сервер:', error);

}

};

Изначально выполняется получение всех заказов из базы данных через GET запрос на сервер и передачей в заголовке токена доступа.

REST API, GET запрос:

async function getUserFromToken(token) {

const decoded = jwt.verify(token, SECRET\_KEY);

const userRole = decoded.role;

if(UserRole === 'admin'){

const [rows] = await pool.query(`

SELECT orders.OrderId, orders.UserId, users.UserName, orders.OrderDate, orders.OrderStatus,

products.ProductId, products.ProductName, products.ProductPrice, order\_products.ProductSize,

order\_products.ProductCount

FROM orders

JOIN order\_products ON orders.OrderId = order\_products.OrderId

JOIN products ON order\_products.ProductId = products.ProductId

JOIN users ON orders.UserId = users.UserId

ORDER BY orders.OrderDate;

`);

const ordersMap = {};

rows.forEach(row => {

const { OrderId, UserId, UserName, OrderDate, OrderStatus, ProductId, ProductName, ProductPrice, ProductSize, ProductCount } = row;

if (!ordersMap[OrderId]) {

ordersMap[OrderId] = {

OrderId,

UserId,

UserName,

OrderDate,

OrderStatus,

TotalPrice: 0,

Products: []

};

}

const productTotalPrice = ProductPrice \* ProductCount;

ordersMap[OrderId].TotalPrice += productTotalPrice;

ordersMap[OrderId].Products.push({

ProductId,

ProductName,

ProductSize,

ProductCount,

ProductPrice,

ProductTotalPrice: productTotalPrice

});

});

const orders = Object.values(ordersMap);

return orders;

}else{

return new Response(JSON.stringify([]), { status: 200 });

}

}

Токен декодируется с помощью jwt.verify для получения роли пользователя.

Если роль пользователя — admin, функция переходит к получению заказов.

Т.е, выполняется SQL-запрос, объединяющий несколько таблиц (orders, order\_products, products, users), чтобы получить детальную информацию о заказах, включая информацию о пользователях, продуктах, количестве товаров и цене.

При формировании структуры заказов используется объект ordersMap для группировки заказов по OrderId.

Если заказ с таким OrderId еще не существует в ordersMap, создается новый объект заказа с базовой информацией.

Затем информация о каждом продукте добавляется в соответствующий заказ, а также обновляется общая стоимость заказа (TotalPrice).

После чего происходит конвертация заказа в более удобный вид для работы, ordersMap преобразуется в массив объектов заказов с помощью Object.values(ordersMap), который возвращается на клиент.

Изменение статуса заказа. Запрос REST API  
export async function PATCH(req) {

const token = req.headers.get('authorization')?.split(' ')[1];

if (!token) {

console.log('Токен не предоставлен');

return new Response(JSON.stringify({ message: 'Tокен не предоставлен' }), { status: 401 });

}

const decoded = jwt.verify(token, SECRET\_KEY);

const role = decoded.role;

if(role === 'admin'){

const { orderId, newStatus } = await req.json();

if (!orderId || !newStatus) {

return new Response(JSON.stringify({ message: 'Необходимы orderId и newStatus' }), { status: 400 });

}

const [result] = await pool.query(

`UPDATE orders SET OrderStatus = ? WHERE OrderId = ?`,

[newStatus, orderId]

);

if (result.affectedRows > 0) {

return new Response(JSON.stringify({ message: 'Статус заказа обновлен' }), { status: 200 });

} else {

return new Response(JSON.stringify({ message: 'Не удалось обновить статус заказа' }), { status: 500 });

}

}else{

return new Response(JSON.stringify([]), { status: 200 });

}

}

Извлечение и проверка токена проводится аналогично предыдущему коду с работой администратора. И далее описываться не будут.

Данные orderId и newStatus извлекаются из тела запроса. Если хотя бы одно из значений отсутствует, возвращается ответ с сообщением "Необходимы orderId и newStatus" и статусом 400.

При обновлении статуса заказа, выполняется SQL-запрос для обновления статуса заказа по указанному orderId.

Если количество затронутых строк больше 0 (result.affectedRows > 0), возвращается сообщение об успешном обновлении статуса с кодом состояния 200.

Если обновление не удалось (например, заказ с таким orderId не найден), возвращается сообщение об ошибке

Получение/Изменение/Добавление/Удаление Товаров

1. **Получение**

Вид таблицы товаров (Табл. 2)

**Категория**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Название** | **Описание** | **Размер S** | **Размер M** | **Размер XL** | **Размер XXL** | Рейтинг | Стоимость | Действие |
|  |  |  |  |  |  |  |  |  |
| Изображение 1, Изображение 2, Изображение .. | | | | | | | | |

const[products, setProducts]=useState([])

function getProducts(){

fetch('/api/products/')

.then((res)=>{

return res.json()

}).then((results)=>{

const groupedByCategory = results.reduce((acc, product) => {

const category = product.ProductCategory;

// Если категории еще нет, создаем массив

if (!acc[category]) {

acc[category] = [];

}

// Добавляем продукт в категорию

acc[category].push(product);

return acc;

}, {});

setProducts(groupedByCategory)

console.log("DDD", groupedByCategory)

})

}

Функция getProducts выполняет запрос к API /api/products/ для получения списка продуктов и преобразует ответ в JSON.

В следующем .then((results) => {...}) происходит обработка полученного массива продуктов results.

Данные приходят в виде:

[

{ "ProductName": "Футблока поло", "ProductCategory": "T-shirt" },

{ "ProductName": "Штаны карго", "ProductCategory": "Pants" },

{ "ProductName": "Штаны Классика", "ProductCategory": "Pants" },

]

Группировка продуктов по категориям:

results.reduce((acc, product) => {...}, {}); - метод reduce используется для группировки продуктов по категории.

const category = product.ProductCategory; - извлекается категория продукта.

Проверка: если категории еще нет в аккумуляторе acc, создается новый массив для этой категории.

Продукт добавляется в соответствующий массив категории.

Итоговый объект groupedByCategory получает вид:

{

" T-shirt ": [

{ "ProductName": " Футблока поло", "ProductCategory": "T-shirt" },

],

" Pants ": [

{ "ProductName": " Штаны карго", "ProductCategory": " Pants " }

{ "ProductName": " Штаны Классика", "ProductCategory": " Pants " }

]

}

setProducts(groupedByCategory) обновляет состояние products с сгруппированными данными

REST API:

«/api/products»

export async function GET() {

try {

const [rows] = await pool.query('SELECT \* FROM products');

// Обрабатываем каждый продукт, преобразуя строку с картинками в массив

const products = rows.map(product => {

const images = product.ProductImages.replace(/"/g, '').split(','); // Преобразуем строку в массив

return {

...product,

ProductImagesArray: images, // Добавляем массив изображений

};

});

return new Response(

JSON.stringify(products),

{ status: 200, headers: { "Content-Type": "application/json" } }

);

} catch (err) {

console.error('Ошибка на сервере:', err);

return new Response(

JSON.stringify({ message: "Ошибка сервера" }),

{ status: 500, headers: { "Content-Type": "application/json" } }

);

}

}

Здесь мы преобразуем строку с расположениями изображений из БД, разделенных запятыми, в массив

const products = rows.map(product => {

const images = product.ProductImages.replace(/"/g, '').split(',');

return {

...product,

ProductImagesArray: images, // Добавляем массив изображений

};

1. **Изменение**

На странице реализованы интерактивные текстовые поля, которые позволяют редактировать значения прямо на странице, а затем отправлять обновления на сервер. Вот как это реализовано

Программный код:

function EditableDescription({ description, onUpdate }) {

const [isEditing, setIsEditing] = useState(false);

const [value, setValue] = useState(description);

const handleEditClick = () => setIsEditing(true);

const handleInputChange = (e) => setValue(e.target.value);

const handleBlur = () => {

setIsEditing(false);

onUpdate(value); // Обновление значения

};

return (

<div className="inline" onClick={handleEditClick}>

{isEditing ? (

<textarea

value={value}

onChange={handleInputChange}

onBlur={handleBlur}

className="field-sizing-content w-full h-max"

/>

) : (

<span>{value || 0}</span>

)}

</div>

);

}

const handleUpdate = async (productId, field, newValue) => {

try {

const token = getCookie('token')

const response = await fetch(`/api/products/${productId}`, {

method: 'PATCH', // Используем метод PATCH для частичного обновления

headers: {

'Content-Type': 'application/json',

'Authorization': `Bearer ${token}`

},

body: JSON.stringify({ [field]: newValue }),

});

if (!response.ok) {

throw new Error(`Ошибка при обновлении: ${response.statusText}`);

}

} catch (error) {

console.error('Ошибка при обновлении данных:', error);

}

};

Фронтенд:

<td className="table-cell p-1 w-1/5 border-x border-b border-stone-700">

<EditableDescription

description={item?.ProductName}

onUpdate={(newProductName) => handleUpdate(item?.ProductId, 'ProductName', newProductName)}

/>

</td>

<td className="table-cell p-1 w-1/2 border-r border-b border-stone-700" >

<EditableDescription

description={item?.ProductDescription}

onUpdate={(newDescription) => handleUpdate(item?.ProductId, 'ProductDescription', newDescription)} />

</td>

Компонент EditableDescription

Состояния:

isEditing: Булевое состояние, определяющее, находится ли компонент в режиме редактирования.

value: Состояние для хранения текущего значения описания.

