МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждения образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ

ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет информационных технологий

Кафедра информационный систем и технологий

Специальность 1-40 05 01-03 Информационные системы и технологии

Направление специальности 1-40 05 01-03 Информационные системы и технологии

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**КУРСОВОГО ПРОЕКТ:**

по дисциплине «Современные технологии разрработки web-приложений»

Тема: Web-приложение для аренды жилья

Исполнитель

студент (ка) 4 курса группы 2 Синяк Павел Сергеевич

(Ф.И.О.)

Руководитель работы ассистент Харланович А.В.

(учен. степень, звание, должность, подпись, Ф.И.О.)

Курсовой проект защищен с оценкой

Минск 2024

Содержание

[Введение 5](#_Toc185239098)

[1 Обзор аналогов и постановка задачи 6](#_Toc185239099)

[1.1 Постановка задачи 6](#_Toc185239100)

[1.2 Аналитический обзор аналогов 6](#_Toc185239101)

[1.2.1 Booking 7](#_Toc185239102)

[1.2.2 Cian 8](#_Toc185239103)

[1.2.3 Sutochno 9](#_Toc185239104)

[1.3 Выводы по разделу 10](#_Toc185239105)

[2 Проектирование web-приложения 12](#_Toc185239106)

[2.1 Разработка функциональных требований и вариантов использования 12](#_Toc185239107)

[2.2 Описание средств разработки 13](#_Toc185239108)

[2.2.1 Visual Studio Code 13](#_Toc185239109)

[2.2.2 Платформа Node.js 13](#_Toc185239110)

[2.2.3 Язык программирования TypeScript 13](#_Toc185239111)

[2.2.4 Библиотека React 14](#_Toc185239112)

[2.2.5 PostgreSQL 14](#_Toc185239113)

[2.3 Диаграммы UML, взаимосвязь всех компонентов. 15](#_Toc185239114)

[2.4 Выводы по разделу 18](#_Toc185239115)

[3 Разработка web-приложения 19](#_Toc185239116)

[3.1 Общая структура 19](#_Toc185239117)

[3.2 Разработка серверной части web-приложения 19](#_Toc185239118)

[3.2.1 Регистрация 23](#_Toc185239119)

[3.2.2 Авторизация 24](#_Toc185239120)

[3.2.3 Управление жильём в системе 25](#_Toc185239121)

[3.2.4 Бронирование жилья 29](#_Toc185239122)

[3.2.5 Оставление отзывов с оцениванием после сделки 30](#_Toc185239123)

[3.3 Разработка клиентской части web-приложения 31](#_Toc185239124)

[3.4 Выводы по разделу 34](#_Toc185239125)

[4 Тестирование web-приложения 35](#_Toc185239126)

[4.1 Тестирование авторизации и регистрации 35](#_Toc185239127)

[4.2 Тестирование добавление жилья 36](#_Toc185239128)

[4.3 Тестирование поиска вакансий 37](#_Toc185239129)

[4.4 Тестирование изменения настроек пользователя 39](#_Toc185239130)

[4.5 Выводы по разделу 40](#_Toc185239131)

[5 Руководство пользователя 41](#_Toc185239132)

[5.1 Руководство пользователя 41](#_Toc185239133)

[5.1.1 Пользователь 41](#_Toc185239134)

[5.1.2 Владелец 45](#_Toc185239135)

[5.1.3 Администратор 47](#_Toc185239136)

[5.2 Установка приложения 49](#_Toc185239137)

[5.3 Выводы по разделу 50](#_Toc185239138)

[Заключение 51](#_Toc185239139)

[Список используемы источников 52](#_Toc185239140)

[Приложение А 53](#_Toc185239141)

[Приложение Б 81](#_Toc185239142)

# Введение

В современном мире, где повышенная мобильность и требования к качеству жизни становятся нормой, аренда жилья превращается в сложный процесс для обеих сторон — арендаторов и арендодателей. Постоянное увеличение количества информации о доступной недвижимости, условиях съёма и предпочтениях арендаторов подчёркивает необходимость эффективных инструментов для упрощения этого процесса. Веб-приложения, разработанные для таких целей, играют ключевую роль, предлагая удобные механизмы поиска, взаимодействия и подачи заявок.

В рамках данной курсовой работы разработано веб-приложение, ориентированное на аренду жилья. Приложение включает функционал поиска, фильтрации, визуализации объектов недвижимости и коммуникации между участниками сделки. Основная задача проекта — создание удобного и простого в использовании интерфейса, который поможет пользователям находить подходящие предложения, сравнивать их и легко взаимодействовать с арендодателями.

Для реализации приложения используются современные технологии: React для разработки пользовательского интерфейса, Node.js для серверной части и PostgreSQL для управления базой данных. Такой подход обеспечивает надёжность, высокую производительность и масштабируемость решения. Принципы модульности и асинхронности, лежащие в основе архитектуры, гарантируют гибкость системы, которая может адаптироваться к изменениям и запросам рынка.

Проект охватывает полный цикл разработки: от анализа требований и проектирования структуры данных до внедрения ключевых функций и тестирования. Итоговая система станет полезным инструментом как для арендаторов, ищущих жильё, соответствующее их запросам, так и для арендодателей, стремящихся найти надёжных жильцов.

Таким образом, данное приложение представляет собой комплексное решение, призванное облегчить процесс аренды жилья, минимизирую усилия пользователей и повышая их удовлетворённость результатом.

# 1 Обзор аналогов и постановка задачи

## 1.1 Постановка задачи

Важным этапом разработки любого веб-приложения является анализ существующих аналогов на рынке. Это позволяет выявить их сильные и слабые стороны, понять основные потребности пользователей и определить направления для внедрения уникальных решений.

Данное веб-приложение, ориентированное на аренду жилья, разработано с использованием современных технологий — React с TypeScript для фронтенда, Node.js для серверной части и PostgreSQL для работы с базой данных. Его цель — автоматизировать и упростить процессы поиска, бронирования и управления жильём. Приложение должно стать удобным инструментом для всех участников: администраторов, владельцев недвижимости и пользователей.

На рынке уже существует множество платформ для аренды жилья. Однако, многие из них имеют недостатки: ограниченная кастомизация поиска, сложность управления объявлениями для владельцев или недостаточное взаимодействие между арендаторами и арендодателями. Это открывает возможность для создания продукта, который предложит гибкий функционал, простой интерфейс и эффективное взаимодействие между пользователями.

Цель проекта — разработка современного веб-приложения, предоставляющего удобные инструменты для поиска, бронирования и управления недвижимостью.

Должны быть выполнены следующие требования:

* обеспечивать возможность регистрации и авторизации;
* поддерживать роли: администратор, владелец, пользователь;
* поиск, фильтрация и сортировку вакансий;
* возможность управлять критериями и категориями жилья;
* возможность управлять пользователями, владельцами и их доступом;
* возможность управлять жильём в системе;
* возможность управлять статусами бронирования;
* поддержка чата между пользователем и владельцем;
* возможность просматривать информацию о жилье;
* возможность бронировать интересующее его жильё;
* возможность оставлять отзывы с оцениванием после сделки.

Проект включает несколько ключевых этапов: проектирование архитектуры приложения, разработка интерфейсов и серверной логики, создание базы данных, тестирование и развертывание готового продукта.

## 1.2 Аналитический обзор аналогов

Были проанализированы цели и задачи, поставленные в данном курсовом проекте, а также рассмотрены аналогичные примеры их решений. На основании анализа всех достоинств и недостатков данных альтернативных решений были сформулированы требования к данному программному средству.

### 1.2.1 Booking

**Booking** [1] — это одна из ведущих платформ для поиска и бронирования жилья, которая предоставляет пользователям возможность находить подходящие варианты аренды, используя удобные фильтры, просматривать подробные описания объектов и бронировать жильё в режиме онлайн. Пример интерфейса данного сервиса представлен на рисунке 1.1.

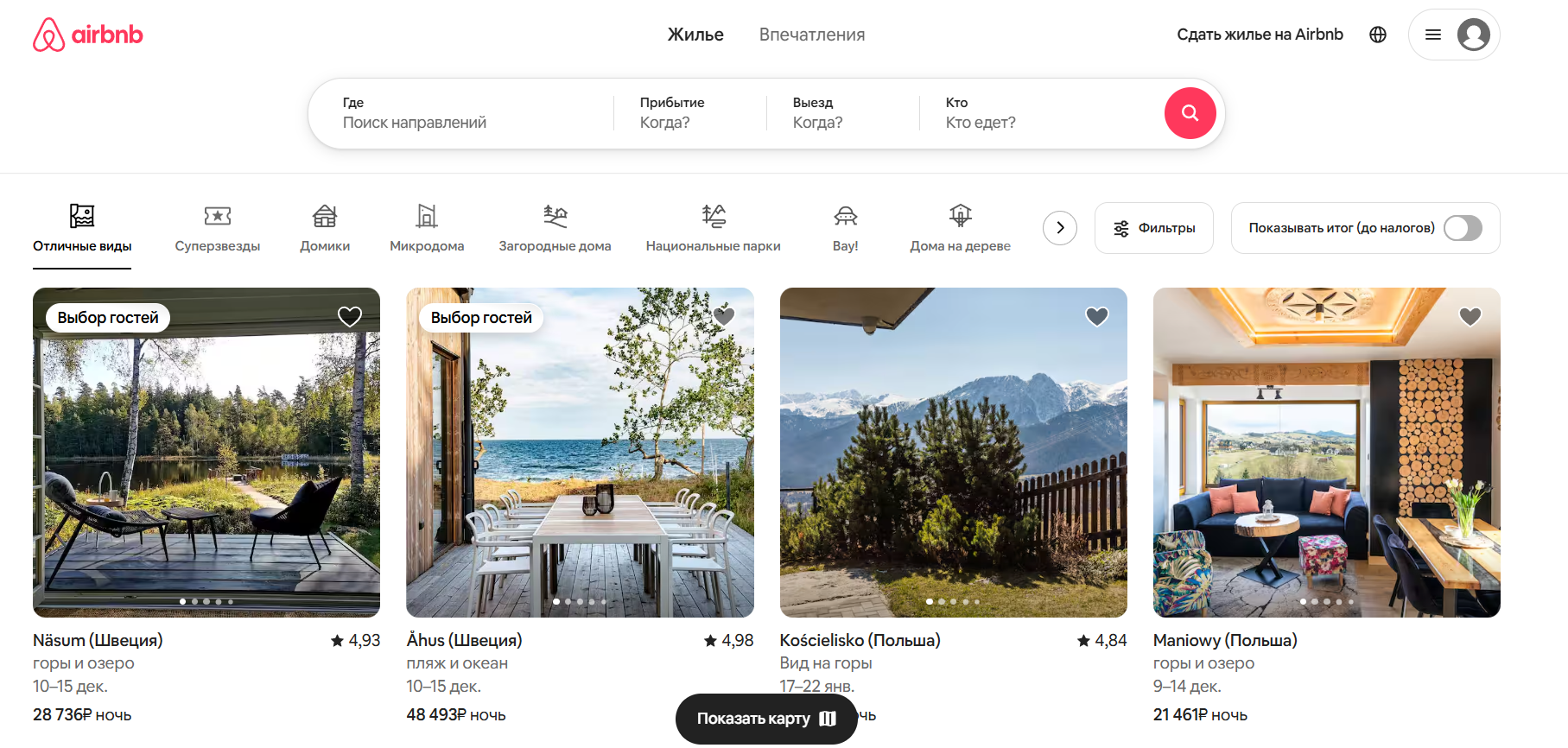


Рисунок 1.1 – Интернет-ресурс **Booking**

Одной из ключевых функций Booking является поиск жилья. Пользователи могут искать объекты недвижимости, применяя фильтры по местоположению, дате, цене, количеству гостей, рейтингу, удобствам и другим параметрам. Сервис также предоставляет персонализированные рекомендации на основе истории поиска и ранее забронированных объектов.

Для удобства пользователей Booking предлагает возможность создавать учётные записи, где можно управлять бронированиями, сохранять интересующие объекты в "Избранное" и оставлять отзывы после завершения проживания. Пользователи могут ознакомиться с отзывами других гостей, что помогает сделать более информированный выбор.

Для владельцев недвижимости Booking предоставляет платформу для регистрации и управления своими объектами. Они могут добавлять описания, фотографии, устанавливать цены, задавать правила отмены бронирования, а также управлять календарём доступности. Сервис предоставляет статистику по каждому объекту, включая количество просмотров, бронирований и средний рейтинг, что помогает владельцам анализировать спрос и оптимизировать свои предложения.

С технической стороны Booking использует сложные алгоритмы поиска и фильтрации, а также высокопроизводительные базы данных для обработки миллионов запросов пользователей ежедневно. Оптимизированная архитектура обеспечивает быструю загрузку страниц и стабильность работы системы даже при высоких нагрузках.

Эта платформа выделяется своей глобальной доступностью, поддержкой множества языков и валют, а также адаптивным интерфейсом, который обеспечивает удобство использования как на компьютерах, так и на мобильных устройствах. Благодаря широкому функционалу и пользовательскому комфорту, Booking остаётся одним из лидеров рынка онлайн-бронирования жилья.

### 1.2.2 ****Cian****

**Cian** [2] — это одна из крупнейших платформ для поиска и аренды недвижимости, которая предоставляет пользователям возможность находить подходящие варианты жилья, используя гибкие фильтры, просматривать подробные описания объектов и напрямую связываться с арендодателями. Пример интерфейса данного сервиса представлен на рисунке 1.2.

