**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

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:

1. React: Популярная библиотека для разработки пользовательских интерфейсов. React обеспечивает модульный подход к созданию компонентов, что идеально сочетается с принципами микросервисной архитектуры.
2. 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.
3. Tailwind CSS: Утилитарный CSS-фреймворк, обеспечивающий гибкость и высокую скорость стилизации. Tailwind позволяет легко создавать адаптивные интерфейсы, используя классы напрямую в разметке.
4. 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.
5. JWT-аутентификация: Технология JSON Web Token используется для авторизации пользователей. Этот подход обеспечивает безопасность и масштабируемость, позволяя эффективно управлять доступом к различным частям системы.
6. MySQL: Реляционная база данных, выбранная для хранения и управления данными. MySQL обеспечивает высокую производительность, отказоустойчивость и гибкость при работе с большими объемами данных.
7. REST API: Подход к взаимодействию между микросервисами и фронтендом. REST API позволяет стандартизировать обмен данными, упрощая интеграцию и масштабирование системы.
8. 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:

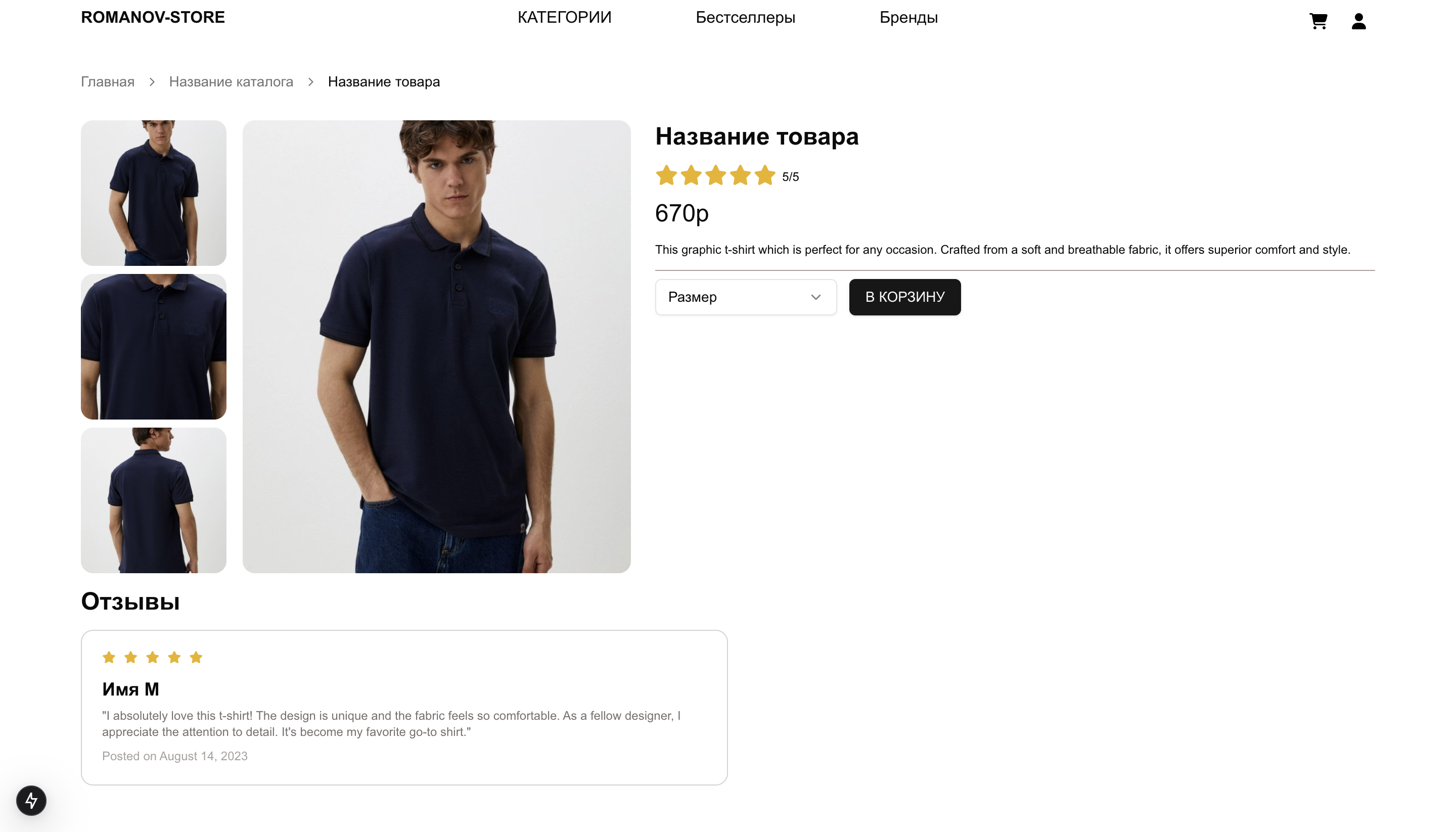