Обработчики событий:

handleEditClick: Переключает режим в редактирование при клике.

handleInputChange: Обновляет value при изменении значения в текстовом поле.

handleBlur: Отключает режим редактирования и вызывает onUpdate для передачи нового значения родительскому компоненту.

Отображение:

В режиме редактирования отображается текстовое поле (textarea).

В режиме просмотра отображается текущее значение (value), обернутое в span.

Функция handleUpdate

Получает productId, field, и newValue для обновления конкретного поля продукта. Затем, выполняет запрос PATCH к API /api/products/${productId}, передавая обновленное значение в теле запроса.

Обработка ошибок:

Если запрос не успешен, вызывается всплывающее окно «Toast»

Описание фронтенда

EditableDescription используется для отображения и редактирования названия продукта (ProductName) и его описания (ProductDescription). При обновлении значения которого, вызывается handleUpdate, которая отправляет изменения на сервер.

Поведение пользователя:

Пользователь может кликнуть по тексту в ячейке, чтобы начать редактирование.

После ввода нового значения и выхода из поля (событие onBlur), происходит отправка изменений на сервер.

При успешном обновлении сервер возвращает подтверждение

Сервреный код:

export async function PATCH(request, { params }) {

try {

const pId = params.id;

const updates = await request.json();

if (!updates || Object.keys(updates).length === 0) {

return new Response(JSON.stringify({ message: 'Нет данных для обновления' }), { status: 400 });

}

const token = request.headers.get('authorization')?.split(' ')[1];

const decoded = jwt.verify(token, SECRET\_KEY);

const role = decoded.role;

if (role !== 'admin') {

return new Response(JSON.stringify({ message: 'Доступ запрещен' }), { status: 401 });

}

const fields = Object.keys(updates)

.map((field) => `${field} = ?`)

.join(', ');

const values = Object.values(updates);

await pool.query(`UPDATE products SET ${fields} WHERE ProductId = ?`, [...values, pId]);

return new Response(JSON.stringify({ message: 'Продукт обновлен успешно' }), { status: 200 });

} catch (error) {

return new Response(JSON.stringify({ message: error.message }), { status: 500 });

}

}

Как устроен принцип работы кода:

Приходит идентификатор продукта:

pId извлекается из params.id.

updates извлекается из тела запроса с помощью request.json(), и, проверяется, есть ли данные для обновления. Если их нет, возвращается ответ с сообщением об ошибке и статусом 400.

Формирование запроса к базе данных:

Список полей для обновления формируется в виде строки SQL. Например:

FIELDS: ProductSizeXXL = ?

VALUES: [ '9' ]

Значения для обновления извлекаются и передаются в массиве values.

Выполняется запрос на обновление данных в таблице products

1. **Добавление**

В данном коде реализована форма для добавления нового продукта в систему. Она использует состояние React для управления значениями ввода, такими как название продукта, описание, категория, цена и доступные размеры. Также предусмотрено добавление изображений для продукта через файловые поля. Код отправляет эти данные на сервер в виде формы, используя API POST запрос с типом контента multipart/form-data.

const [sizes, setSizes] = useState({

sizeS: '',

sizeM: '',

sizeXL: '',

sizeXXL: '',

})

const handleChange = (event) => {

const { id, value } = event.target;

setSizes((prevSizes) => ({

...prevSizes,

[id]: value, // Обновляем соответствующее поле

}));

};

const [productName, setProductName] = useState('')

const [productDescr, setProductDescr] = useState('')

const [productCategory, setProductCategory] = useState('')

const [productPrice, setProductPrice] = useState('')

const [selectedFiles, setSelectedFiles] = useState([]);

const [isSuccessDialogOpen, setSuccessDialogOpen] = useState(false)

const [isErrorDialogOpen, setErrorDialogOpen] = useState(false)

const [progressValue, setProgressValue] = useState(0)

const [backdropProgress, setBackdropProgress] = useState('invisible')

const handleCategoryChange = (value) => {

setProductCategory(value); // Обновляем состояние выбранной категории

}

const handleFileChange = (event) => {

// Получаем все выбранные файлы

setSelectedFiles(event.target.files);

};

const handleSubmit = async (event) => {

event.preventDefault();

if (!productCategory) {

alert('Пожалуйста, выберите категорию!');

return;

}

setBackdropProgress('visible')

const token = getCookie('token');

const sizesArray = Object.values(sizes).filter(size => size !== ''); // Фильтруем пустые значения

console.log(sizesArray);

const formData = new FormData();

setProgressValue(30)

// Добавляем данные в FormData

formData.append('name', productName);

formData.append('descr', productDescr);

formData.append('category', productCategory);

formData.append('price', productPrice);

formData.append('sizes', sizesArray);

Array.from(selectedFiles).forEach((file, index) => {

formData.append('files', file); // "files" — это имя поля на сервере, через которое будет получен массив файлов

});

console.log('data', formData);

setProgressValue(80)

try {

await fetch('/api/products/add', {

method: 'POST',

headers: {

'Authorization': `Bearer ${token}`,

},

body: formData

}).then(() => {

// Обнуление полей формы

setProductName(''); // Обнуление названия продукта

setProductDescr(''); // Обнуление описания продукта

setProductCategory(''); // Обнуление категории

setProductPrice(''); // Обнуление цены

setSizes({ // Обнуление размеров

sizeS: '',

sizeM: '',

sizeXL: '',

sizeXXL: '',

});

setSelectedFiles([]); // Обнуление файлов

document.getElementById('picture').value = '';

setProgressValue(100)

setBackdropProgress('invisible')

setSuccessDialogOpen(true);

}).catch(() => {

setBackdropProgress('invisible')

setErrorDialogOpen(true);

});

} catch (error) {

console.error('Error uploading files:', error);

}

};

1. Использование состояния React (useState)

В коде используется хук useState для управления состоянием формы. Каждое поле формы, включая текстовые поля (например, название, описание) и поля для выбора файлов, имеют свое начальное состояние в виде пустых строк и массивов.

Для каждого поля формы, например, для размеров, используется функция для обновления состояния:

const handleChange = (event) => {

const { id, value } = event.target;

setSizes((prevSizes) => ({

...prevSizes,

[id]: value,

}));

};

Этот обработчик получает новое значение поля и обновляет соответствующее состояние для размеров, таким образом, создавая динамическое изменение данных формы в зависимости от ввода пользователя.