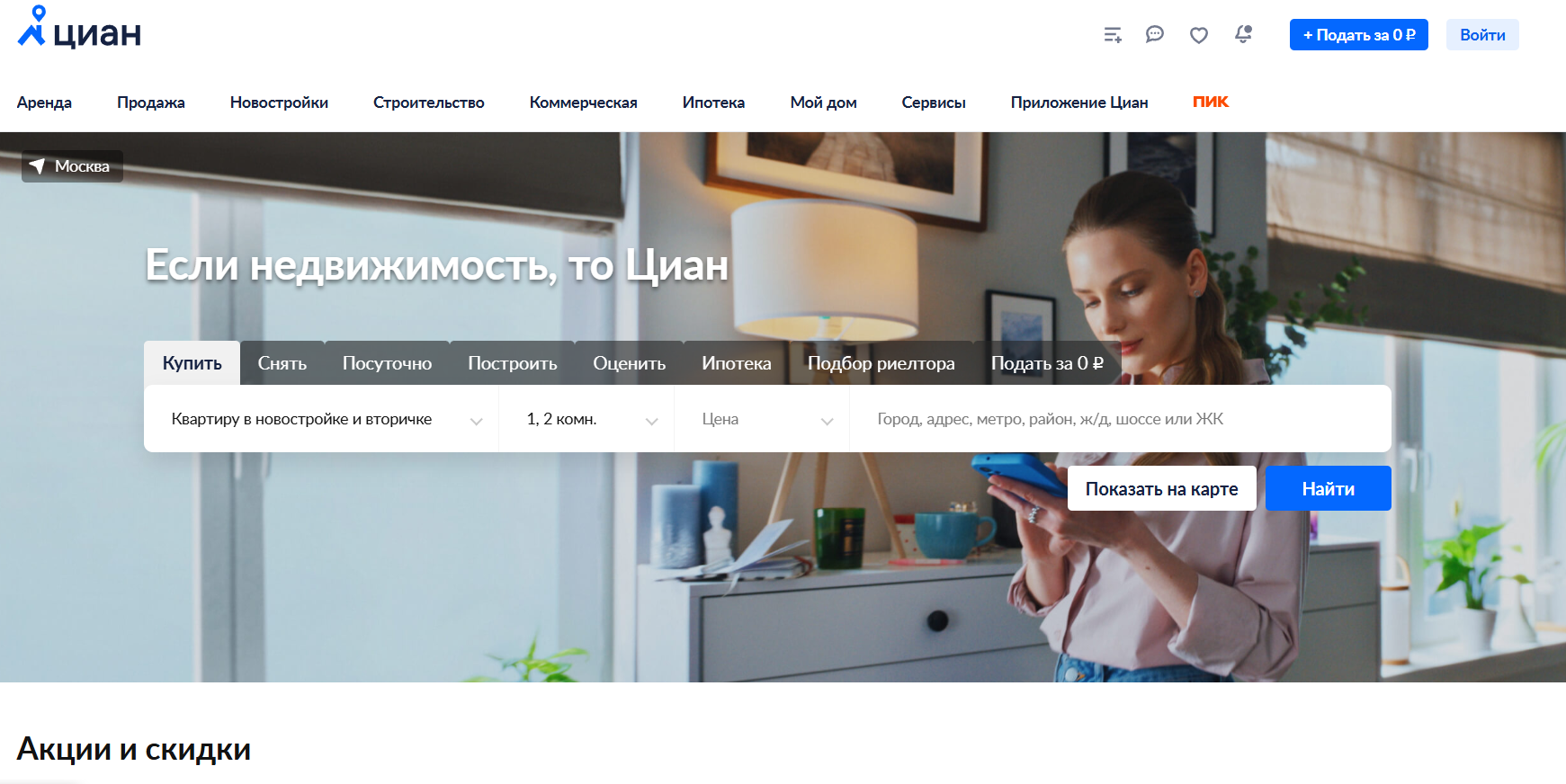


Рисунок 1.2 – Интернет-ресурс **Cian**

Ключевой функцией Cian является поиск недвижимости. Пользователи могут находить объекты, применяя разнообразные фильтры: по цене, району, типу жилья, площади, количеству комнат, сроку аренды и другим параметрам. Также платформа предоставляет удобную карту, где объекты отображаются с указанием точного местоположения, что упрощает выбор подходящего варианта.

Для удобства пользователей Cian предлагает возможность создавать личные кабинеты, где можно сохранять интересующие объявления в "Избранное", управлять своими объявлениями (для арендодателей), отслеживать статус откликов и получать уведомления о новых предложениях.

Для арендодателей и продавцов недвижимости Cian предоставляет функционал для размещения и управления объявлениями. Владельцы могут добавлять фотографии, описания, указывать стоимость и условия аренды, а также редактировать или удалять объекты. Платформа предоставляет статистику для каждого объявления, включая количество просмотров и откликов, что помогает анализировать спрос и корректировать предложения.

С технической стороны Cian использует современные технологии для работы с большими объёмами данных, что обеспечивает быструю загрузку страниц, точный поиск и стабильную работу сервиса. Платформа поддерживает интеграцию с внешними системами, такими как геолокационные сервисы и платёжные системы, что делает её удобной как для арендаторов, так и для арендодателей.

Удобный и интуитивно понятный интерфейс Cian адаптирован для работы как на компьютерах, так и на мобильных устройствах, благодаря чему пользователи могут быстро находить и просматривать предложения в любое время. Благодаря обширной базе данных объектов, гибкости поиска и дополнительным функциям, Cian остаётся одним из лидеров среди платформ для аренды и покупки недвижимости.

### 1.2.3 Sutochno

Sutochno [3] — это онлайн-платформа, специализирующаяся на краткосрочной аренде жилья. Она предоставляет пользователям удобные инструменты для поиска и бронирования квартир, домов, комнат и других объектов недвижимости по всей стране. Пример интерфейса данного сервиса представлен на рисунке 1.3.

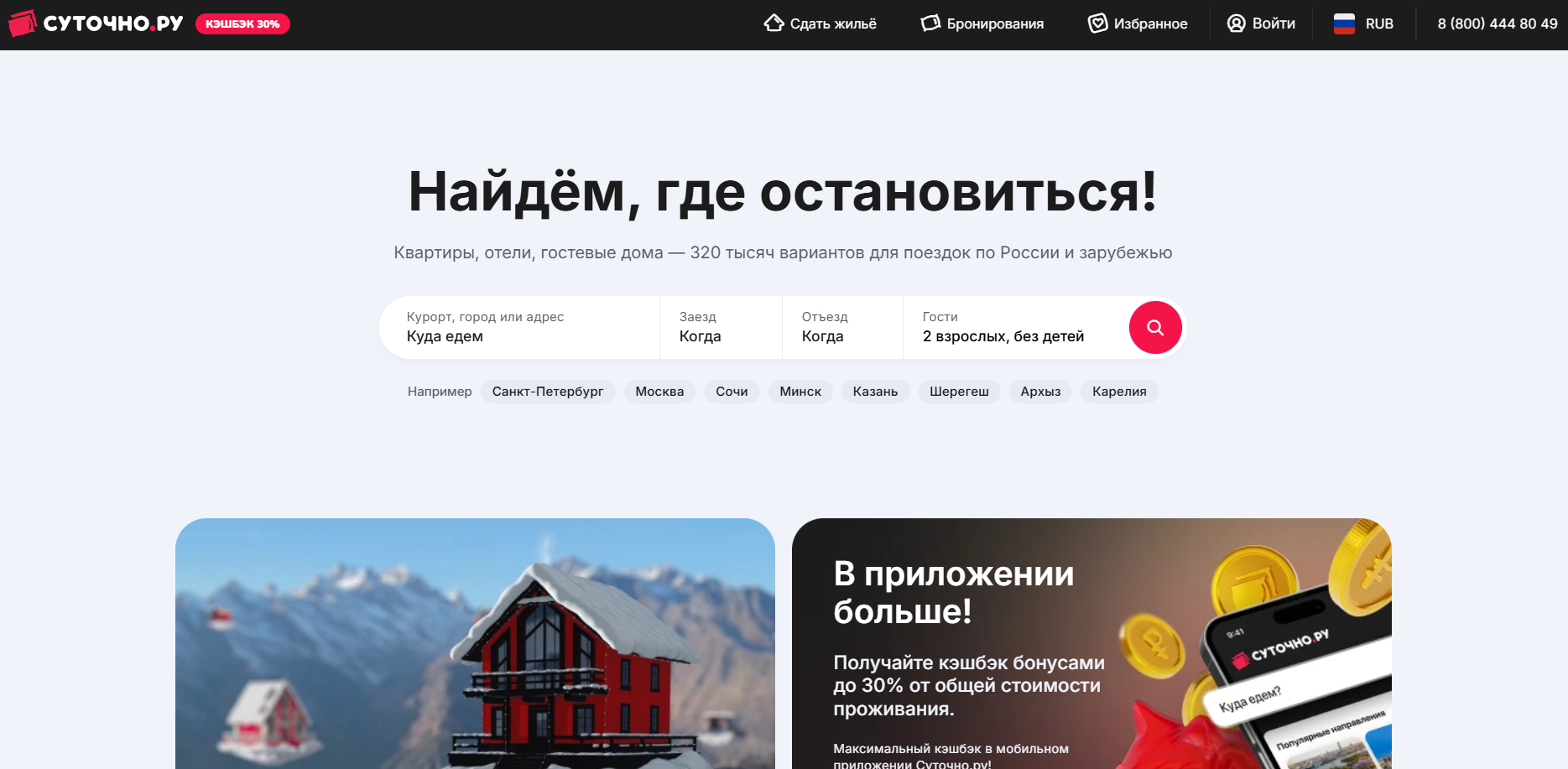


Рисунок 1.3 – Интернет-ресурс Sutochno

Основной функционал Sutochno сосредоточен на быстром поиске и аренде жилья на сутки или более короткие сроки. Пользователи могут искать объекты, применяя фильтры по цене, дате, количеству комнат, типу жилья, местоположению и другим параметрам. Система также предлагает карту для визуального поиска, на которой объекты отмечены с указанием стоимости аренды.

Платформа позволяет пользователям регистрировать личные кабинеты, где можно управлять бронированиями, сохранять интересующие объекты в "Избранное" и оставлять отзывы после завершения проживания. Пользователи могут общаться с арендодателями через встроенный чат, уточняя детали проживания.

Для арендодателей Sutochno предоставляет возможность размещать и редактировать объявления. Владельцы могут добавлять описания, фотографии, указывать правила проживания, условия отмены бронирования и доступные даты. Платформа поддерживает аналитические инструменты, позволяющие владельцам отслеживать популярность их объектов: количество просмотров, бронирований и отзывы.

С технической стороны Sutochno использует современные инструменты для обеспечения быстродействия и стабильности работы платформы. Высокопроизводительные базы данных и алгоритмы фильтрации позволяют обрабатывать большие объёмы информации, обеспечивая пользователям точные результаты поиска.

Sutochno выделяется своей ориентацией на рынок, предлагая широкий выбор объектов в крупных городах и популярных туристических направлениях. Благодаря функционалу, направленному на удобство и прозрачность для всех участников, платформа остаётся востребованным инструментом как для арендаторов, так и для арендодателей, обеспечивая комфортное взаимодействие и безопасные сделки.

## 1.3 Выводы по разделу

В данном разделе был проведён обзор аналогичных веб-приложений, таких как Booking, Cian и Sutochno, что позволило выявить ключевые функции, которые должны быть реализованы в нашем приложении. Эти платформы предоставляют пользователям широкий спектр возможностей: от регистрации и авторизации до поиска и бронирования жилья с использованием удобных фильтров, а также управления объектами недвижимости для владельцев.

Особое внимание уделено функционалу для различных ролей пользователей. Для владельцев объектов аренды важными элементами являются инструменты для добавления, редактирования и удаления объектов, управления бронированиями, а также ведения переписки с арендаторами. Пользователям необходимы функции просмотра и поиска жилья, бронирования, отслеживания статуса бронирований, а также возможности оставлять отзывы.

Также обсуждены требования к удобному интерфейсу, который должен быть интуитивно понятным и обеспечивать эффективное взаимодействие между владельцами недвижимости, арендаторами и администраторами системы. Для визуализации основных процессов взаимодействия и проектирования системы была разработана UML-диаграмма, отражающая ключевые сущности и связи между ними.

Результаты обзора аналогичных приложений позволили определить направления для улучшения и выделить уникальные особенности, которые будут учтены при разработке функционала нашего веб-приложения.

# 2 Проектирование web-приложения

## 2.1 Разработка функциональных требований и вариантов использования

Приложение должно предоставлять функционал для регистрации и авторизации, позволяя пользователям идентифицировать себя и получать доступ к возможностям в зависимости от их роли: администратор, владелец недвижимости или пользователь.

Администратор получит возможность управлять системой, включая редактирование категорий и критериев для поиска жилья, управление пользователями и их доступом, а также просмотр и анализ статистики по всем объектам и бронированиям. Это позволит обеспечить порядок и прозрачность работы приложения.

Владелец недвижимости сможет добавлять, редактировать и удалять свои объекты аренды, управлять статусами бронирований, а также просматривать подробную статистику по своим объектам, включая количество просмотров, бронирований и отзывы. Для удобства взаимодействия с пользователями владельцу будет доступен встроенный чат, через который можно уточнять детали бронирования и отвечать на вопросы.

Пользователь сможет искать жильё, используя удобные фильтры и сортировку, бронировать интересующие варианты, просматривать статус своих бронирований и историю операций. Для улучшения качества сервиса пользователь сможет оставлять отзывы и оценки после завершения аренды, а также управлять своими личными данными через настройки профиля.

На рисунке 2.1 представлена диаграмма вариантов использования.

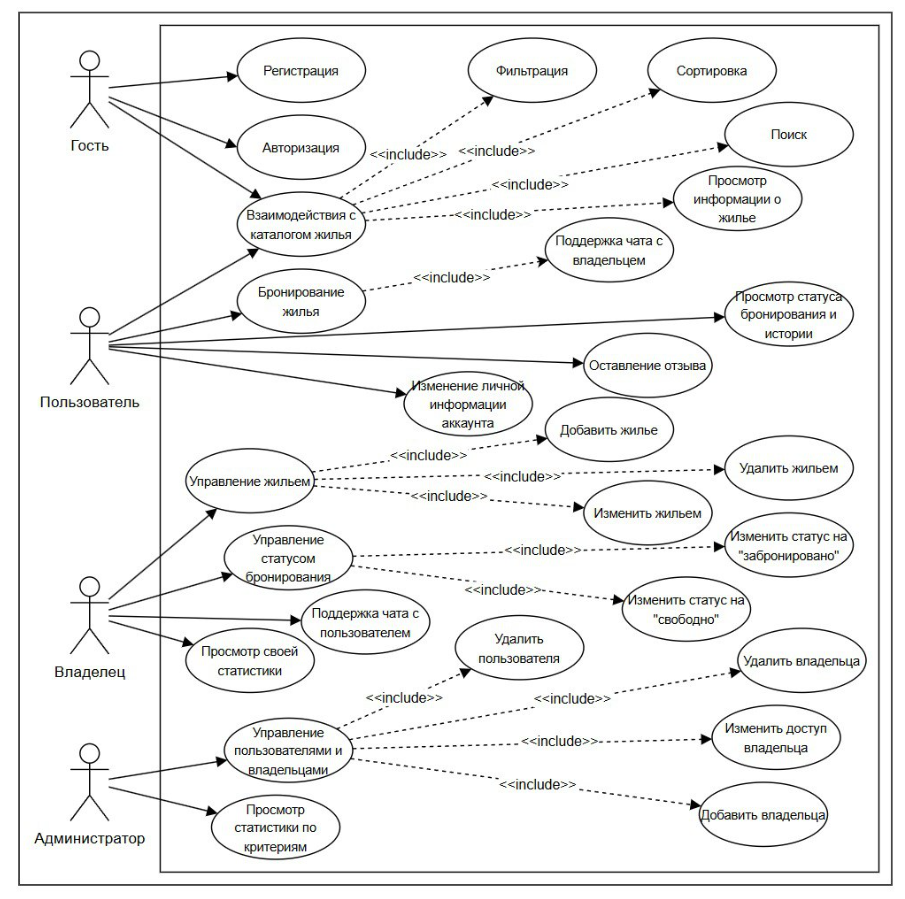


Рисунок 2.1 – Диаграмма вариантов использования

Диаграмма UML вариантов использования позволяет понять, что доступно каждой роли, доступной в данном веб-приложении.

## 2.2 Описание средств разработки

При разработке приложения были использованы:

* интегрированная среда разработки Visual Studio Code;
* платформа Node.js;
* язык разработки проекта TypeScript;
* библиотека React;
* база данных PostgreSQL.