Рис 1 Страница товара

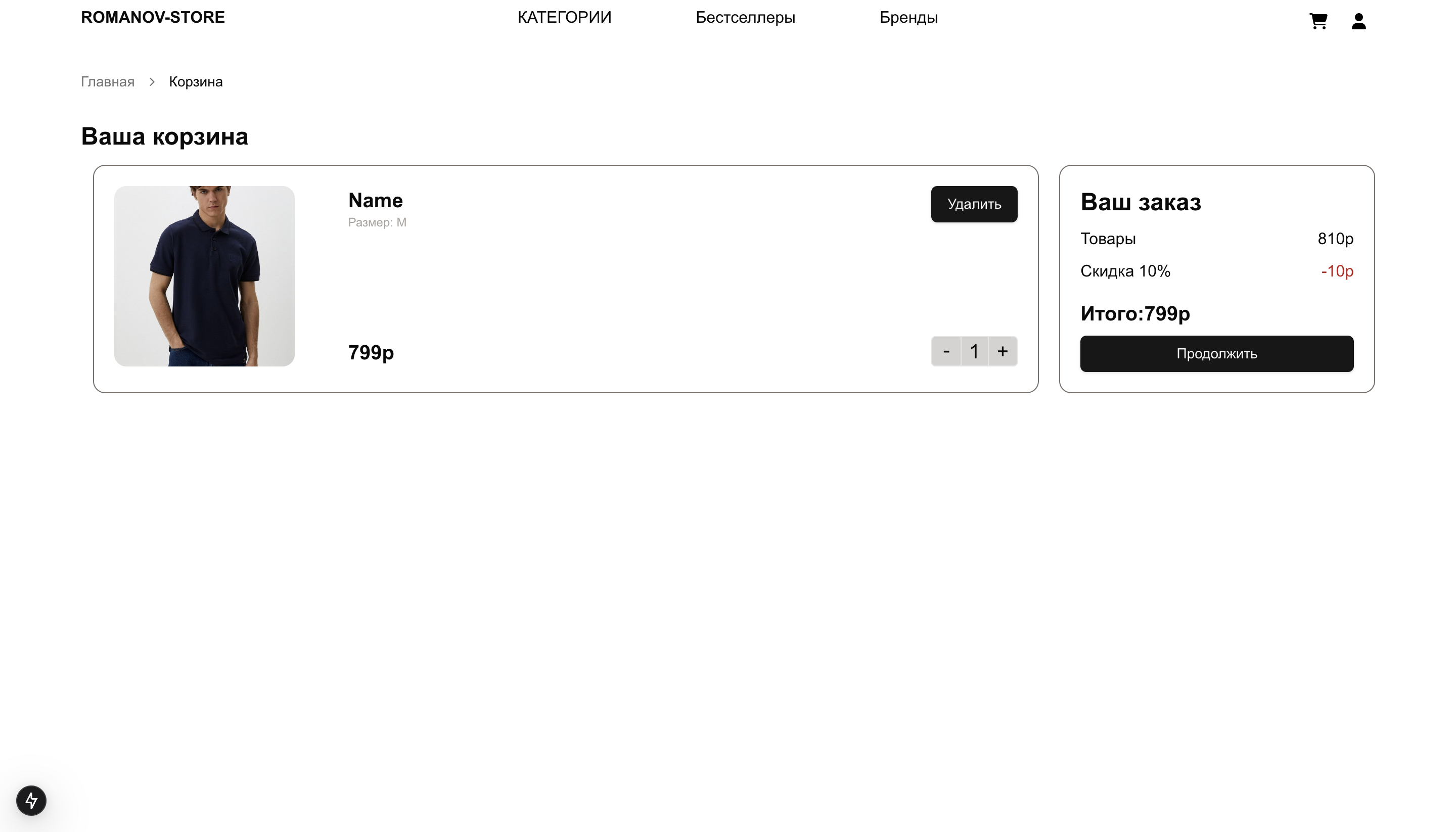


Рис 2 Корзина

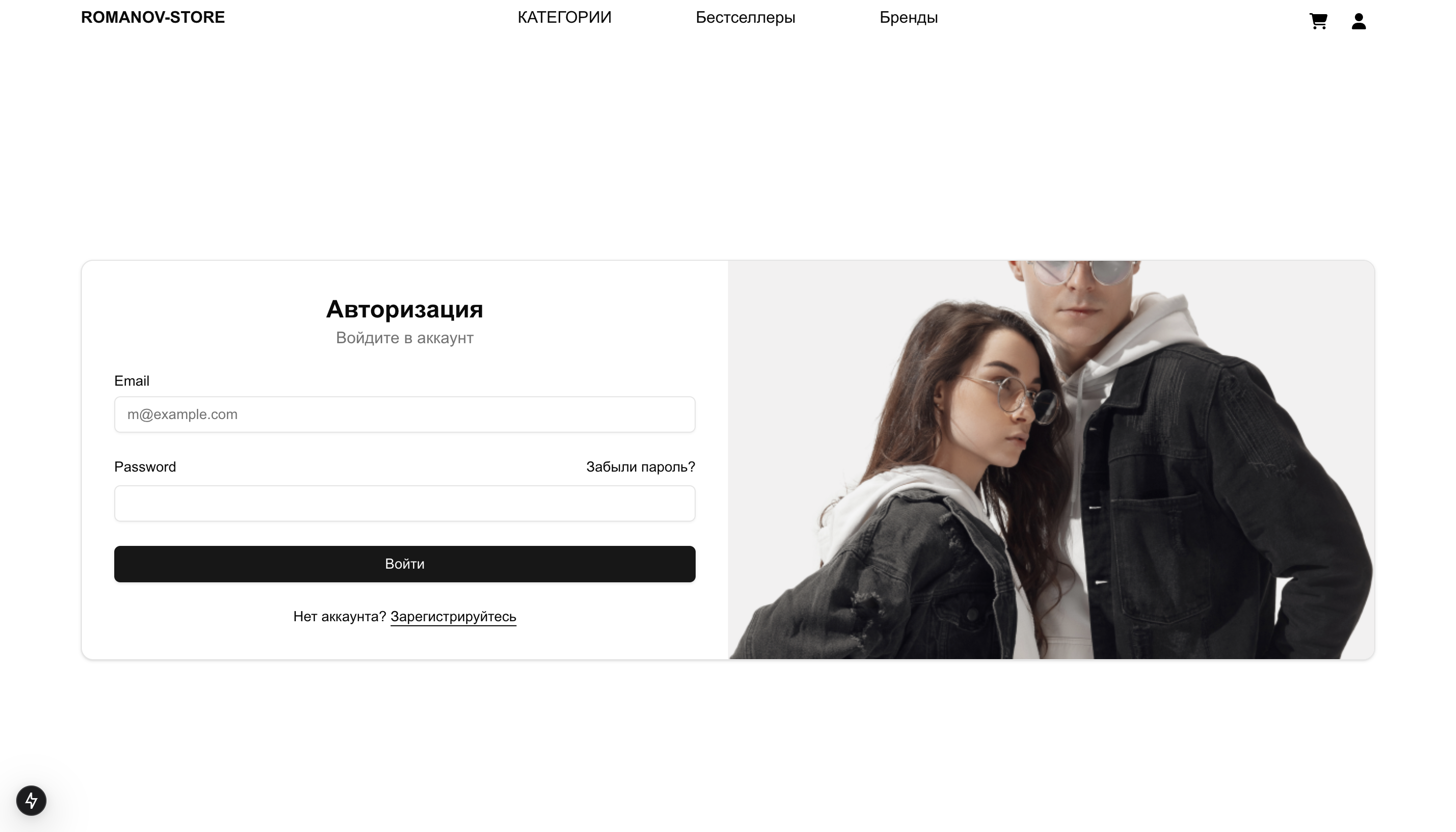


Рис 3 Авторизация Рис 4 Регистрация

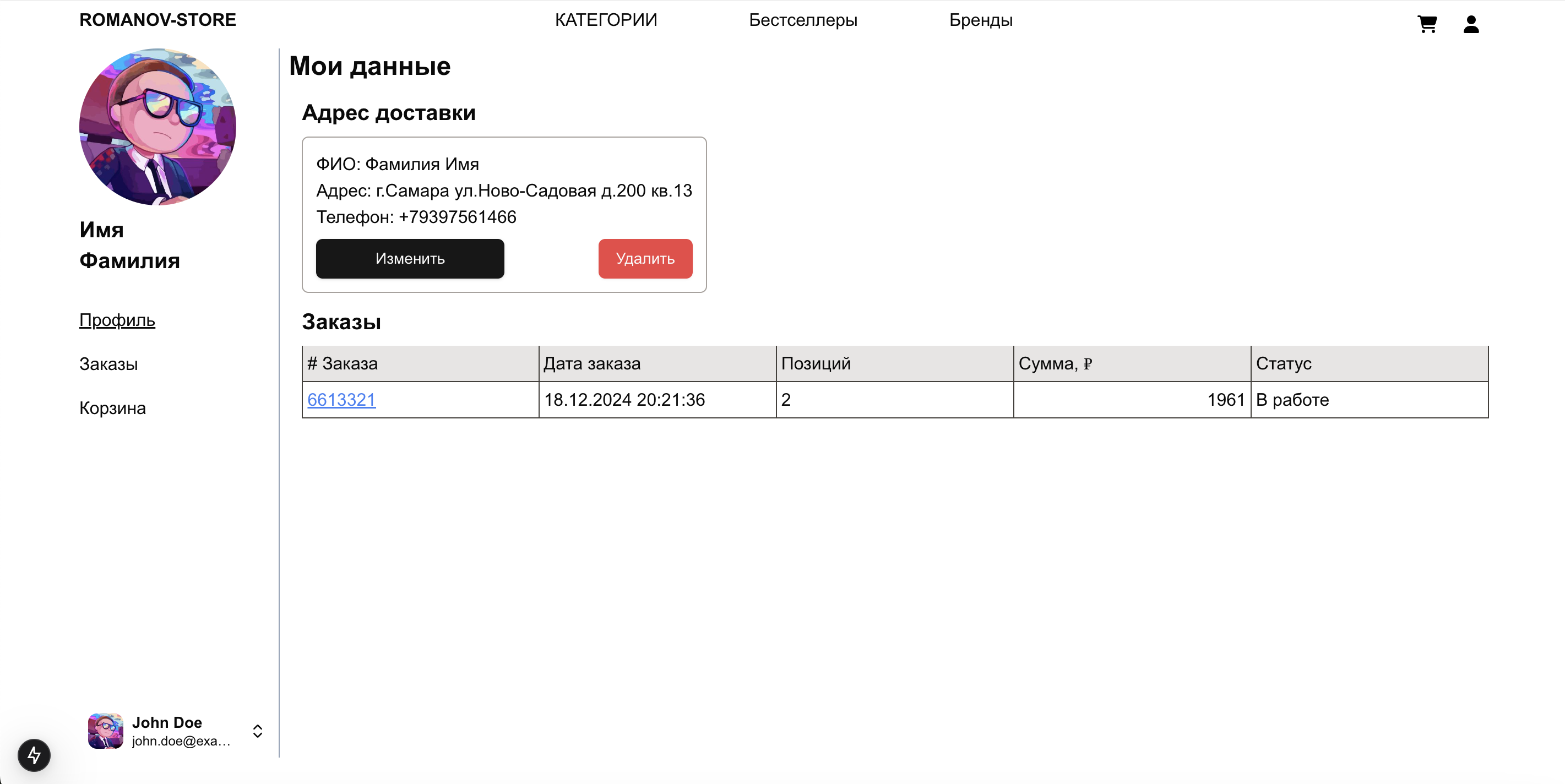


Рис 5 Личный кабинет

**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 входа при попытке доступа к защищённым маршрутам (например, /dashboard).

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.