Одним из важнейших возможностей формы является возможность загрузки файлов изображений товара. Для этого используется элемент формы типа file, а файлы сохраняются в массив selectedFiles

При отправке формы (обработчик handleSubmit), данные из формы обрабатываются и собираются в объект FormData, который поддерживает отправку данных, включая файлы, на сервер

const formData = new FormData();

formData.append('name', productName);

formData.append('descr', productDescr);

formData.append('category', productCategory);

formData.append('price', productPrice);

formData.append('sizes', sizesArray);

Array.from(selectedFiles).forEach((file) => {

formData.append('files', file);

});

Здесь создается экземпляр FormData, в который добавляются все данные формы. Для файлов используется метод append для добавления каждого файла в форму

Отправка данных на сервер выполняется через асинхронную функцию с использованием fetch. В данном случае отправка выполняется с методом POST и заголовками, включая авторизационный токен

Этот запрос отправляет все данные на сервер, где они будут обработаны

При отправке формы также отслеживается прогресс загрузки. Это важно для UX, чтобы пользователь понимал, что процесс идет. В коде используется состояние progressValue для отображения прогресса. За это отвечает UI элемент tailwind/shadcn <Progress/>

REST API:

export const config = {

api: {

bodyParser: false, // Отключаем bodyParser для работы с большими файлами

},

};

export async function POST(req, res) {

try {

{Код обработки токена}

if (role !== 'admin') {

return new Response(JSON.stringify({ message: 'Доступ запрещен' }), { status: 403 });

}

const formData = await req.formData(); // Получаем formData

const name = formData.get('name');

const descr = formData.get('descr');

const category = formData.get('category');

const price = formData.get('price');

const sizeRaw = formData.getAll('sizes'); // Возвращает массив значений для ключа 'category'

const sizes = sizeRaw.flatMap(cat => cat.split(','));

const files = formData.getAll('files'); // Получаем все файлы

if (!files.length) {

return new Response(JSON.stringify({ error: 'Файлы не были отправлены' }), { status: 400 });

}

// Генерация случайного имени папки

const randomFolderName = crypto.randomBytes(16).toString('hex');

const folderPath = path.join(process.cwd(), 'public/products', randomFolderName);

// Проверяем и создаем папку

if (!fs.existsSync(folderPath)) {

fs.mkdirSync(folderPath, { recursive: true });

}

// Массив для хранения относительных путей файлов

const filePaths = [];

// Сохранение файлов

for (let i = 0; i < files.length; i++) {

const file = files[i];

const fileExtension = path.extname(file.name); // Расширение файла

const fileName = `${i + 1}${fileExtension}`; // Имя файла

const filePath = path.join(folderPath, fileName);

// Записываем файл

await fs.promises.writeFile(filePath, Buffer.from(await file.arrayBuffer()));

// Извлечение относительного пути начиная с 'products/'

const relativePath = path.relative(path.join(process.cwd(), 'public'), filePath);

// Добавляем относительный путь в массив

filePaths.push(`/${relativePath.replace(/\\/g, '/')}`); // Добавляем '/' перед относительным путем

}

// Преобразование массива путей в строку

const filePathsString = filePaths.join(',');

// Пример вставки в базу данных

await pool.query('INSERT INTO products (ProductName, ProductCategory, ProductImages, ProductDescription, ProductSizeM, ProductSizeS, ProductSizeXL, ProductSizeXXL, ProductPrice) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)', [name, category, filePathsString, descr, sizes[0], sizes[1], sizes[2], sizes[3], price]);

return new Response(JSON.stringify({ message: 'Файлы успешно загружены', folder: randomFolderName }), { status: 200 });

} catch (error) {

console.error('Ошибка при загрузке файлов:', error);

return new Response(

JSON.stringify({ error: 'Ошибка загрузки файлов', details: error.message }),

{ status: 500 }

);

}

}

Данный код представляет собой обработчик POST-запроса для загрузки данных о продукте, включая файлы, в базу данных. Этот процесс включает несколько ключевых этапов:

1. Отключение bodyParser

В конфигурации API установлен параметр bodyParser: false, чтобы отключить стандартный парсинг тела запроса. Это необходимо, так как мы работаем с большими файлами, и используем formData для обработки данных формы

2. Получение и обработка данных из формы

После отключения bodyParser, данные формы получаются с помощью метода req.formData(). Это позволяет извлекать как текстовые данные, так и файлы, загруженные пользователем

const formData = await req.formData(); // Получаем formData

const name = formData.get('name');

const descr = formData.get('descr');

Далее извлекается массив с размерами товаров и файлы, прикрепленные к форме

const sizeRaw = formData.getAll('sizes'); // Получаем все размеры

const sizes = sizeRaw.flatMap(cat => cat.split(','));

const files = formData.getAll('files'); // Получаем все файлы

Для каждого нового продукта создается уникальная папка с помощью генерации случайного имени с использованием криптографической функции. Эта папка используется для хранения файлов (/public/products)

const randomFolderName = crypto.randomBytes(16).toString('hex');

const folderPath = path.join(process.cwd(), 'public/products', randomFolderName);

// Проверяем и создаем папку

if (!fs.existsSync(folderPath)) {

fs.mkdirSync(folderPath, { recursive: true });

}

Каждый файл из формы сохраняется в созданную папку. Имя файла генерируется на основе индекса, чтобы избежать конфликтов имен. После того как файл записан, его относительный путь добавляется в массив filePaths:

const filePaths = [];

for (let i = 0; i < files.length; i++) {

const file = files[i];

const fileExtension = path.extname(file.name); // Расширение файла

const fileName = `${i + 1}${fileExtension}`; // Имя файла

const filePath = path.join(folderPath, fileName);

await fs.promises.writeFile(filePath, Buffer.from(await file.arrayBuffer()));

const relativePath = path.relative(path.join(process.cwd(), 'public'), filePath);

filePaths.push(`/${relativePath.replace(/\\/g, '/')}`);

}

После загрузки файлов в директорию и получения их путей, информация о продукте сохраняется в базе данных. В данном случае, помимо данных о названии, категории, описании и цене, сохраняются пути к изображениям. Эти данные добавляются в таблицу products

**Удаление товара**

Удаление происходит на странице со списком всех товаров, нажатием кнопки «удалить»

async function deleteItem(item) {

const token = getCookie('token');

try {

const response = await fetch('/api/products/remove', {

method: 'DELETE',

headers: {

'Content-Type': 'application/json',

'Authorization': `Bearer ${token}`

},

body: JSON.stringify(item)

});

if (!response.ok) {

const errorData = await response.json();

throw new Error(errorData.message || 'Ошибка удаления');

}

toast(`Успешно удалено!`)

getProducts()

} catch (error) {

alert(error)

}

}

<Button className="w-full" variant="destructive" onClick={()=>{deleteItem(item?.ProductId)}}>Удалить</Button>

При нажатии на кнопку, в функцию отправляется идентификатор продукта. Далее выполняется запрос к серверу с запросом DELETE на адрес /api/products/remove, чтобы удалить продукт. Тело запроса содержит идентификатор продукта, который нужно удалить.

После отправки запроса, выполняется проверка ответа от сервера. Если запрос выполнен успешно, выполняется отображение сообщения с помощью toast библиотеки tailwind/shadcn (Приложение-А: рис. 8)

Сервер:

export async function DELETE(req) {

  try {

    const token = req.headers.get('authorization')?.split(' ')[1];

    const productId = await req.json()

    console.log(productId);

    if (!token) {

      console.log('Токен не предоставлен');

      return new Response(JSON.stringify({ message: 'Токен не предоставлен' }), { status: 401 });

    }

    let decoded;

    try {

      decoded = jwt.verify(token, SECRET\_KEY);

    } catch (err) {

      console.error('Ошибка декодирования токена:', err);

      return new Response(JSON.stringify({ message: 'Неверный токен' }), { status: 401 });

    }

    const role = decoded.role;

    if (role !== 'admin') {

      return new Response(JSON.stringify({ message: 'Доступ запрещен' }), { status: 403 });

    }

    const result = await pool.query('DELETE FROM products WHERE ProductId = ?', productId);

    if (result.rowCount === 0) {

      return new Response(JSON.stringify({ message: 'Продукт не найден' }), { status: 404 });

    }

    return new Response(JSON.stringify({ message: 'Продукт успешно удален' }), { status: 200 });

  } catch (error) {

    console.error('Ошибка сервера:', error);

    return new Response(

      JSON.stringify({ error: 'Произошла ошибка при удалении товара' }),

      { status: 500 }

    );

  }

}

Принцип работы: Из тела запроса извлекается ID продукта, который необходимо удалить. Он передается в теле запроса как JSON объект

const productId = await req.json();

После успешной авторизации токена, выполняется запрос к базе данных на удаление продукта с указанным ID. Используется SQL-запрос DELETE

Docker и файлы работы

Docker-compose.yml

services:

app:

build:

context: .

ports:

- "3000:3000"

env\_file:

- .env

depends\_on:

- db

db:

image: mysql:latest

container\_name: mysql

restart: always

environment:

MYSQL\_ROOT\_PASSWORD: 19012002

MYSQL\_DATABASE: kp

ports:

- "3306:3306"

volumes:

- ./kp.sql:/docker-entrypoint-initdb.d/kp.sql

phpmyadmin:

image: phpmyadmin/phpmyadmin:latest

container\_name: phpmyadmin

restart: always

environment:

PMA\_HOST: db

MYSQL\_ROOT\_PASSWORD: 19012002

ports:

- "8080:80"

**app** (Frontend/Backend приложение)

build:

context: Указывает на директорию с Dockerfile и другими файлами для сборки контейнера.

ports: Устанавливает соответствие между портом на хосте и портом внутри контейнера. В данном случае, порт 3000 на хосте связан с портом 3000 в контейнере.

env\_file: Загружает переменные окружения из файла .env.

depends\_on: Указывает, что сервис app зависит от сервиса db. Это означает, что db будет запущен перед app.