### 2.2.1 Visual Studio Code

Visual Studio Code (VS Code) — это популярная и мощная интегрированная среда разработки, которая поддерживает широкий спектр языков программирования и технологий, включая JavaScript, TypeScript, Python, C++, а также фреймворки и библиотеки, такие как React и Node.js. Одной из главных особенностей VS Code является его легковесность и гибкость, что делает его идеальным инструментом для разработчиков, которые ценят скорость работы и настройку под свои потребности. VS Code предоставляет множество встроенных инструментов для написания, отладки и тестирования кода, а также поддерживает расширения, которые помогают работать с различными базами данных, фреймворками и инструментами. Это позволяет разработчикам эффективно работать на всех этапах разработки, начиная от написания кода и заканчивая его тестированием и деплоем.

### 2.2.2 Платформа Node.js

**Node.js** — это серверная среда выполнения для JavaScript, основанная на движке V8 от Google Chrome. Она позволяет запускать JavaScript-код на сервере, а не только в браузере, что делает возможным создание полноценного серверного приложения. Node.js особенно популярен для создания асинхронных и масштабируемых веб-приложений благодаря своей неблокирующей, ориентированной на события модели ввода-вывода. Она идеально подходит для приложений, которым требуется высокая производительность, таких как чат-системы, онлайн-игры, системы обмена сообщениями, а также для приложений, обрабатывающих большое количество одновременных запросов. Благодаря большому количеству доступных библиотек в **npm** (Node Package Manager), Node.js предоставляет гибкие возможности для разработки приложений различной сложности.

### 2.2.3 Язык программирования TypeScript

**TypeScript** — это язык программирования, являющийся надстройкой над JavaScript, который добавляет статическую типизацию. TypeScript был разработан для того, чтобы сделать процесс разработки на JavaScript более предсказуемым и безопасным, обеспечивая строгую проверку типов на этапе компиляции. Это позволяет легче обнаруживать ошибки и предотвращать их еще до того, как код будет запущен. TypeScript также поддерживает более удобное масштабирование приложений, улучшает читаемость кода и способствует более качественной поддержке крупных проектов. В TypeScript можно использовать все стандартные возможности JavaScript, при этом добавляются дополнительные функции, такие как интерфейсы, перечисления и типы данных, что помогает создавать более структурированные и поддерживаемые приложения.

### 2.2.4 Библиотека React

**React** — это популярная JavaScript-библиотека для создания пользовательских интерфейсов, разработанная компанией Facebook. React позволяет строить веб-приложения на основе компонентного подхода, где каждый компонент представляет собой независимую часть интерфейса с собственным состоянием и поведением. Это упрощает разработку и повторное использование компонентов, что особенно важно при создании крупных и сложных интерфейсов. Одной из ключевых особенностей React является использование виртуального DOM, что помогает повысить производительность приложения. Виртуальный DOM позволяет React минимизировать количество операций с реальным DOM, обновляя только те части интерфейса, которые изменились. Это приводит к ускорению рендеринга и улучшению общего пользовательского опыта. React активно используется в веб-разработке для создания динамичных и отзывчивых интерфейсов, особенно для одностраничных приложений (SPA).

### 2.2.5 PostgreSQL

**PostgreSQL** — это объектно-реляционная система управления базами данных (СУБД) с открытым исходным кодом, которая поддерживает сложные запросы, транзакции, различные типы данных и расширяемость. PostgreSQL является одним из самых популярных решений для хранения данных в крупных и высоконагруженных веб-приложениях. Благодаря поддержке ACID-транзакций, PostgreSQL обеспечивает надежность и консистентность данных. Он также поддерживает работу с географическими данными, а также предоставляет возможность использования хранимых процедур и триггеров, что делает его идеальным выбором для сложных приложений. PostgreSQL также известен своей масштабируемостью и высокими показателями производительности, особенно при обработке больших объемов данных. Его гибкость и расширяемость позволяют интегрировать его в широкий спектр приложений, от малых стартапов до крупных корпоративных решений.

## 2.3 Диаграммы UML, взаимосвязь всех компонентов.

Диаграмма базы данных таблиц (Database Table Diagram) – это визуальное представление структуры базы данных и отношений между таблицами, которые хранятся в этой базе данных. Диаграмма представлена на рисунке 2.2.

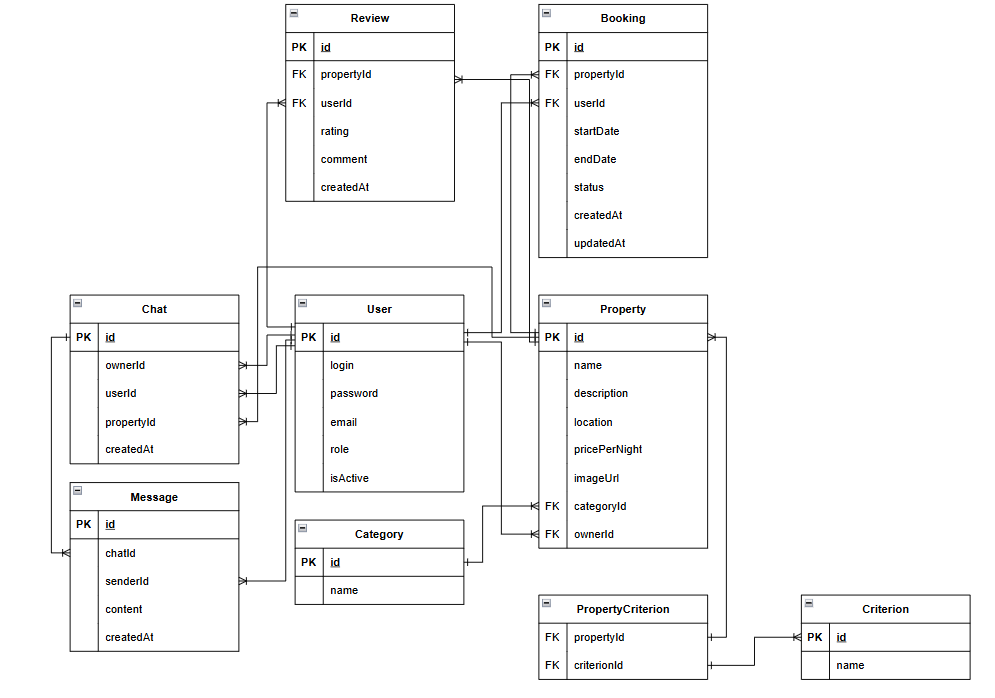


Рисунок 2.2 – Логическая схема базы данных

В этой базе данных используется сложная структура для управления пользователями, недвижимостью, бронированиями и отзывами. Модели и таблицы в базе данных связаны между собой с помощью различных связей и перечислений, что позволяет эффективно хранить и извлекать данные. Давайте подробно разберем каждую из моделей.

Таблица User представляет собой пользователей системы. В этой таблице хранятся такие данные, как уникальный идентификатор пользователя (поля id), его логин (login), зашифрованный пароль (password), а также уникальный адрес электронной почты (email). Это основные поля, необходимые для аутентификации пользователей в системе. Поле role указывает на роль пользователя в системе, которая может быть одной из трех: пользователь (USER), владелец жилья (OWNER), или администратор (ADMIN). Каждому пользователю присваивается одна из этих ролей, которая определяет его доступ к различным функциям приложения. Роль по умолчанию для пользователя — это (USER). Каждый пользователь может иметь несколько связанных объектов недвижимости, если он является владельцем жилья, и это отражается через связь с таблицей Property. Кроме того, пользователи могут оставлять отзывы о жилье и бронировать жилье, что также фиксируется в базе данных в таблицах Review и Booking.

Таблица Property описывает объекты недвижимости, которые могут быть забронированы пользователями. В этой таблице хранится информация о каждом жилье, включая его уникальный идентификатор (id), название (name), описание (description), местоположение (location) и цену за ночь (pricePerNight). Также указывается ownerId, который связывает данное жилье с конкретным пользователем, являющимся его владельцем. Связь с владельцем осуществляется через поле owner, которое ссылается на модель User. Важно, что одно жилье может быть связано только с одним владельцем, и это позволяет организовать структуру, где каждый объект недвижимости принадлежит конкретному пользователю. Кроме того, жилье может быть связано с категориями через поле categoryId, которое указывает на таблицу Category. Категории позволяют классифицировать недвижимость, например, на дома, апартаменты, квартиры и так далее. Связь с категорией реализована через поле category, которое ссылается на модель Category. Для каждого объекта недвижимости можно указать несколько критериев (например, наличие Wi-Fi, бассейна и прочее), которые описаны в таблице Criterion. Эта связь реализована через промежуточную таблицу PropertyCriterion, где каждый объект недвижимости может быть связан с несколькими критериями, что предоставляет гибкость в поиске и фильтрации объектов.

Таблица Review хранит отзывы пользователей о различных объектах недвижимости. Каждый отзыв включает в себя уникальный идентификатор (id), оценку (rating) и комментарий (comment). Эти данные помогают пользователям оценить качество жилья, с которым они сталкивались. Каждый отзыв связан с конкретным жильем через поле propertyId, которое ссылается на модель Property, и с пользователем, который оставил отзыв, через поле userId, которое ссылается на модель User. Кроме того, с помощью уникального ограничения на сочетание propertyId и userId обеспечивается, что каждый пользователь может оставить только один отзыв для конкретного объекта недвижимости. Это важно, чтобы избежать дублирования информации.

Таблица Booking фиксирует информацию о бронированиях, сделанных пользователями. Каждый объект бронирования связан с конкретным жильем через поле propertyId и с пользователем, который сделал бронирование, через поле userId. Бронирование содержит информацию о периоде (даты начала и окончания), а также о статусе бронирования, который может быть в одном из трех состояний: ожидает подтверждения (PENDING), подтверждено (CONFIRMED), или отменено (CANCELLED). Кроме того, в таблице сохраняются временные метки создания (createdAt) и последнего обновления (updatedAt), что позволяет отслеживать изменения в статусе бронирования. Каждое бронирование связано с конкретным жильем, которое было забронировано пользователем, что позволяет легко отслеживать все бронирования для конкретных объектов недвижимости.

Таблица Chat хранит информацию о чатах, используемых для общения между владельцем объекта недвижимости и пользователем, заинтересованным в этом объекте. Каждый чат имеет уникальный идентификатор (id) и связан с объектом недвижимости через поле propertyId, которое ссылается на таблицу Property. Поле ownerId указывает на владельца объекта недвижимости (пользователя из таблицы User), а поле userId — на пользователя, который инициировал общение или заинтересован в объекте. Также хранится временная метка (createdAt), фиксирующая время создания чата. Эта структура позволяет легко отслеживать чаты, группировать их по объектам недвижимости и участникам общения, что делает систему коммуникации удобной и прозрачной.

Таблица Message предназначена для хранения данных о сообщениях, отправленных в рамках чатов. Каждое сообщение имеет уникальный идентификатор (id) и связано с определенным чатом через поле chatId, которое ссылается на таблицу Chat. Поле senderId указывает на отправителя сообщения (пользователя из таблицы User), а в поле content хранится текст сообщения. Дополнительно сохраняется временная метка (createdAt), фиксирующая время отправки сообщения. Эта таблица обеспечивает структурированное хранение всех сообщений, упрощая управление историей общения, идентификацию участников и анализ переписки.

Таблица Category используется для классификации объектов недвижимости. В таблице Category хранится уникальное название каждой категории, например, "Дом", "Апартаменты", "Коттедж". Эта классификация позволяет пользователям удобно искать и фильтровать жилье по типам. Каждая категория может быть связана с несколькими объектами недвижимости, что отображается в поле properties, ссылающемся на таблицу Property.

Таблица Criterion хранит критерии, которые могут быть применены к объектам недвижимости. Примеры таких критериев включают наличие Wi-Fi, бассейна, парковки и так далее. Эти критерии помогают пользователям фильтровать объекты недвижимости по нужным им характеристикам. В таблице Criterion хранится уникальное название критерия, а через промежуточную таблицу PropertyCriterion осуществляется связь между объектами недвижимости и критериями.

Таблица PropertyCriterion является промежуточной таблицей, которая реализует связь "многие ко многим" между объектами недвижимости и критериями. Каждый объект недвижимости может быть связан с несколькими критериями, и каждый критерий может быть связан с несколькими объектами. Это позволяет гибко управлять фильтрацией объектов по множеству параметров. Составной первичный ключ таблицы, состоящий из полей propertyId и criterionId, гарантирует уникальность каждой связи.

Перечисление Role определяет возможные роли пользователей в системе. В системе три роли: USER (соискатель), OWNER (владелец жилья) и ADMIN (администратор). Роль пользователя влияет на его доступ к функционалу приложения. Например, только владельцы жилья могут создавать объекты недвижимости, а администраторы могут управлять всеми пользователями и объектами.

Перечисление BookingStatus определяет возможные статусы бронирования. Это позволяет отслеживать, на каком этапе находится каждое бронирование. Статусы включают: PENDING — бронирование ожидает подтверждения, CONFIRMED — бронирование подтверждено, CANCELLED — бронирование отменено. Это перечисление помогает системе отслеживать процесс бронирования и контролировать состояние каждой заявки.

## 2.4 Выводы по разделу

Разработка архитектуры проекта является важнейшим этапом, поскольку она служит основой для создания структурированной и функционально полноценной системы. В процессе разработки архитектуры важно не только определить основные компоненты приложения, но и четко представить, как эти компоненты будут взаимодействовать друг с другом для достижения цели. В данном разделе мы подробно рассматривали структуру базы данных для системы управления недвижимостью. На основе предложенной архитектуры можно сделать несколько важных выводов.

# 3 Разработка web-приложения

## 3.1 Общая структура

Структура данного проекта включает две ключевые части: серверную и клиентскую. Серверная часть обрабатывает бизнес-логику приложения, управляет взаимодействием с базой данных и обрабатывает запросы, поступающие от клиента. Клиентская часть отвечает за создание пользовательского интерфейса, взаимодействие с сервером и отображение данных пользователю.

## 3.2 Разработка серверной части web-приложения

Серверная часть приложения разработана с использованием платформы Node.js и языка TypeScript. Основной файл проекта — index.ts, который является точкой входа в приложение. В нем инициализируются все необходимые модули и настраиваются маршруты. Взаимодействие с клиентом происходит через различные маршруты, которые в свою очередь направляют запросы на соответствующие контроллеры для обработки.

Для работы с базой данных используется Prisma, который служит эффективным ORM-инструментом для взаимодействия с PostgreSQL. Prisma значительно упрощает выполнение запросов и манипуляцию данными, позволяя работать с базой через удобный API. На рисунке 3.1 будет представлена схема структуры данного проекта.



Рисунок 3.1 – Структура проекта

В приложении основная роль в обработке запросов принадлежит контроллерам и маршрутам. Контроллеры отвечают за обработку входящих HTTP-запросов, а маршруты направляют эти запросы на соответствующие контроллеры для выполнения нужных действий, таких как создание, получение, обновление или удаление данных. Контроллеры выполняют логику обработки запросов и вызывают соответствующие функции для работы с данными, что позволяет организовать код более структурированно.

Каждый контроллер связан с определенным функционалом приложения. Например, в проекте контроллер catalogController занимается созданием жилья с загрузкой изображений, а другие контроллеры, такие как authRoutes, userRoutes, propertyRoutes и adminRoutes, обрабатывают различные функциональные блоки, такие как авторизация, работа с пользователями, управление недвижимостью и администрирование.

Маршруты (routes) служат связующим звеном между входящими запросами и контроллерами. В отличие от контроллеров, маршруты не содержат бизнес-логику, а просто перенаправляют запросы на соответствующие функции в контроллерах. В коде маршруты для авторизации, пользователей, каталога, недвижимости и администрирования подключаются в файле index.ts, который является точкой входа в приложение.

Файл index.ts является главным файлом сервера, где настраиваются маршруты и запускается сервер. В этом файле создается экземпляр приложения Express, подключаются все маршруты через app.use(), а также настраиваются middleware для обработки запросов, таких как CORS, парсинг JSON и загрузка файлов через multer. В конце происходит запуск сервера и подключение к базе данных через Prisma. Таким образом, index.ts является основным файлом, где происходит настройка всего приложения, подключение всех необходимых компонентов и их взаимодействие.

Структура проекта организована таким образом, чтобы контроллеры и маршруты могли легко взаимодействовать друг с другом, при этом каждая часть приложения отвечает только за свою область ответственности. Файл index.ts представлен в листинге 3.1.

const app = express();

const PORT = process.env.PORT || 3000;

app.use(cors({

  origin: 'http://localhost:3001',

}));

app.use(express.json());

const storage = multer.diskStorage({

  destination: (req, file, cb) => {

    cb(null, 'images/');

  },

  filename: (req, file, cb) => {

    const uniqueSuffix = Date.now() + '-' + Math.round(Math.random() \* 1e9);

    cb(null, file.fieldname + '-' + uniqueSuffix + path.extname(file.originalname));

  },

});

const upload = multer({ storage });

app.post('/housing', upload.single('image'), createHousing);

app.use('/images', express.static(path.join(\_\_dirname, '../images')));

app.use('/auth', authRoutes);

app.use('/users', userRoutes);

app.use('/catalog', catalogRoutes);

app.use('/property', propertyRoutes);

app.use('/admin', adminRoutes);

app.listen(PORT, async () => {

  await prisma.$connect();

  console.log(`Сервер запущен на http://localhost:${PORT}`);

});

Листинг 3.1 – Файл index.ts

Файл tsconfig.json в проекте на TypeScript играет ключевую роль в настройке компилятора и определении того, как будет происходить преобразование TypeScript в JavaScript. В вашем случае файл настроен для проекта, использующего Node.js и Express, и включает ряд параметров, которые важны для корректной работы приложения. Данный файл представлен в листинге 3.2.

{

  "compilerOptions": {

    "target": "ES6",

    "module": "commonjs",

    "esModuleInterop": true,

    "strict": true,

    "skipLibCheck": true,

    "forceConsistentCasingInFileNames": true,

    "outDir": "./dist",

    "typeRoots": [

      "./node\_modules/@types",

      "./src/types"],

    "types": ["node", "express"]

  },

  "include": [

    "src/\*\*/\*.ts",

    "src/\*\*/\*.tsx"

  ],

  "exclude": [

    "node\_modules"

  ]

}

Листинг 3.2 – Файл tsconfig.json

target: устанавливает версию ECMAScript, в которую будет компилироваться TypeScript. В вашем случае это ES6, что означает, что компилятор будет генерировать код, соответствующий стандарту ECMAScript 2015 (ES6), включая поддержку классов, стрелочных функций, промисов и других функций этого стандарта.

module: Указывает, какой модульный формат будет использоваться. В вашем случае это commonjs, который является стандартом для Node.js. Это означает, что компилятор будет генерировать код с использованием require() и module.exports для импорта и экспорта модулей.

esModuleInterop: Включает поддержку взаимодействия с модулями, экспортируемыми по стандарту ES, и позволяет использовать их в стиле CommonJS. Это упрощает работу с модулями, которые используют разные стандарты, и позволяет делать импорты в стиле import \* as X from 'module'.

strict: Включает строгий режим, который активирует множество проверок типов, таких как noImplicitAny, strictNullChecks и другие, что помогает предотвратить потенциальные ошибки и улучшить качество кода.

skipLibCheck: Отключает проверку типов в файлах .d.ts (определения типов), что может ускорить компиляцию, особенно в больших проектах с множеством зависимостей.

forceConsistentCasingInFileNames: Устанавливает требование использовать согласованный регистр в именах файлов, предотвращая проблемы с кросс-платформенной совместимостью (например, различия между Windows и Linux).

outDir: Указывает, куда будет скомпилирован результат работы компилятора. В вашем случае это папка ./dist, куда будут выводиться все скомпилированные .js файлы.

typeRoots: Определяет директории, в которых компилятор будет искать типы. В вашем случае компилятор будет искать типы сначала в папке node\_modules/@types (стандартные типы, установленные через npm), а затем в src/types (пользовательские типы, которые вы можете создать).

types: Определяет, какие типы следует подключить. В вашем случае это типы для Node.js и Express, что позволяет использовать их в проекте для автодополнения и проверки типов.

include: указывает, какие файлы TypeScript должны быть включены в проект. В вашем случае это все файлы с расширениями .ts и .tsx в папке src, что означает, что компилятор будет обрабатывать все файлы TypeScript в этой директории.

exclude: определяет, какие файлы или директории должны быть исключены из процесса компиляции. В вашем случае это папка node\_modules, которая не должна быть скомпилирована, так как она уже содержит скомпилированный код и типы для всех установленных зависимостей.

В рамках данного проекта будут рассматриваться следующие основные функции:

* регистрация;
* авторизация;
* управление жильём в системе;
* бронирование жилья;
* оставление отзывов с оцениванием после сделки.

### 3.2.1 Регистрация

В любом приложении одной из ключевых задач является правильная и безопасная регистрация и авторизация пользователей. Эти процессы должны быть спроектированы таким образом, чтобы гарантировать защиту данных пользователей, обеспечивать удобство использования и беспрепятственную интеграцию с другими частями системы. В данном разделе будет рассмотрен процесс регистрации пользователей. Для реализации регистрации в приложении используется контроллер authController, который управляет обработкой запроса на регистрацию. В запросе от клиента передаются такие данные, как email, пароль и роль пользователя, при этом роль пользователя указывается на стороне клиента. Код, реализующий логику регистрации, будет представлен в листинге 3.3.

export const register = async (req: Request, res: Response): Promise<void> => {

  const { login, password, email } = req.body;

    if (existingUser) {

      res.status(400).json({ message: 'Пользователь с таким логином или email уже существует.' });

      return;

    }

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

    const newUser = await prisma.user.create({

      data: {

        login,

        password: hashedPassword,

        email,

        role: 'USER',

      },

    });

    const token = jwt.sign(

      { userId: newUser.id, role: newUser.role },

      JWT\_SECRET,

      { expiresIn: '72h' }

    );

    res.status(201).json({

      message: 'Пользователь успешно зарегистрирован',

      token,

    });

  }

};

Листинг 3.3 – Код регистрации

Код представляет собой обработчик маршрута для регистрации пользователя в приложении. Когда клиент отправляет запрос на регистрацию с данными пользователя, такими как логин, пароль и email, сервер выполняет несколько шагов для создания нового пользователя. Сначала данные из запроса извлекаются и проверяется, существует ли уже пользователь с таким логином или email в базе данных. Если такой пользователь найден, сервер возвращает ошибку, уведомляя, что логин или email уже заняты. Если же такого пользователя нет, то сервер переходит к следующему шагу — хешированию пароля. Для этого используется библиотека bcrypt, которая безопасно преобразует исходный пароль в хеш, чтобы защитить его от несанкционированного доступа. После хеширования пароля создается новый пользователь в базе данных с предоставленными данными. Роль пользователя по умолчанию устанавливается как 'USER'. Также, в этот момент создается JWT токен. Этот токен содержит идентификатор пользователя и его роль, а также срок действия — токен будет действителен в течение 72 часов. JWT токен используется для аутентификации пользователя в дальнейшем. Если процесс регистрации проходит успешно, сервер отправляет клиенту ответ, содержащий сообщение об успешной регистрации и сгенерированный токен. В случае возникновения ошибки в процессе регистрации (например, проблемы с базой данных или хешированием пароля), сервер отправляет ошибку с описанием проблемы. Этот процесс позволяет обеспечить надежную защиту данных пользователей, безопасность паролей и удобное управление сессиями с помощью токенов.

### 3.2.2 Авторизация

Авторизация — это ключевая часть безопасности приложения, которая предоставляет пользователям доступ к ограниченным ресурсам после успешной проверки их личности. Этот процесс включает в себя подтверждение данных пользователя и создание сессии, что позволяет контролировать доступ к защищенным частям системы. В данном случае процесс авторизации осуществляется через проверку учетных данных пользователя, таких как логин и пароль, с последующей выдачей токена для дальнейшего взаимодействия с системой. Логика авторизации обеспечит безопасность и управление правами пользователей в системе. Код, реализующий процесс авторизации, будет представлен в листинге 3.4.

export const login = async (req: Request, res: Response) : Promise<void> => {

  const { login, password } = req.body;

  try {

    const user = await prisma.user.findUnique({

      where: { login },

    });

    if (!user) {

      res.status(400).json({ message: 'Неверный логин или пароль' });

      return;

    }

    const isPasswordValid = await bcrypt.compare(password, user.password);

    if (!isPasswordValid) {

      res.status(400).json({ message: 'Неверный логин или пароль' });

      return;

    }

    const token = jwt.sign({ userId: user.id, role: user.role }, JWT\_SECRET, {

      expiresIn: '72h',

    });

    res.status(200).json({ message: 'Авторизация успешна', token });

  } catch (error) {

    res.status(500).json({ message: 'Ошибка при авторизации', error });

  }

};

Листинг 3.4 – Код авторизации

Этот контроллер реализует процесс авторизации пользователя. Когда пользователь пытается войти в систему, отправляется запрос с логином и паролем. Контроллер извлекает эти данные из тела запроса и ищет пользователя в базе данных, используя логин. Если пользователь с таким логином не найден, сервер отправляет ошибку с сообщением, что логин или пароль неверные. Если пользователь найден, следующим шагом идет проверка пароля. Для этого используется метод bcrypt, который сравнивает введенный пароль с хешированным паролем, хранящимся в базе данных. Если пароли не совпадают, также возвращается ошибка с аналогичным сообщением. Если логин и пароль верны, генерируется JWT токен, который содержит информацию о пользователе (его ID и роль). Этот токен используется для авторизации и доступа к защищенным частям приложения. Токен отправляется пользователю в ответе, и с его помощью он может выполнять дальнейшие запросы в систему без необходимости вводить логин и пароль повторно. Если во время выполнения процесса авторизации возникает ошибка, например, при работе с базой данных или другим компонентом, сервер возвращает сообщение об ошибке с соответствующим кодом состояния.

### 3.2.3 Управление жильём в системе

Управление жильем в системе — это процесс, который включает в себя создание, изменение и удаление информации о жилье, которое доступно на платформе для бронирования. В данной части приложения пользователи (в частности, владельцы жилья) могут добавлять новое жилье в систему, редактировать его характеристики или удалять, если оно больше не доступно. Управление жильем помогает поддерживать актуальность информации на платформе и предоставляет возможность пользователям искать жилье по определенным критериям. В листинге 3.5 будет представлен код для создания жилья.

export const createHousing = async (req: Request, res: Response): Promise<void> => {

  const { name, description, location, pricePerNight, ownerId, categoryId, criteria } = req.body;

  const imageUrl = req.file ? `/images/${req.file.filename}` : null;

  if (!name || !description || !location || !pricePerNight || !ownerId || !categoryId) {

    res.status(400).json({ message: 'Все поля, кроме изображения, обязательны' });

    return;

  }

  try {

    const newProperty = await prisma.property.create({

      data: {

        name,

        description,

        location,

        pricePerNight: parseFloat(pricePerNight),

        owner: {

          connect: { id: parseInt(ownerId) },

        },

        category: {

          connect: { id: parseInt(categoryId) },

        },

        imageUrl,

      },

    });

};

Листинг 3.5 – Код для создания жилья

Контроллер createHousing отвечает за процесс создания нового жилья в системе. Вначале он извлекает данные из тела запроса, такие как название жилья, описание, местоположение, цена за ночь, ID владельца, ID категории и критерии. Также, если изображение жилья было загружено, извлекается путь к файлу. После этого контроллер проверяет, что все обязательные поля присутствуют в запросе. Если какое-то поле отсутствует (кроме изображения), возвращается ошибка с кодом 400 и соответствующим сообщением. Это позволяет предотвратить создание записи с недостаточным набором данных. Если все поля в порядке, контроллер использует Prisma для создания записи о жилье в базе данных. Он связывает новое жилье с владельцем и категорией через их ID, передаваемые в запросе. В случае наличия изображения, сохраняется путь к этому изображению. Если в запросе также указаны критерии для жилья (например, наличие бассейна или Wi-Fi), контроллер связывает эти критерии с новым жильем через промежуточную таблицу, добавляя соответствующие записи в базу данных. По завершении всех операций контроллер возвращает клиенту ответ с кодом 201 (успешно создано), включая данные нового жилья. В случае ошибки (например, при взаимодействии с базой данных) контроллер перехватывает исключение и отправляет ошибку с кодом 500, информируя клиента о проблемах на сервере. Таким образом, контроллер эффективно управляет процессом добавления нового жилья в систему, обеспечивая валидацию данных и корректное взаимодействие с базой данных.

В листинге 3.6 будет представлен код для удаления жилья.

export const deleteHousing = async (req: Request, res: Response): Promise<void> => {

  try {

    const property = await prisma.property.findUnique({

      where: { id: parseInt(id) },

      include: { bookings: true, criteria: true

      },

    });

    await prisma.booking.deleteMany({

      where: {propertyId: parseInt(id),

      },

    });

    await prisma.propertyCriterion.deleteMany({

      where: {propertyId: parseInt(id),

      },

    });

    await prisma.property.delete({

      where: { id: parseInt(id) },

    });

  }

};

Листинг 3.6 – Код для удаления жилья

Контроллер deleteHousing отвечает за удаление жилья из системы. Когда поступает запрос на удаление, контроллер извлекает ID жилья из параметров запроса. Также он получает данные о текущем пользователе, которые предположительно были добавлены в объект req через middleware, например, для проверки авторизации. На первом этапе контроллер проверяет, авторизован ли пользователь. Если данных о пользователе нет (например, он не прошел аутентификацию), возвращается ошибка с кодом 401 (неавторизован) и сообщением, что нужно авторизоваться. Затем контроллер пытается найти жилье в базе данных по ID, который был передан в запросе. Вместе с данными о жилье он также запрашивает связанные с ним бронирования и критерии (если они есть), чтобы удостовериться, что все связанные сущности будут корректно обработаны. Если жилье не найдено, возвращается ошибка с кодом 404 (не найдено), и процесс удаления не продолжается. После того как жилье найдено, контроллер проверяет, имеет ли пользователь право на его удаление. Это право есть только у администратора или владельца этого жилья. Если пользователь не является владельцем жилья или администратором, возвращается ошибка с кодом 403 (Forbidden), информируя, что у пользователя нет прав на выполнение данного действия. Если проверки прошли успешно, контроллер сначала удаляет все бронирования, связанные с данным жильем, через метод deleteMany. Затем он удаляет все связи этого жилья с критериями через таблицу propertyCriterion, которая является промежуточной. Эти операции необходимы для того, чтобы в базе данных не оставалось зависимых данных, которые могут нарушить целостность данных после удаления самого жилья. После того как все зависимые сущности были удалены, контроллер удаляет само жилье через метод delete, и отправляет пользователю ответ с кодом 200 (OK) и сообщением о том, что жилье и все связанные с ним данные были успешно удалены. Если в процессе удаления возникает ошибка, она перехватывается, и пользователю отправляется ответ с кодом 500 (Internal Server Error), а также логируется информация о произошедшей ошибке для дальнейшего анализа. Таким образом, данный контроллер обеспечивает не только удаление самого жилья, но и гарантирует, что все связанные данные, такие как бронирования и критерии, будут корректно удалены, предотвращая проблемы с целостностью базы данных.