**db** (MySQL база данных)

image: Используется официальный образ MySQL с последней версией.

container\_name: Задает имя контейнера (mysql).

restart: Гарантирует автоматический перезапуск контейнера в случае его остановки.

environment:

MYSQL\_ROOT\_PASSWORD: Пароль для пользователя root.

MYSQL\_DATABASE: Имя базы данных, которая будет создана при запуске контейнера.

ports: Связывает порт 3306 на хосте с портом 3306 в контейнере, который используется для подключения к MySQL.

volumes:

Монтирует локальный файл базы данных - kp.sql в контейнер, чтобы при запуске база данных инициализировалась этим файлом.

**phpmyadmin** (Веб-интерфейс для управления MySQL)

image: Используется официальный образ phpMyAdmin.

container\_name: Задает имя контейнера (phpmyadmin).

restart: Автоматический перезапуск контейнера при его остановке.

environment:

PMA\_HOST: Указывает, к какому хосту подключаться (в данном случае — db, имя сервиса MySQL).

MYSQL\_ROOT\_PASSWORD: Пароль для подключения к MySQL.

ports: Связывает порт 8080 на хосте с портом 80 в контейнере, что позволяет получить доступ к phpMyAdmin по адресу http://localhost:8080.

Общие пояснения:

volumes: Позволяют сохранять данные и передавать файлы между хостом и контейнером. В данном случае файл kp.sql используется для инициализации базы данных.

depends\_on: Гарантирует, что зависимые сервисы будут запущены перед основным

**Dockerfile**

# Используем официальный образ Node.js

FROM node:20.10

# Устанавливаем рабочую директорию

WORKDIR /app

# Копируем package.json и package-lock.json

COPY package\*.json ./

# Устанавливаем зависимости

RUN npm install

# Копируем весь проект

COPY . .

# Строим Next.js проект

RUN npm run build

# Экспонируем порт приложения

EXPOSE 3000

# Запускаем приложение

CMD ["npm", "start"]