В листинге 3.7 будет представлен код для редактирования жилья.

export const updateProperty = async (req: Request, res: Response): Promise<void> => {

  try {

    const updatedProperty = await prisma.property.update({

      where: { id: Number(id) },

      data: {

        name,

        description,

        location,

        pricePerNight: parseFloat(pricePerNight),

        category: categoryId ? { connect: { id: parseInt(categoryId) } } : undefined,

        ...(imageUrl && { imageUrl }),

      },

    });

    if (validCriteria.length > 0) {

      await prisma.propertyCriterion.deleteMany({

        where: { propertyId: Number(id) },

      });

      const criterionConnections = validCriteria.map((criterionId: number) => ({

        propertyId: updatedProperty.id,

        criterionId,

      }));

      await prisma.propertyCriterion.createMany({

        data: criterionConnections,

      });

    }

};

Листинг 3.7 – Код для редактирования жилья

Контроллер updateProperty отвечает за обновление информации о жилье. Сначала он извлекает ID жилья из параметров запроса и данные для обновления из тела запроса. Если переданы данные для загрузки нового изображения, контроллер сохраняет путь к файлу. Если передается критерий (criteria), контроллер проверяет, является ли он строкой или массивом. Если это строка, она преобразуется в массив с проверкой на корректность значений. Если это уже массив, фильтруются только валидные числовые значения. Затем контроллер пытается найти жилье в базе данных по ID. Если жилье не найдено, возвращается ошибка с кодом 404 (не найдено). Если жилье найдено, контроллер обновляет его данные в базе, включая основную информацию (название, описание, местоположение, цена, категория) и, при наличии, новое изображение. Если были переданы корректные критерии, сначала удаляются старые связи этого жилья с критериями, а затем создаются новые, используя промежуточную таблицу для связи жилья и критериев. Наконец, обновленные данные о жилье возвращаются в ответе с кодом 200 (OK). Если возникает ошибка, она логируется, и пользователю отправляется сообщение об ошибке с кодом 500 (внутренняя ошибка сервера).

### 3.2.4 Бронирование жилья

Бронирование жилья — это процесс, позволяющий пользователям резервировать выбранное жилье на определенные даты. В системе бронирования важно учитывать доступность жилья на заданный период, чтобы избежать пересечений с уже подтвержденными бронированиями. Этот функционал обычно включает проверку на доступность, создание заявки на бронирование и управление статусами бронирований (ожидание, подтверждение или отмена). Реализация бронирования будет показана в листинге 3.8.

export const createBooking = async (req: Request, res: Response): Promise<void> => {

  const { propertyId, userId, startDate, endDate } = req.body;

  try {

    const existingBookings = await prisma.booking.findMany({

        where: {

          propertyId: propertyId,

          status: 'CONFIRMED',

          OR: [{

              startDate: { gte: new Date(startDate) },

              endDate: { lte: new Date(endDate) }

            },

            {

              startDate: { lte: new Date(endDate) },

              endDate: { gte: new Date(startDate) }

            }

          ],

        },

      });

    const newBooking = await prisma.booking.create({

      data: {

        userId,

        propertyId,

        startDate: new Date(startDate),

        endDate: new Date(endDate),

        status: 'PENDING',

      },

    });

};

Листинг 3.8 – Создание бронирования жилья

Контроллер createBooking используется для обработки запросов на создание бронирования жилья. Основная задача этого контроллера — гарантировать, что новое бронирование не пересекается с уже существующими, и создать запись о бронировании в базе данных. Когда клиент отправляет запрос, содержащий propertyId (идентификатор жилья), userId (идентификатор пользователя), а также даты начала (startDate) и окончания (endDate) бронирования, контроллер начинает выполнение с проверки доступности жилья на указанный период. Первый шаг — это поиск существующих подтвержденных бронирований (status: 'CONFIRMED') для данного жилья, которые пересекаются с указанным диапазоном дат. Если хотя бы одно пересечение найдено, контроллер возвращает ответ с кодом состояния 400 и сообщает, что указанный период занят. Если никаких пересечений не обнаружено, система переходит к созданию нового бронирования. Для этого в базу данных добавляется запись с переданными userId, propertyId, датами начала и окончания. Новое бронирование изначально получает статус PENDING, что означает, что оно ожидает подтверждения. Это может быть полезно, если система предполагает ручное или автоматическое подтверждение бронирований на следующем этапе. В случае успешного создания бронирования контроллер возвращает ответ с кодом состояния 201, содержащий данные о созданной записи. Если в процессе выполнения возникает ошибка (например, из-за проблем с базой данных), контроллер обрабатывает её и возвращает ответ с кодом состояния 500, указывая, что произошла ошибка на сервере. Этот подход обеспечивает целостность данных и предотвращает конфликты бронирований, сохраняя удобство для пользователей и стабильность работы системы.

### 3.2.5 Оставление отзывов с оцениванием после сделки

Оставление отзывов и оценивание после сделки — это важный механизм, который позволяет пользователям делиться своим опытом взаимодействия с системой или другими пользователями. В контексте аренды жилья отзывы служат инструментом обратной связи, где арендаторы могут оценивать жилье, делиться своими впечатлениями о его состоянии, удобствах и соответствии ожиданиям. Оценки помогают новым пользователям принимать более осознанные решения, а также мотивируют владельцев жилья поддерживать высокий уровень сервиса. Логика работы этого функционала будет представлена в листинге 3.9.

export const addReview = async (req: Request, res: Response): Promise<void> => {

  try {

    if (existingReview) {

      const newReview = await prisma.review.create({

        data: {

          propertyId: Number(propertyId),

          userId: Number(userId),

          rating,

          comment,

        },

        include: { user: true }, // Подключаем данные пользователя

      });

      res.status(201).json(newReview);  // Отправляем новый отзыв

    }

  }

};

Листинг 3.9 – Код для добавления отзыва

Контроллер addReview отвечает за добавление или обновление отзыва пользователя для конкретного жилья. Его задача — обработать данные, отправленные клиентом, проверить существование отзыва и в зависимости от результата либо создать новый, либо обновить уже существующий. Когда запрос поступает, контроллер сначала извлекает из тела запроса следующие данные: идентификатор жилья (propertyId), идентификатор пользователя (userId), оценку (rating) и комментарий (comment). Эти данные нужны, чтобы связать отзыв с конкретным жильем и пользователем. Затем контроллер проверяет, существует ли уже отзыв, оставленный этим пользователем для указанного жилья. Для этого в базе данных выполняется поиск отзыва по уникальной комбинации propertyId и userId. Если такой отзыв находится, он обновляется: заменяются старая оценка и комментарий на новые данные, предоставленные клиентом. После обновления контроллер возвращает обновленный отзыв с кодом успешного выполнения 200. Если отзыв не найден, контроллер создает новый. Он записывает в базу данных все предоставленные данные, включая идентификатор жилья, пользователя, оценку и комментарий. Новый отзыв возвращается клиенту с кодом успешного создания 201. На каждом этапе проверяются возможные ошибки. Если происходит сбой, например, при работе с базой данных, контроллер регистрирует ошибку в журнале и возвращает клиенту сообщение с кодом состояния 500, указывающим на ошибку сервера. Таким образом, этот контроллер реализует удобный механизм, который автоматически определяет, нужно ли обновить существующий отзыв или создать новый, упрощая взаимодействие пользователя с системой.

## 3.3 Разработка клиентской части web-приложения

Клиентская часть веб-приложения была разработана с учетом принципов кроссплатформенности и работает в браузере. Все данные обновляются динамически, благодаря чему взаимодействие с интерфейсом становится плавным и удобным.

Для обмена данными с сервером клиентская часть использует встроенный API fetch. Эта технология позволяет отправлять запросы к серверу и получать ответы в асинхронном режиме, что предотвращает задержки в работе интерфейса. Запросы отправляются с использованием различных HTTP-методов, таких как GET, POST, PUT и DELETE, на заранее определённые пути.

Подход с использованием fetch обеспечивает простоту реализации и достаточную гибкость для большинства задач. Это позволяет динамически обновлять данные на страницах, не прерывая работу приложения. Благодаря этому интерфейс остаётся отзывчивым, а взаимодействие пользователя с приложением — интуитивно понятным.

На рисунке 3.2 представлена структура проекта.

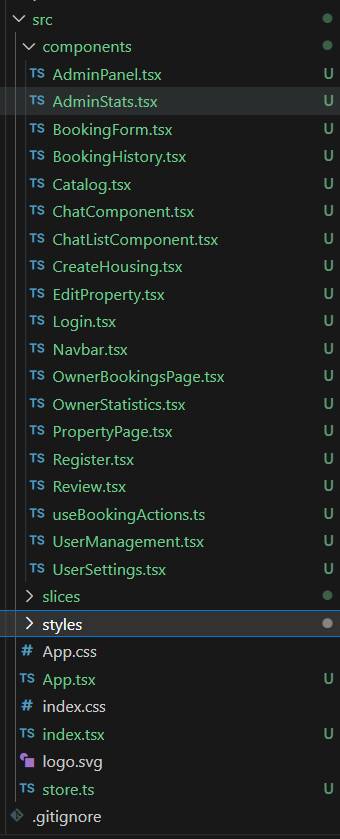


Рисунок 3.2 – Структура проекта

Директория components содержит отдельные файлы компонентов приложения, организованных по функциональному назначению. Это основные части интерфейса. Папка slices содержится userSlice.ts, App.tsx – главный компонент, в котором определены маршруты приложения и подключение основных компонентов, index.tsx – точка входа в клиентское приложение, store.ts – содержит настройку Redux-хранилища для глобального управления состоянием, package.json – файл конфигурации npm, где указаны зависимости, скрипты и метаинформация о проекте, tsconfig.json –конфигурационный файл TypeScript, задающий правила компиляции и обработки кода.

В листинге 3.10 будет представлен слайс, который управляет состоянием.

import { createSlice, PayloadAction } from '@reduxjs/toolkit';

const initialState: UserState = {

  role: localStorage.getItem('role') || null,

  userName: localStorage.getItem('userName') || null,

  userId: localStorage.getItem('userId') || null,

};

const userSlice = createSlice({

  name: 'user',

  initialState,

  reducers: {

    setUserRole(state, action: PayloadAction<string>) {

      state.role = action.payload;

      localStorage.setItem('role', action.payload); // Сохраняем роль в localStorage

    },

    setUserName(state, action: PayloadAction<string>) {

      state.userName = action.payload;

      localStorage.setItem('userName', action.payload);

    },

    setUserId(state, action: PayloadAction<string>) {

      state.userId = action.payload;

      localStorage.setItem('userId', action.payload);

    },

    logout(state) {

      state.role = null;

      state.userName = null;

      state.userId = null;

      localStorage.removeItem('role');

      localStorage.removeItem('userName');

      localStorage.removeItem('userId');

    },

  },

});

Листинг 3.10 – Слайс для хранения состояния

userSlice представляет собой слайс состояния для управления данными о пользователе в Redux. В нем хранятся три основных свойства: роль пользователя (role), имя пользователя (userName) и идентификатор пользователя (userId). Эти данные сохраняются в localStorage, чтобы оставаться доступными при перезагрузке страницы. Инициализация состояния происходит с извлечением этих значений из localStorage, если они там есть.

Слайс включает несколько редьюсеров: setUserRole обновляет роль пользователя в состоянии и сохраняет её в localStorage, setUserName обновляет имя пользователя и также сохраняет его в localStorage, setUserId обновляет идентификатор пользователя и сохраняет его в localStorage, logout сбрасывает данные о пользователе в состоянии и удаляет информацию из localStorage, что позволяет реализовать выход из системы.

Состояние хранится в Redux, и данные о пользователе могут быть легко обновлены с помощью этих редьюсеров.

store будет представлен в листинге 3.11.

import { configureStore } from '@reduxjs/toolkit';

import userReducer from './slices/userSlice';

export const store = configureStore({

  reducer: {user: userReducer,},});

export type RootState = ReturnType<typeof store.getState>;

export type AppDispatch = typeof store.dispatch;

Листинг 3.11 – Код хранилища

store — это конфигурация Redux-хранилища, которое управляет глобальным состоянием приложения. Хранилище настраивается с использованием configureStore из библиотеки @reduxjs/toolkit. В данном случае, хранилище включает редьюсер пользователя (userReducer), который управляет состоянием, связанным с пользователем.

Для удобства работы с состоянием и диспетчером в коде также определены типы: RootState — тип, описывающий всё состояние приложения, который позволяет использовать типизацию при доступе к состоянию, AppDispatch — тип диспетчера, который используется для отправки экшенов в Redux.

Таким образом, store централизует управление состоянием приложения и позволяет эффективно работать с данными о пользователе.

## 3.4 Выводы по разделу

В данном разделе рассмотрены структура и технологии как серверной, так и клиентской частей приложения. Серверная часть основана на Node.js, который реализует регистрацию, авторизацию, управлением жильём, бронирование жилья и оставление отзывов. Клиентская часть построена на React с React-Redux для управления состоянием, а маршрутизация и асинхронная загрузка страниц реализованы с использованием react-router-dom.

# 4 Тестирование web-приложения

## 4.1 Тестирование авторизации и регистрации

При тестировании данного приложения, были применены сценарии, которые могли бы привести к ошибке. В этой главе рассмотрим некоторые такие сценарии и посмотрим на их обработку.

В момент авторизации, возможна такая ситуация, в которой пользователь ничего не ввёл. Обработка данного сценария приведена на рисунке 4.1.



Рисунок 4.1 — Обработка пустых полей при авторизации

При вводе некорректных данных, возникает ошибка, представленная на рисунке 4.2.

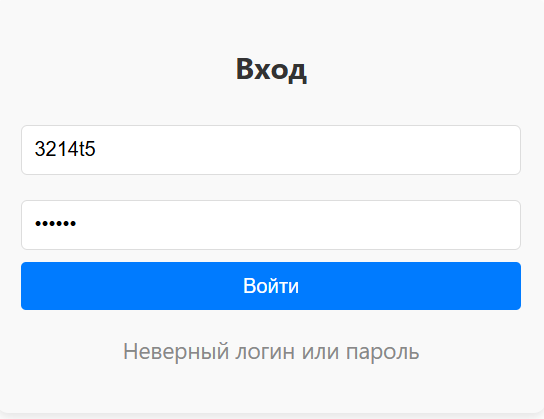


Рисунок 4.2 — Обработка некорректных данных

При регистрации может быть попытка создать аккаунт с логином, который уже занят, поэтому было принято решение проверить это.

Предусловием будет то, что уже создан аккаунт с логином 123, но я буду пытать создать еще один аккаунт с таким логином. Результат данного теста представлен на рисунке 4.3.

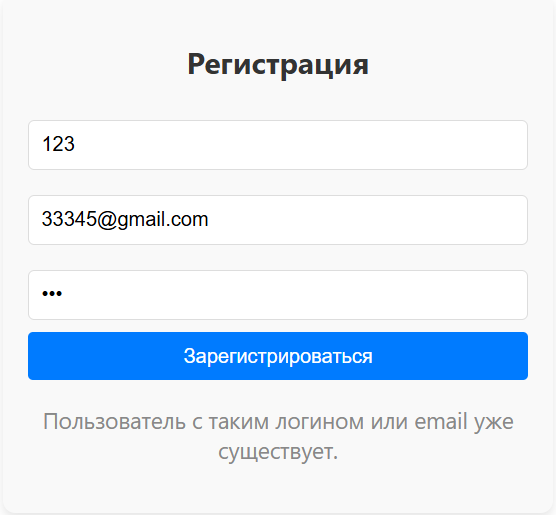


Рисунок 4.3 — Обработка ввода уже зарегистрированного email

Дальше нужно попробовать ввести некорректные данные. Например, некорректную почту. Результат данного тестирования представлен на рисунке 4.4.

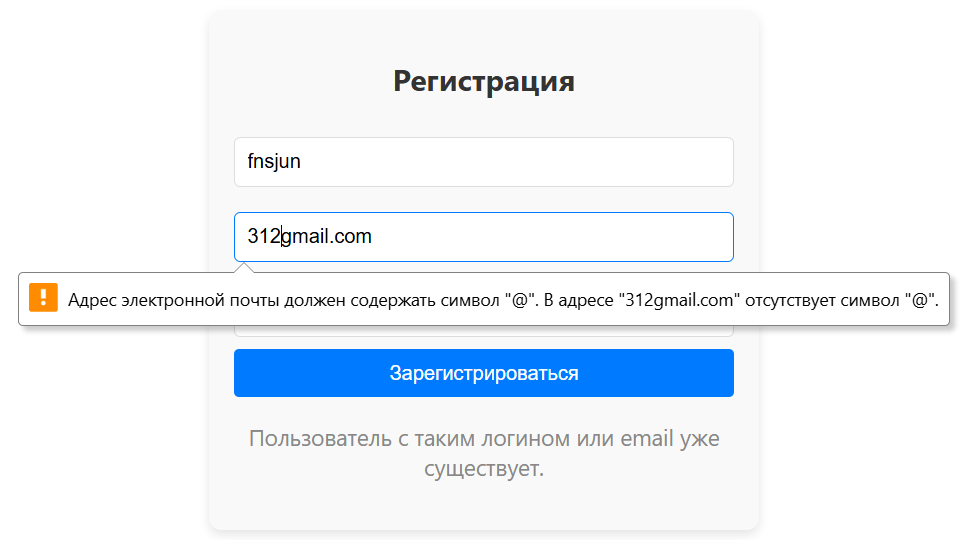


Рисунок 4.4 — Результат c паролем, состоящим из 3 символов

Результаты тестирования авторизации и регистрации говорят о том, что она работает корректно.

## 4.2 Тестирование добавление жилья

Тестирование добавления жилья является важным этапом для проверки корректной работы функционала. Рассмотрим несколько сценариев. Первый сценарий — это попытка добавить жильё с пустыми всеми полями. В этом случае система должна отклонить запрос и отобразить соответствующее сообщение об ошибке.

На рисунке 4.5 представлен результат тестирования.



Рисунок 4.5 – Добавление жилья с пустыми полями

При создании жилья необходимо заполнить обязательные поля. Если они отсутствуют, создание вакансии будет невозможно. Проведенное тестирование подтвердило, что система корректно обрабатывает такой сценарий.

## 4.3 Тестирование поиска вакансий

В функционале поиска предусмотрены определенные поля для ввода текста и выпадающие меню для выбора параметров. Первый сценарий тестирования предполагает ввод пустого названия с несколькими пробелами. Ожидается, что в результате не будет показано жилье.

На рисунке 4.6 представлен результат данного тестирования.

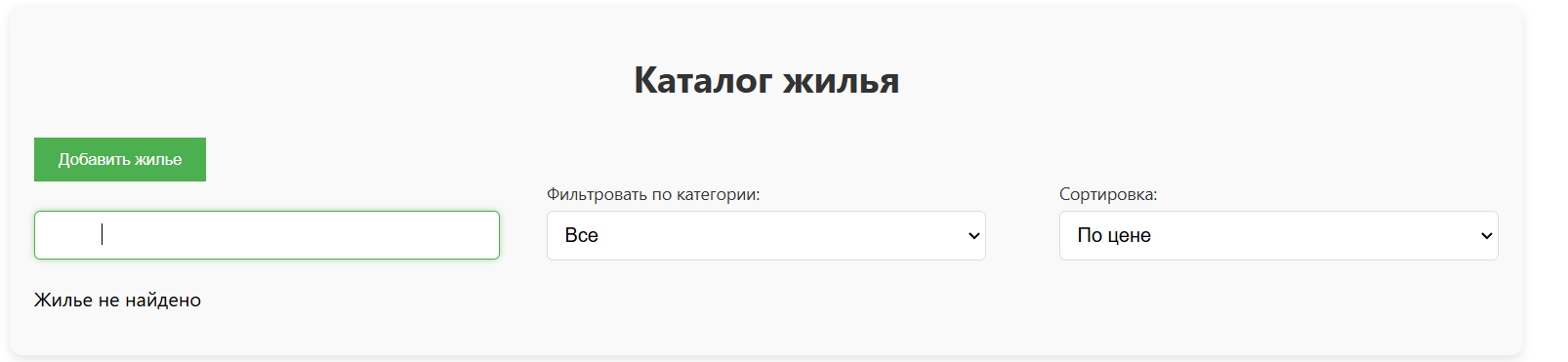


Рисунок 4.6 – Поиск с пробелами в input

Этот тест был успешно выполнен. Следующим этапом было тестирование поиска жилья с именем инс, имея в виду часть слова "Минск". На рисунке 4.7 представлен результат данного тестирования.

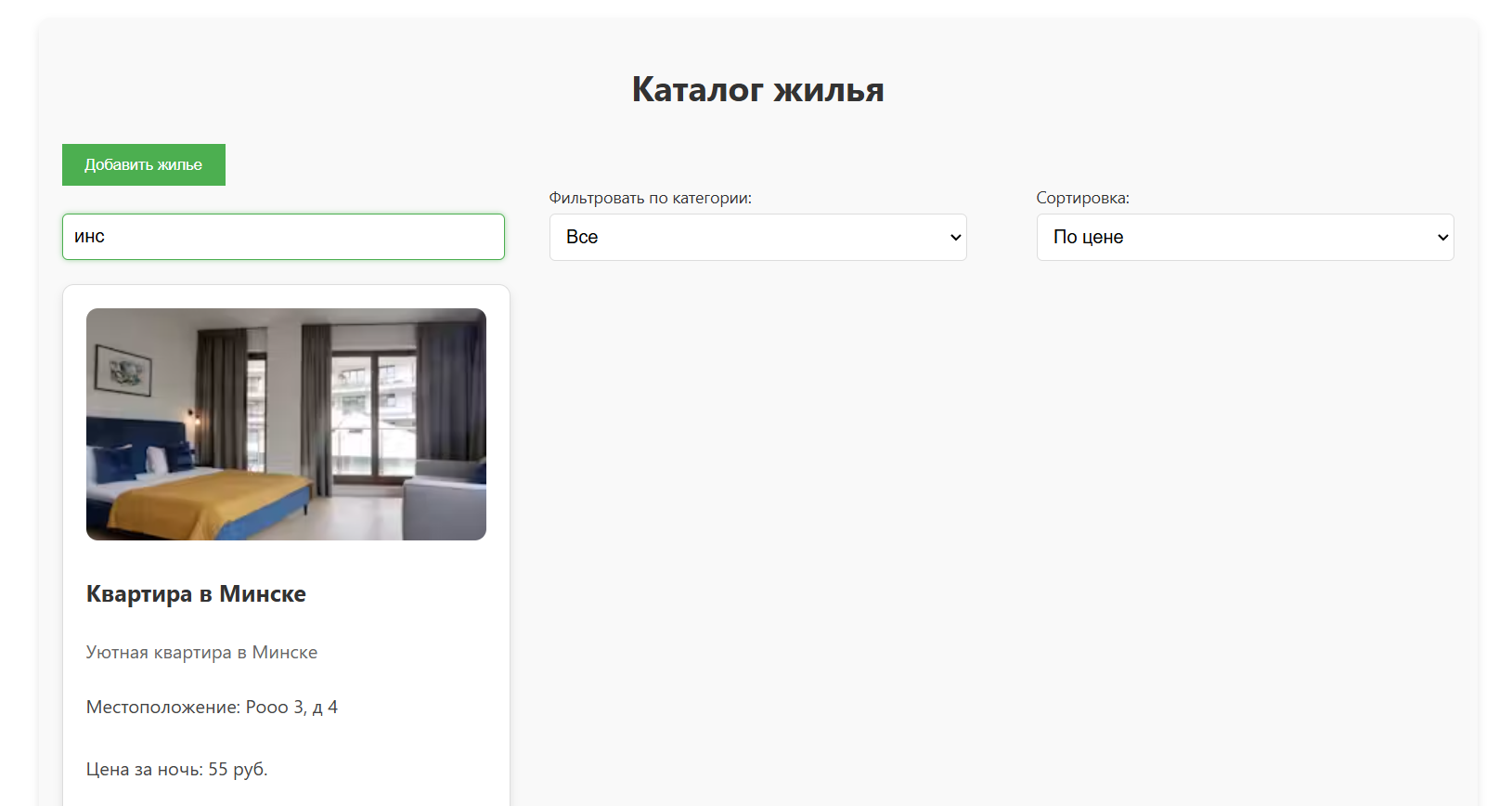


Рисунок 4.7 – Поиск определнным именем

Следующим этапом тестирования была проверка фильтрации по категории жилья. На рисунке 4.8 представлен результат данного тестирования.

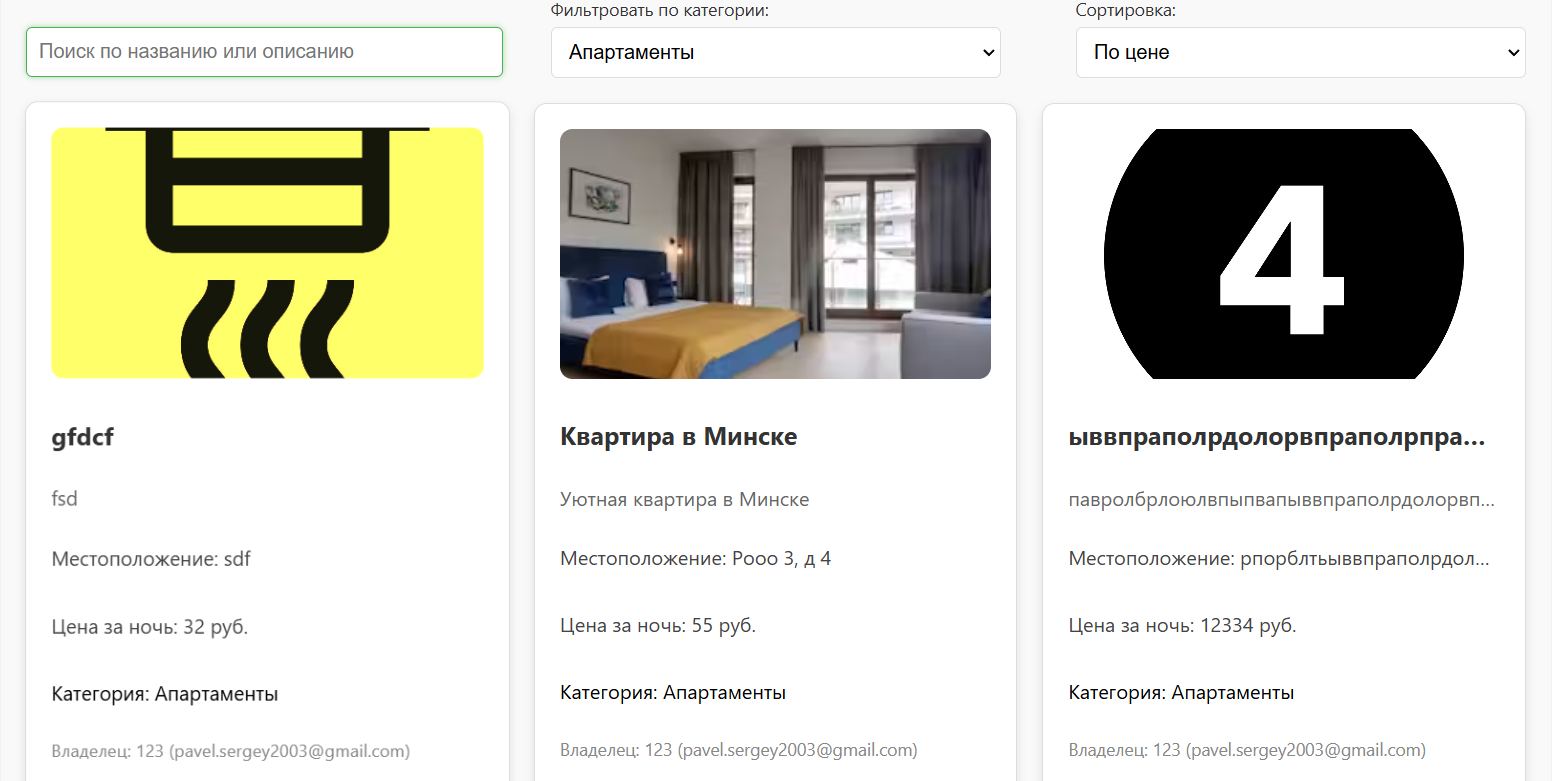


Рисунок 4.8 – Фильтрация по категории

Тесты по поиску жилья были проведены успешно, подтверждая корректную работу фильтров и отображение результатов.

## 4.4 Тестирование изменения настроек пользователя

Для тестирования изменения настроек пользователя был рассмотрен случай, когда логин, на который я хочу поменять уже занят. На рисунке 4.9 представлен результат данного тестирования.

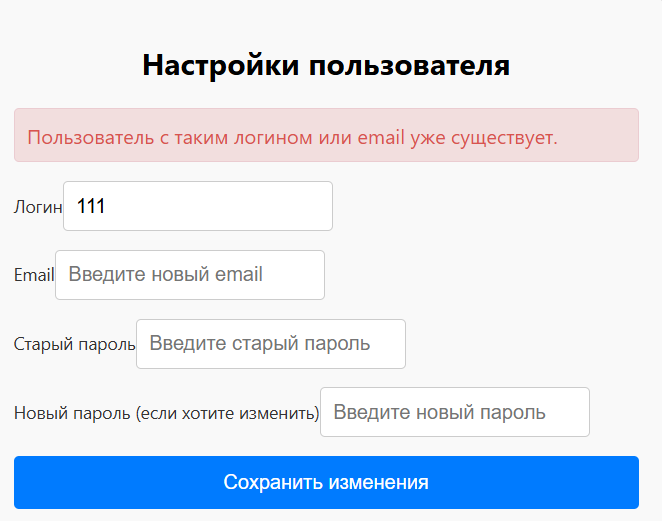


Рисунок 4.9 – Проверка введия существующего логина

Далее была проведена проверка на возможность изменения пароля, введя некорректный старый пароль. На рисунке 4.10 представлен результат данного тестирования.