GitHub: <https://github.com/Defolt163/pp_coursework>

Запуск проекта: docker-compose up -d

ПРИЛОЖЕНИЕ – А

**(Изображения)**

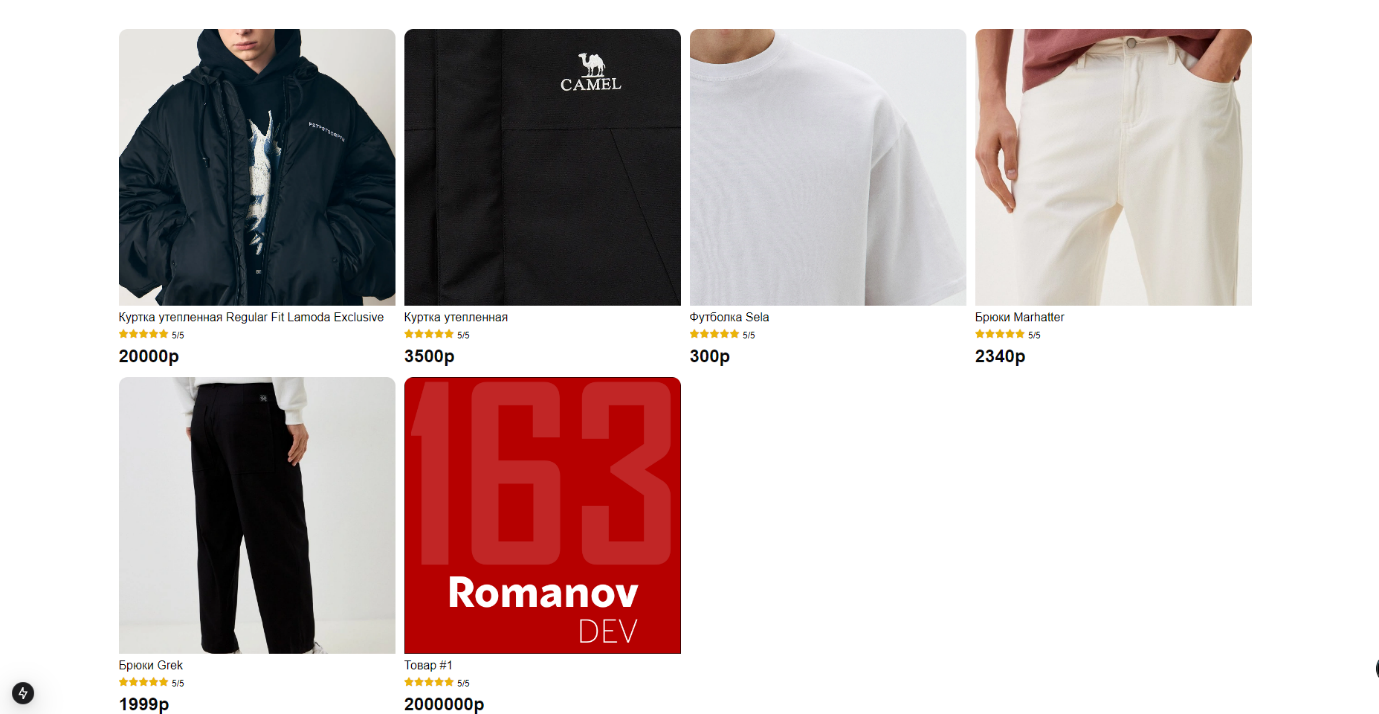


Рисунок 6 Каталог товаров

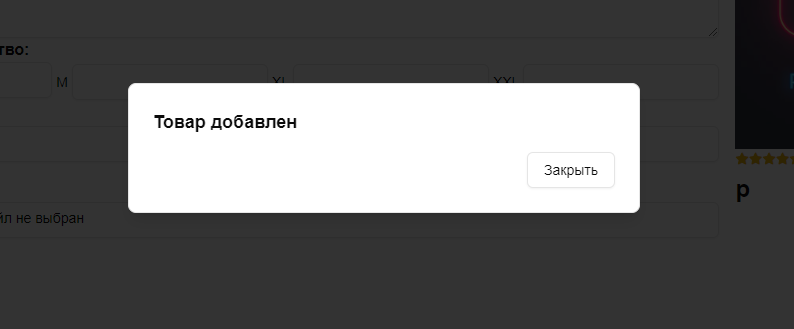


Рисунок 7 Модальное окно об успешном добавлении товара

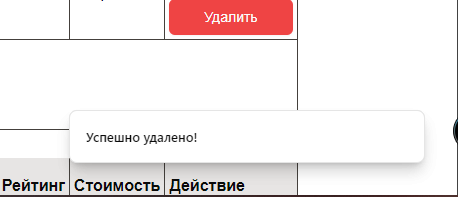


Рисунок 8 Всплывающее окно об успешном запросе

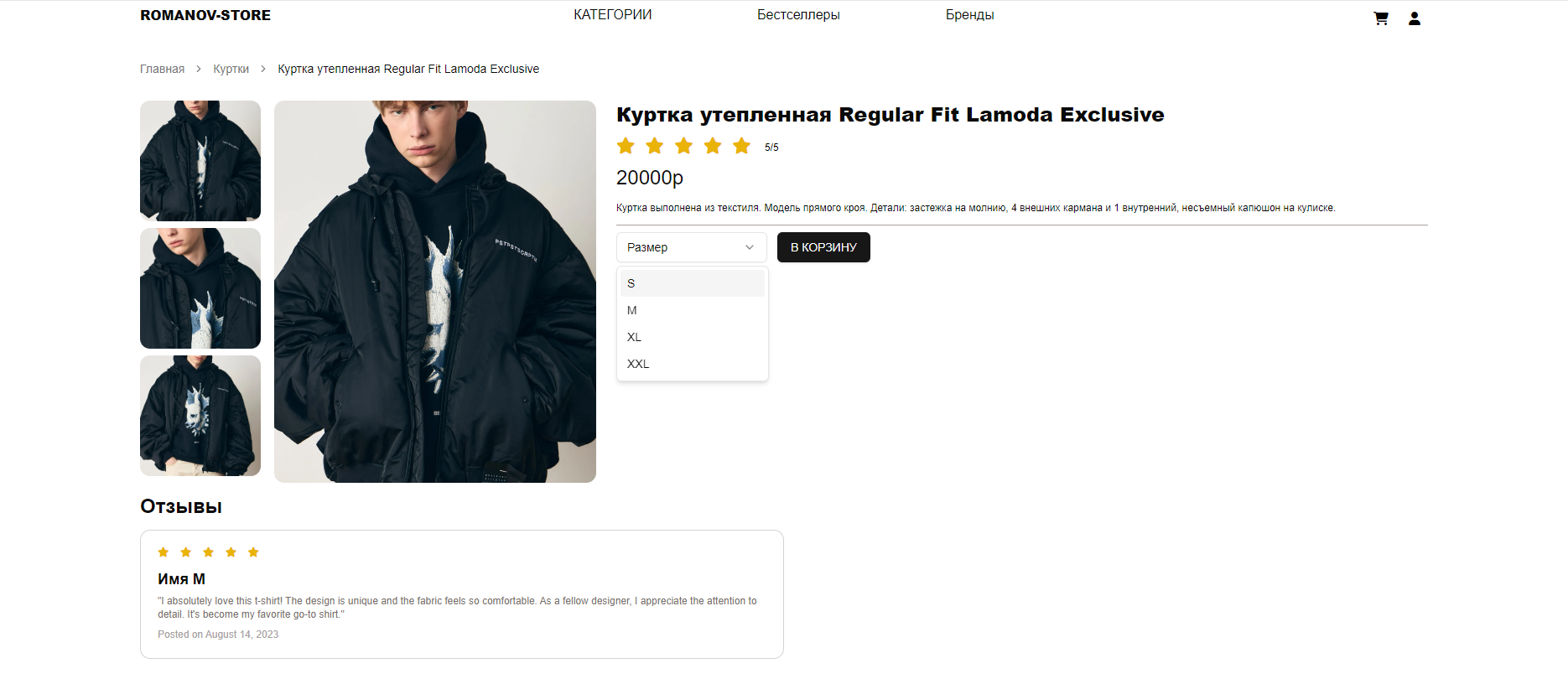


Рисунок 9 Страница товара

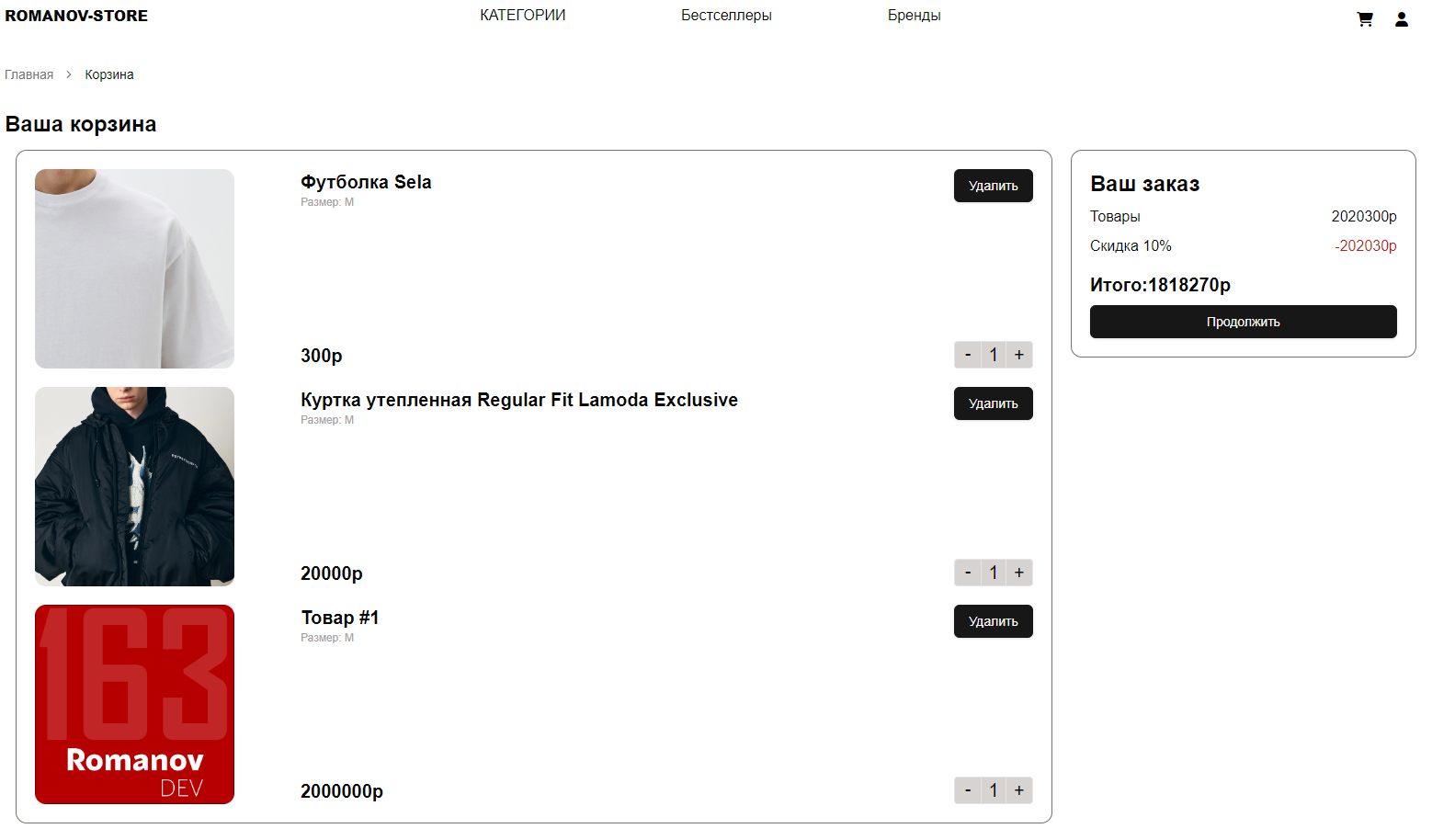


Рисунок 10 Корзина