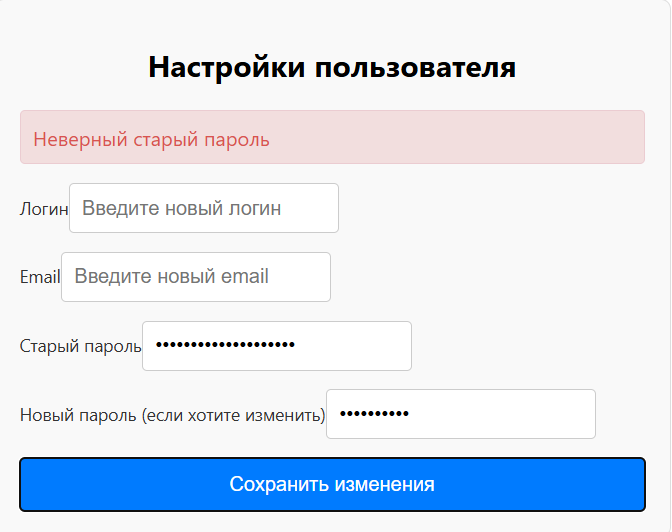


Рисунок 4.10 — Результат некорректного ввода старого пароля

Тест прошел успешно, так как для смены пароля необходимо ввести корректный старый пароль, для подтверждения, что вы являетесь владельцем данного аккаунта.

## 4.5 Выводы по разделу

Таким образом, были проведены тестирования нового функционала, включая добавление жилья, поиск жилья, а также изменения настроек пользователя. Все тесты прошли успешно, функционал работает корректно. Система правильно обрабатывает обязательные поля, фильтрацию данных и обработку новых данных. В случае ввода некорректных данных система корректно генерирует ошибки и предотвращает сохранение неверных данных. Проведенные тесты подтвердили правильность работы всех функций, что свидетельствует о высокой надежности и безопасности приложения.

# 5 Руководство пользователя

## 5.1 Руководство пользователя

При открытии веб-приложения, первой страницей является страница каталога. Если у пользователя еще нет аккаунта, он может перейти на страницу регистрации, где необходимо ввести свой логин, почту и пароль. Страница регистрации, на которой представлены данные поля, представлена на рисунке 5.1.

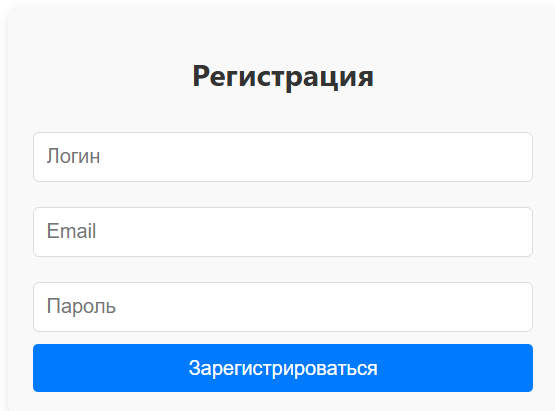


Рисунок 5.1 — Страница регистрации

После регистрации пользователю присваивается роль обычного пользователя, то есть человека, который собирается арендовать жилье через данное приложение.

### 5.1.1 Пользователь

Пользователь также, как и гость может просматривать каталог жилья, искать по названию, сортировать или фильтровать жильё по своему усмотрению. Помимо этого, в шапке сайта можно будет увидеть кнопку для перехода в каталог, имя пользователя, его роль в данный момент, кнопка история бронирований, кнопка стать владельцем, кнопка настроек пользователя, чаты и кнопка для выхода из аккаунта. На рисунке 5.2 представлена страница каталога пользователя.

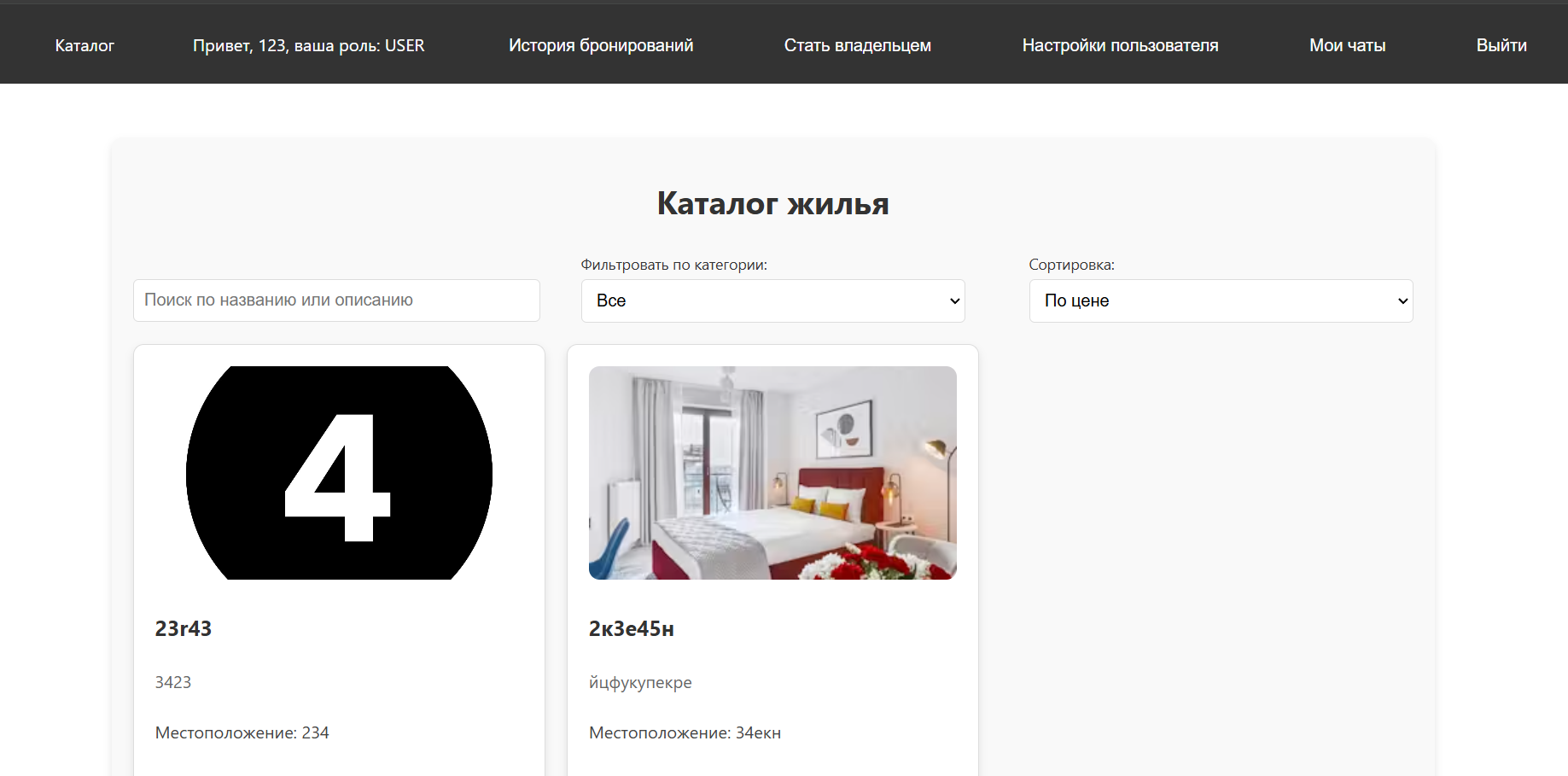


Рисунок 5.2 — Каталог жилья

Кликнув по карточке жилья, пользователь сможет подробнее ознакомится с деталями, а после этого подать запрос на бронирование, если бронирование будет подтверждено, у пользователя появится возможность оставить отзыв под данным жильем, также он может видеть историю бронирований данного жилья. На рисунке 5.3 представлена данная страница, где показана история бронирований, календари, в которых пользователь выбирает даты бронирования, а также раздел с отзывами.

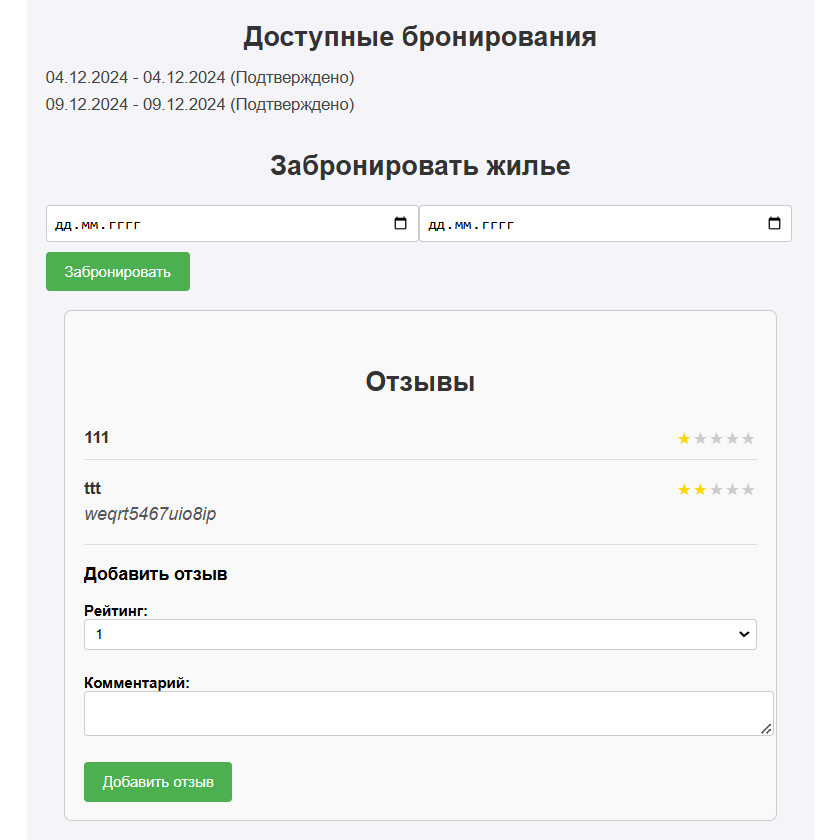


Рисунок 5.3 — Страница жилья

Как уже было сказано до этого в шапке сайта есть кнопка история бронирования, на которой будет отображаться все бронирования данного пользователя. На рисунке 5.4 представлена эта страница.



Рисунок 5.4 — Страница истории бронирования

Помимо этого, есть кнопка настроек пользователя, где можно сменять логин, почту или пароль. Система проверит есть ли данный логин или почта в базе данных и оповестит пользователя, что данные изменены, либо в случае нахождения совпадений в базе данных, оповестит об ошибке. В случае же желания смены пароля, пользователю необходимо будет ввести старый пароль. На рисунке 5.5 представлена данная страница с возможностью смены логина, почты и пароля.

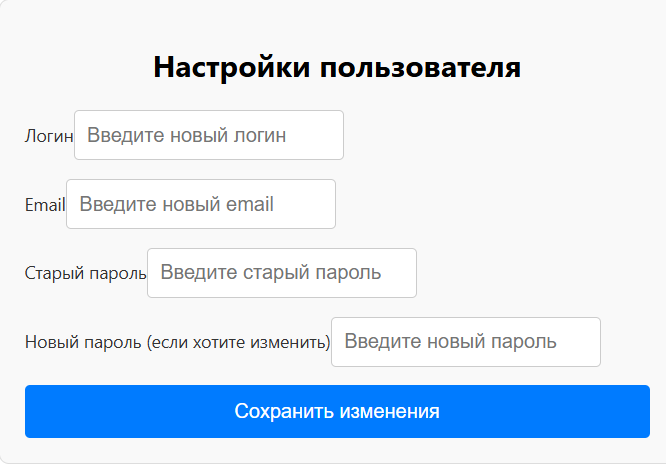


Рисунок 5.5 — Страница настроек пользователя

Помимо всего прочего существует кнопка мои чаты, кликнув на которую, пользователь сможет увидеть список чатов, в которых участвует данный пользователь. Данная страница представлена на рисунке 5.6.

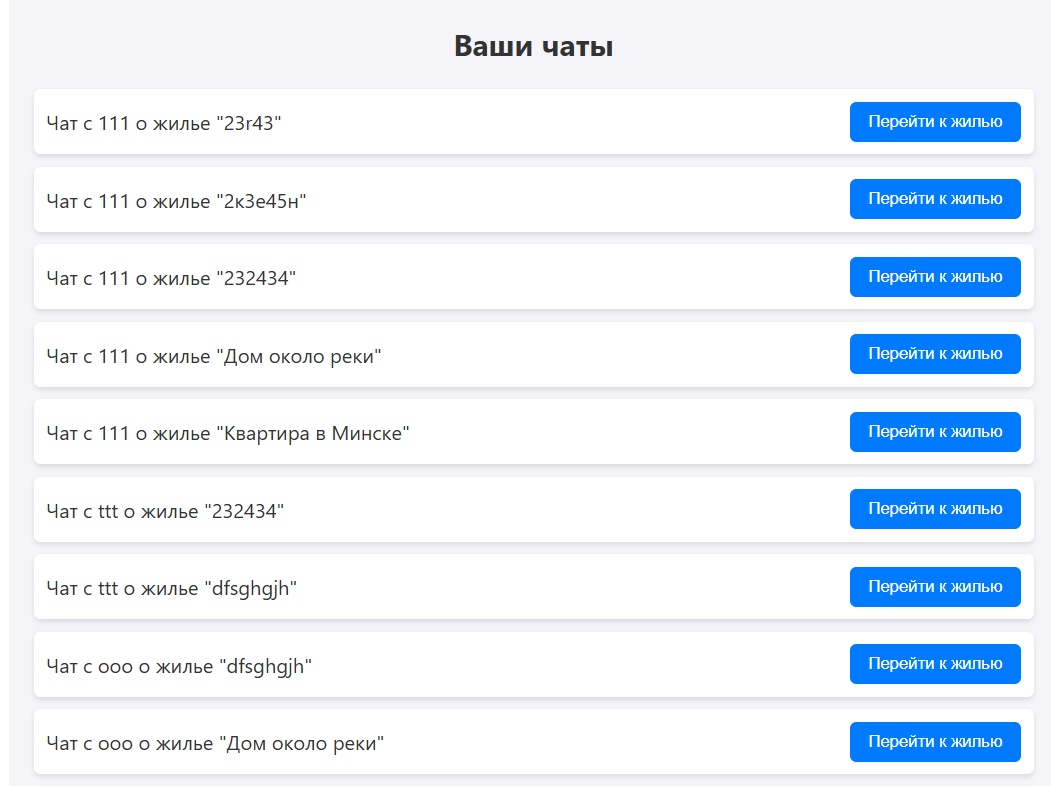


Рисунок 5.6 — Страница с чатами пользователя

Если нажать на кнопку перейти к жилью, то откроется страница с деталями о жилье. И внизу будет отображаться активный чат с владельцем данного жилья. Если его нету, то будет кнопка для создания чата. Данный функционал представлен на рисунке 5.7.