Рисунок 1 Страница оформления заказа

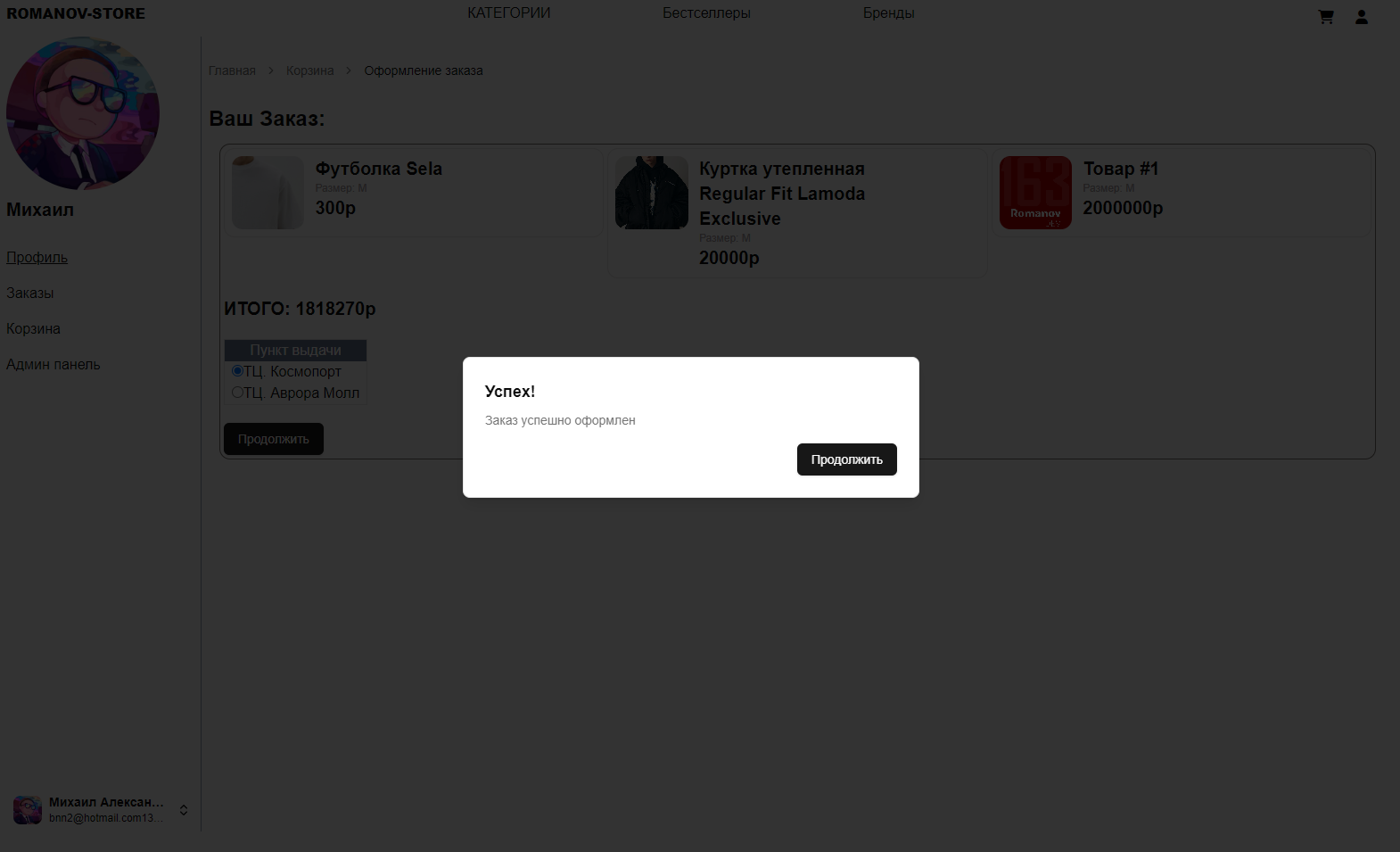


Рисунок 11.1 Успешное оформление заказа

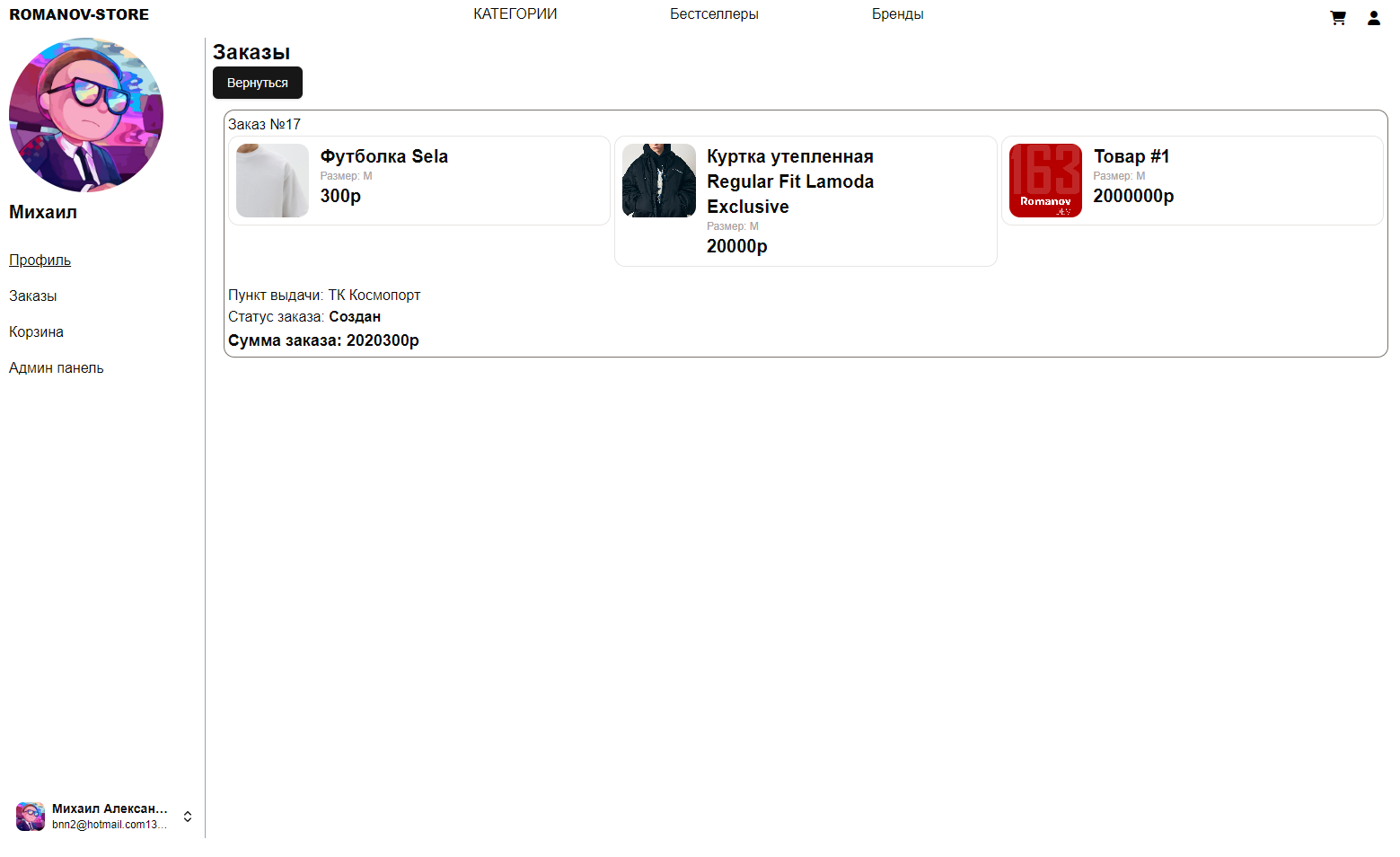


Рисунок 11.2 Страница созданного заказа

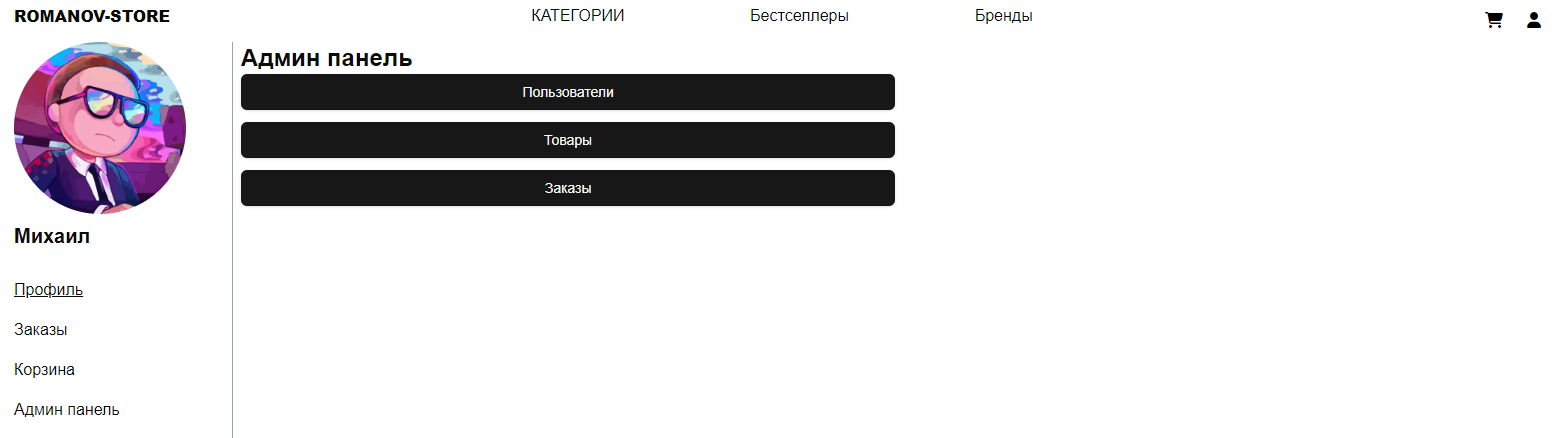


Рисунок 12 Панель администратора

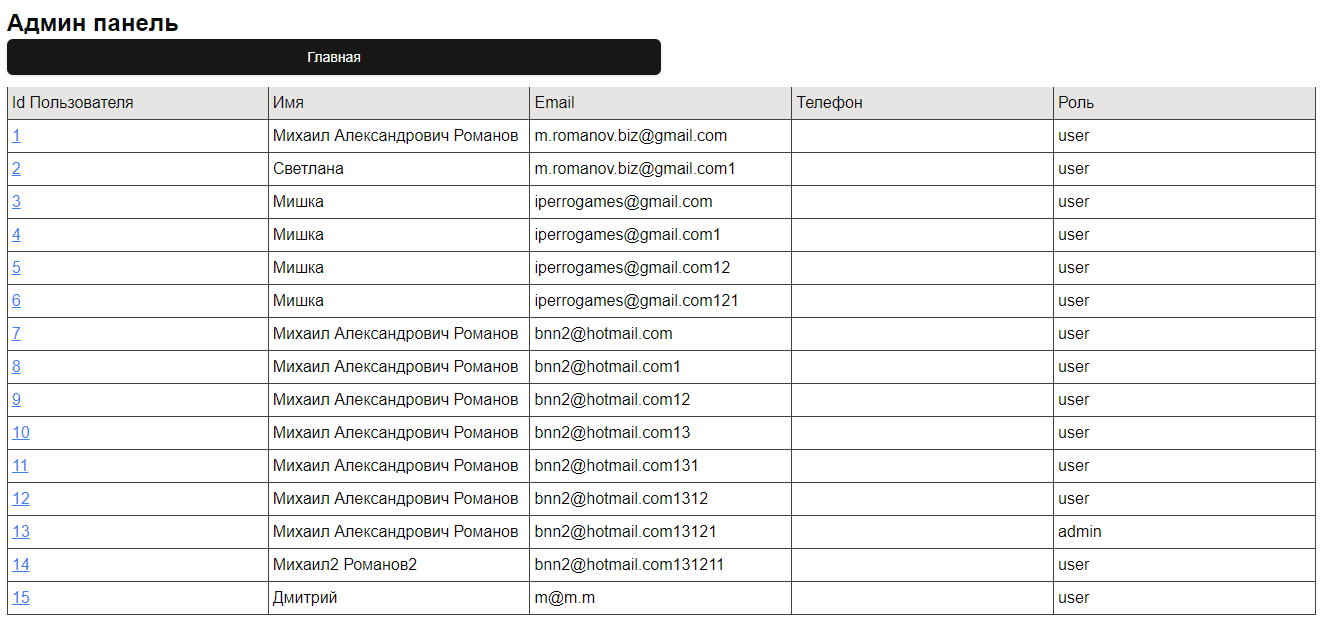


Рисунок 12.1(а) Список пользователей

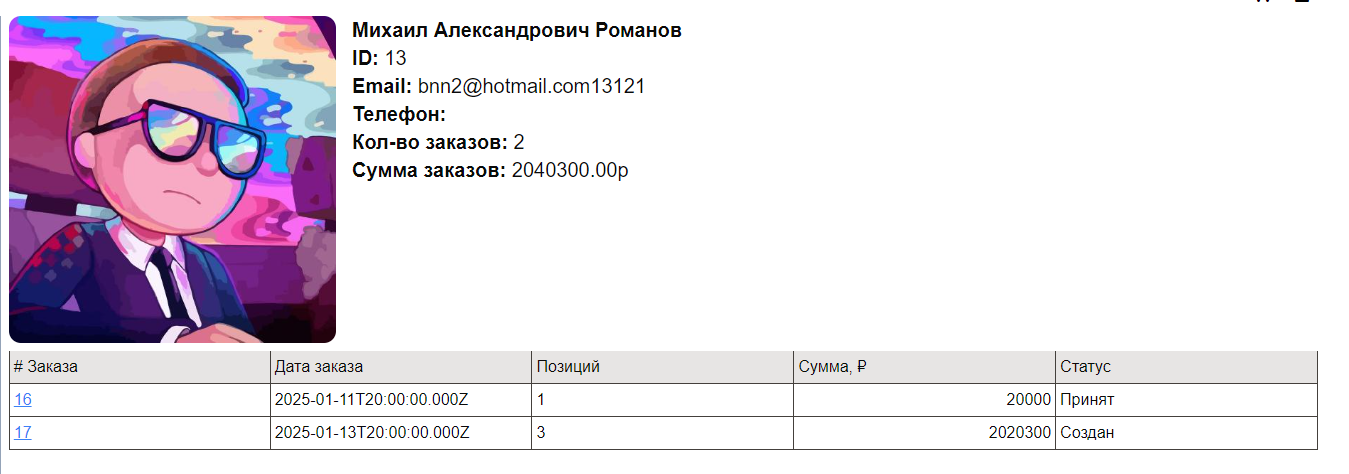


Рисунок 12.1(б) Страница пользователя со всей информацией

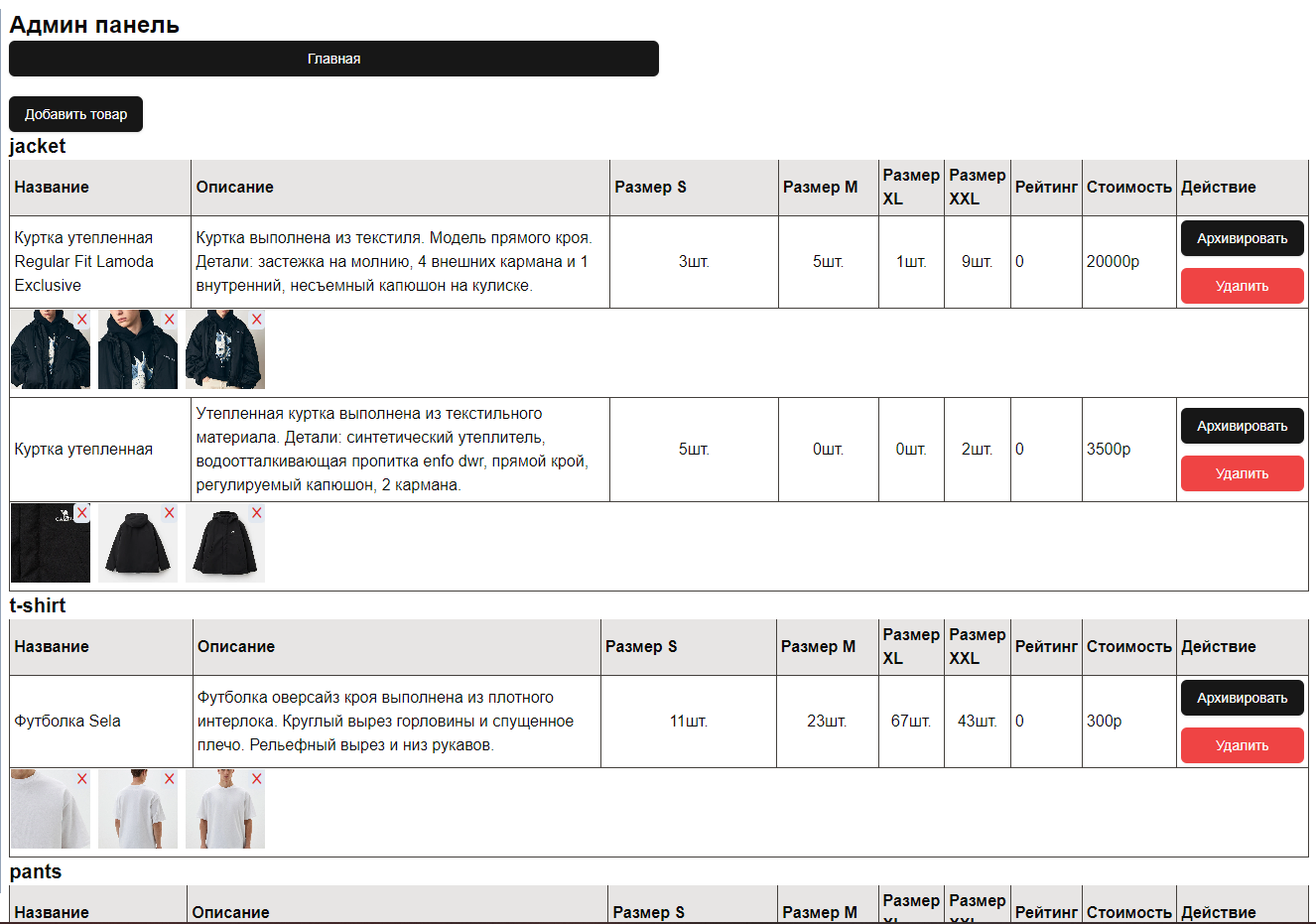


Рисунок 12.2 (а) Список товаров

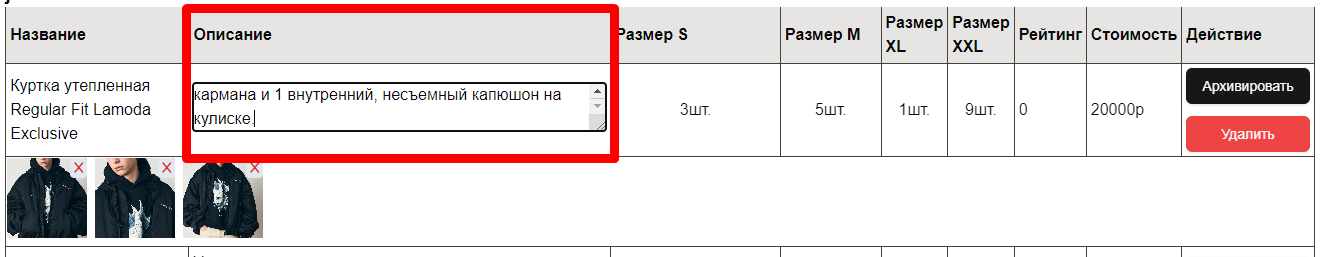


Рисунок 12.2(б) Возможность редактирование информаци

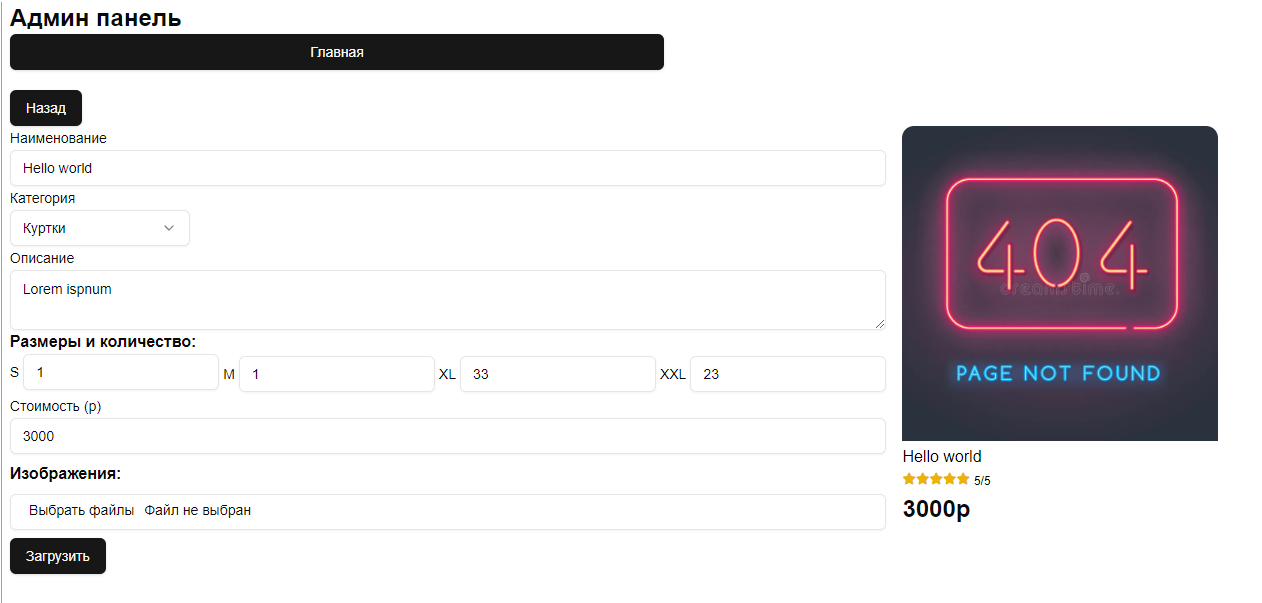


Рисунок 12.2(в) Создание карточки товара

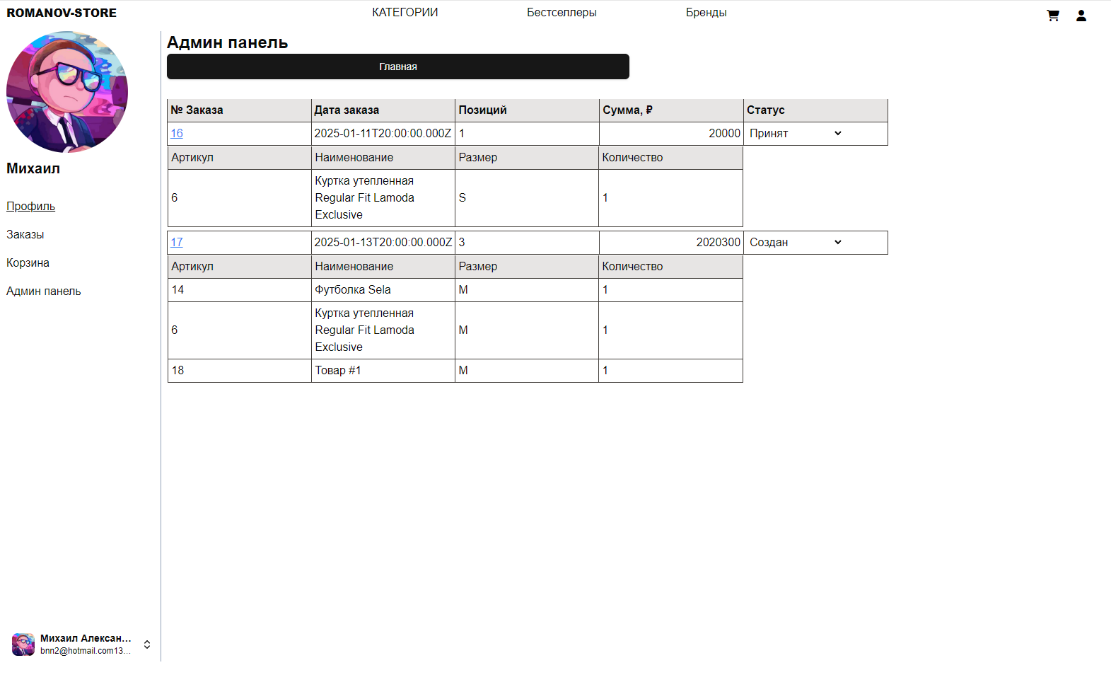


Рисунок 12.3(а) Список всех созданных заказов

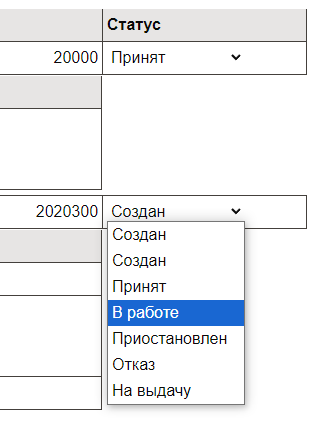


Рисунок 12.3(б) Возможность изменения статуса заказа

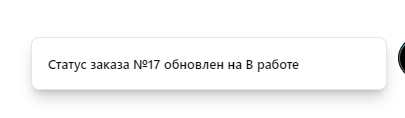


Рисунок 12.3.(в) Уведомление об успешном обновлении статуса

Список используемых источников

1. React: Официальная документация. URL: https://react.dev/docs
2. Next.js: Официальная документация. URL: https://nextjs.org/docs
3. Tailwind CSS: Официальная документация. URL: https://tailwindcss.com/docs
4. ShadCN: Официальная документация. URL: https://shadcn.dev/docs
5. JWT (JSON Web Token): Официальная документация. URL: https://jwt.io/introduction
6. MySQL: Официальная документация. URL: https://dev.mysql.com/doc
7. REST API: Руководство по проектированию. URL: https://www.restapitutorial.com
8. Docker: Официальная документация. URL: https://docs.docker.com