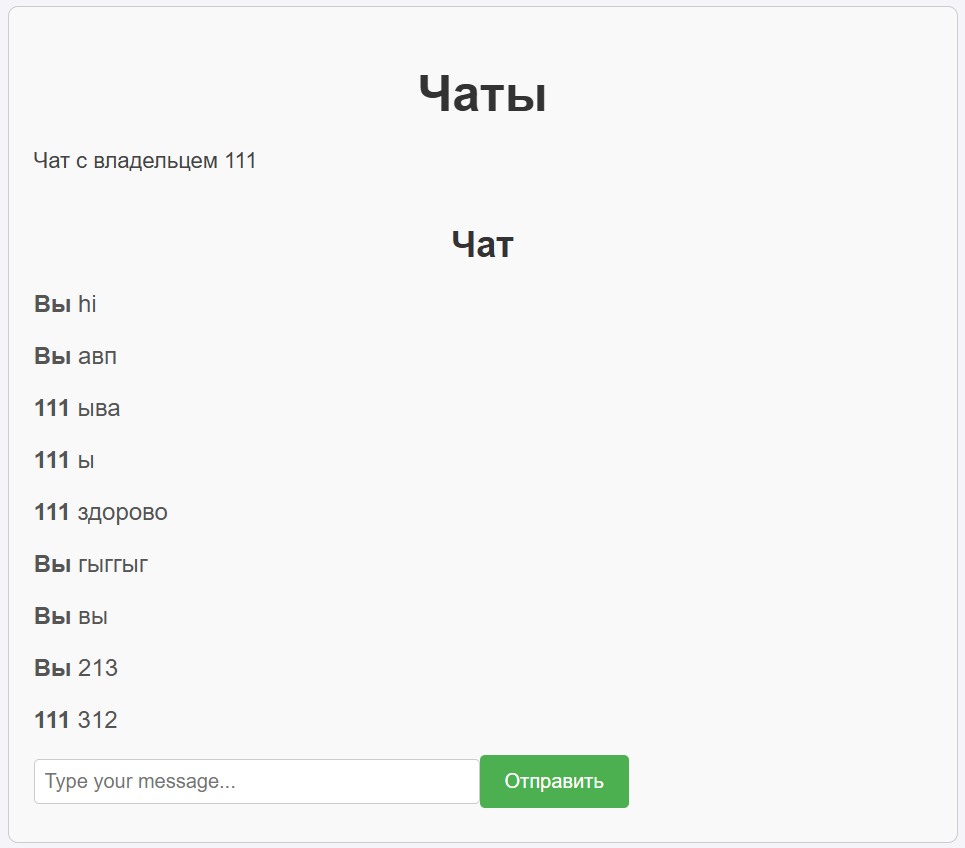


Рисунок 5.7 — Компонент чата

Как уже было сказано, в шапке сайта есть кнопка стать владельцем, нажав на которую у пользователя сменится роль на сайте.

### 5.1.2 Владелец

Став владельцем каталог жилья изменится на жилье, которое находится во владении данного пользователя. Помимо этого, появится кнопка добавления жилья, а также кнопка для удаления жилья. На рисунке 5.8 представлен каталог с новой кнопкой.

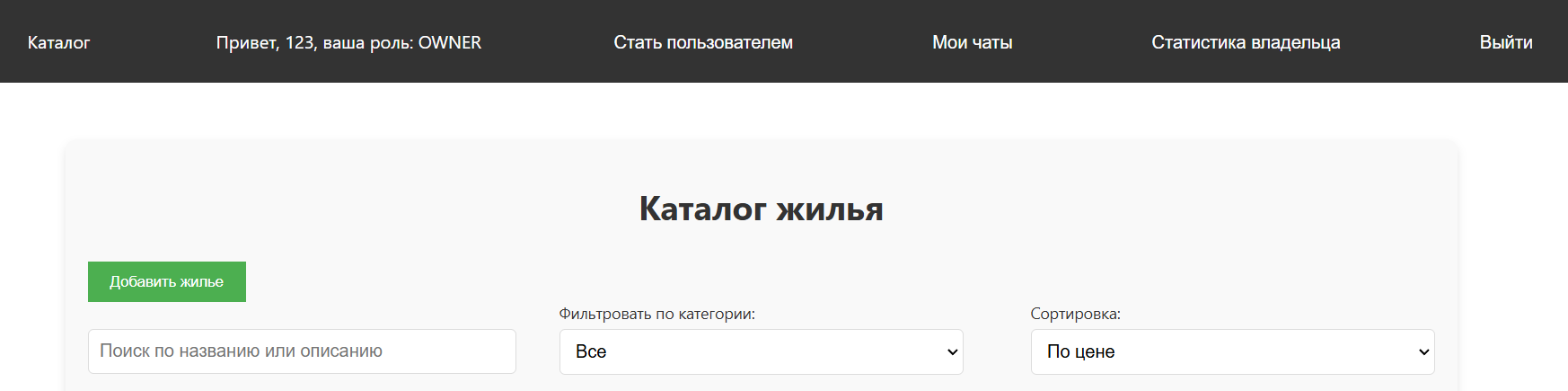


Рисунок 5.8 — Кнопка добавления жилья

Нажав на кнопку добавления жилья, откроется страница, на которой потребуется ввести название жилья, описание, местоположение, цену за ночь, выбрать категорию жилья, при необходимости выбрать определенные критерии, а также выбрать изображение жилья, после чего нажать на кнопку создать. На рисунке 5.9 представлена страница создания жилья.

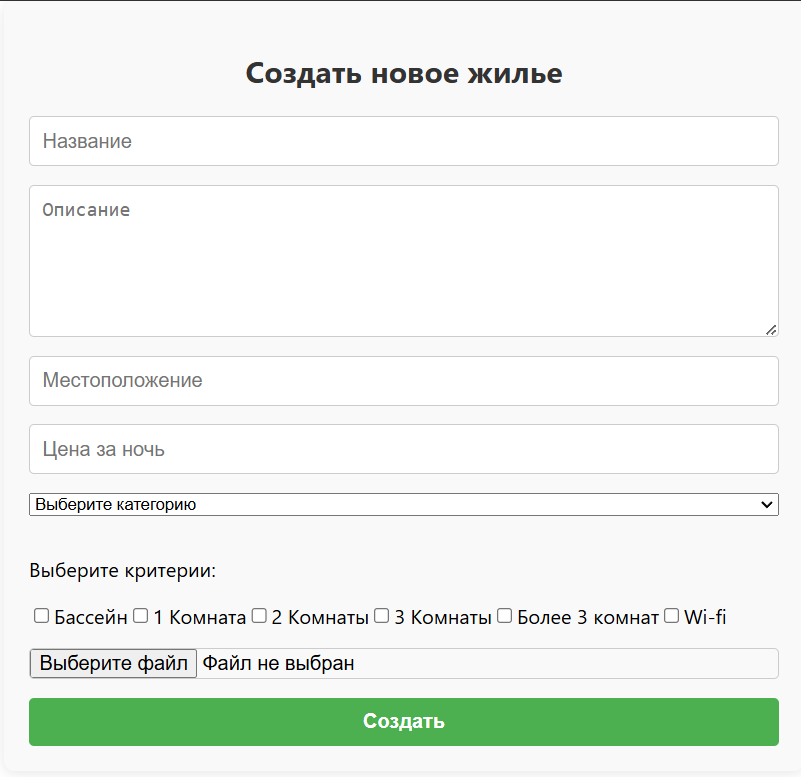


Рисунок 5.9 — Создание жилья

Помимо этого, страница с жильем также претерпела некоторые изменения у владельца. Появилась возможно удаление данного жилья, а также редактирования, помимо этого появилась возможность принять или отменить бронирование как показано на рисунке 5.10.

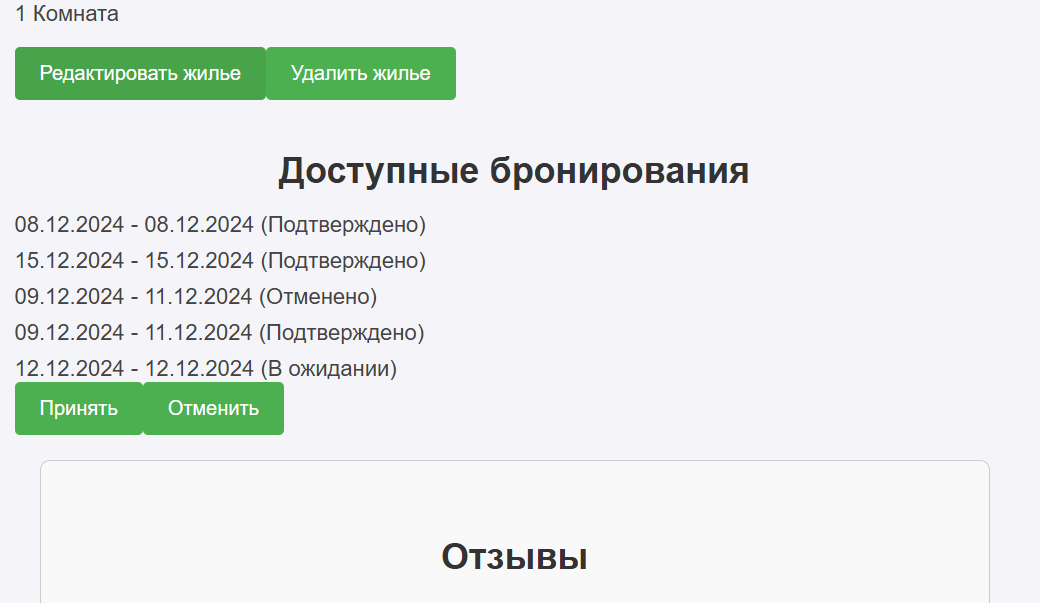


Рисунок 5.10 — Страница информации о жилье для владельца

Страница же редактирования будет содержать старые данные о жилье, а именно: название, описание, местоположение, цену за ночь, категорию, критерии, а также изображение. Данная страница представлена на рисунке 5.11.

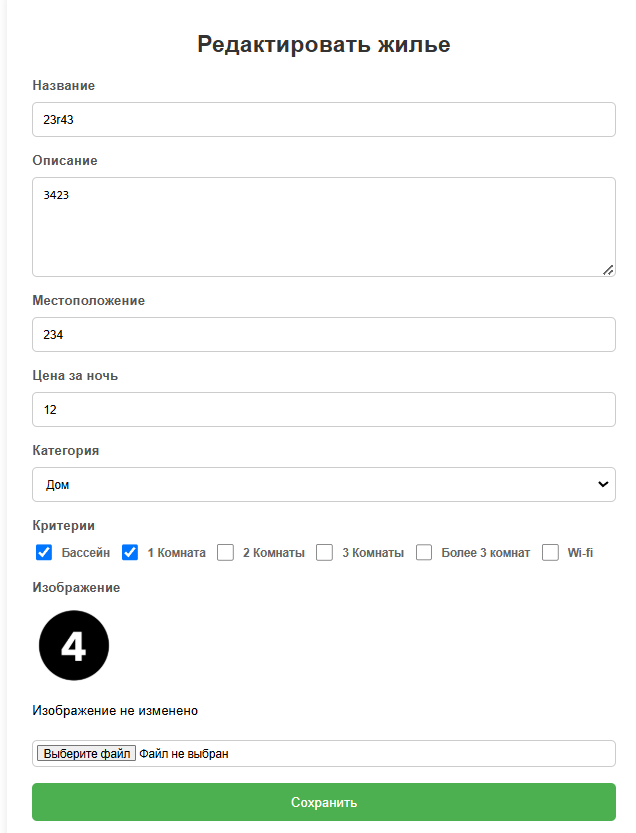


Рисунок 5.11 — Страница редактирования жилья

Итак, будучи владельцем, когда приходит очередной запрос на бронирование в шапке сайта будет появляться кнопка, оповещавшая сколько активных запросов в данный момент времени, нажав на данную кнопку, владелец окажется на странице, в которой будет активные запросы на бронирование у каждого жилья с возможностью перейти в конкретное жилье как показано на рисунке 5.12.

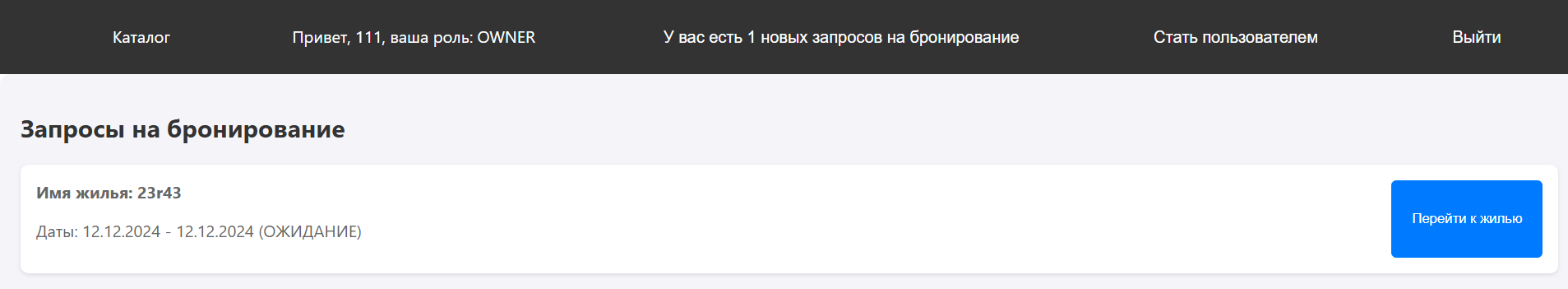


Рисунок 5.12 — Страница с активными запросами на бронирование

И как уже понятно владелец может нажать на кнопку стать пользователем, чтобы снова поменять свою роль, если появилась необходимость арендовать жилье, стоит отметить, что, будучи пользователем в каталоге, он не сможет увидеть свое жилье.

Также есть кнопка статистика владельца, которая представлена на рисунке 5.13.

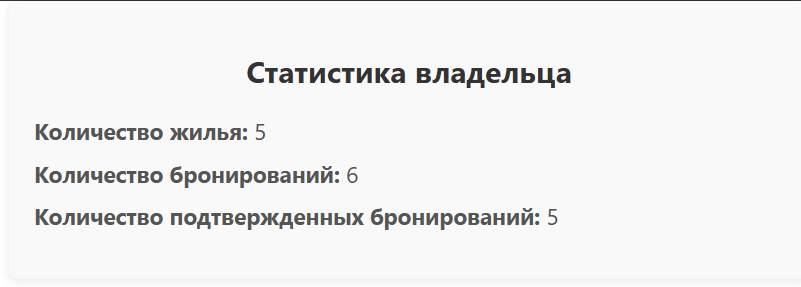


Рисунок 5.13 — Страница со статистикой владельца

На данной странице представлено количество жилья данного владельца, количество бронирований и количество подтвержденных бронирований владельца.

### 5.1.3 Администратор

После авторизации администратора, его перекинет на страницу каталога жилья, где он также, как и владелец будет иметь возможность удалять жилье, помимо этого в шапке сайта, будет кнопка панели админа, как показано на рисунке 5.14.

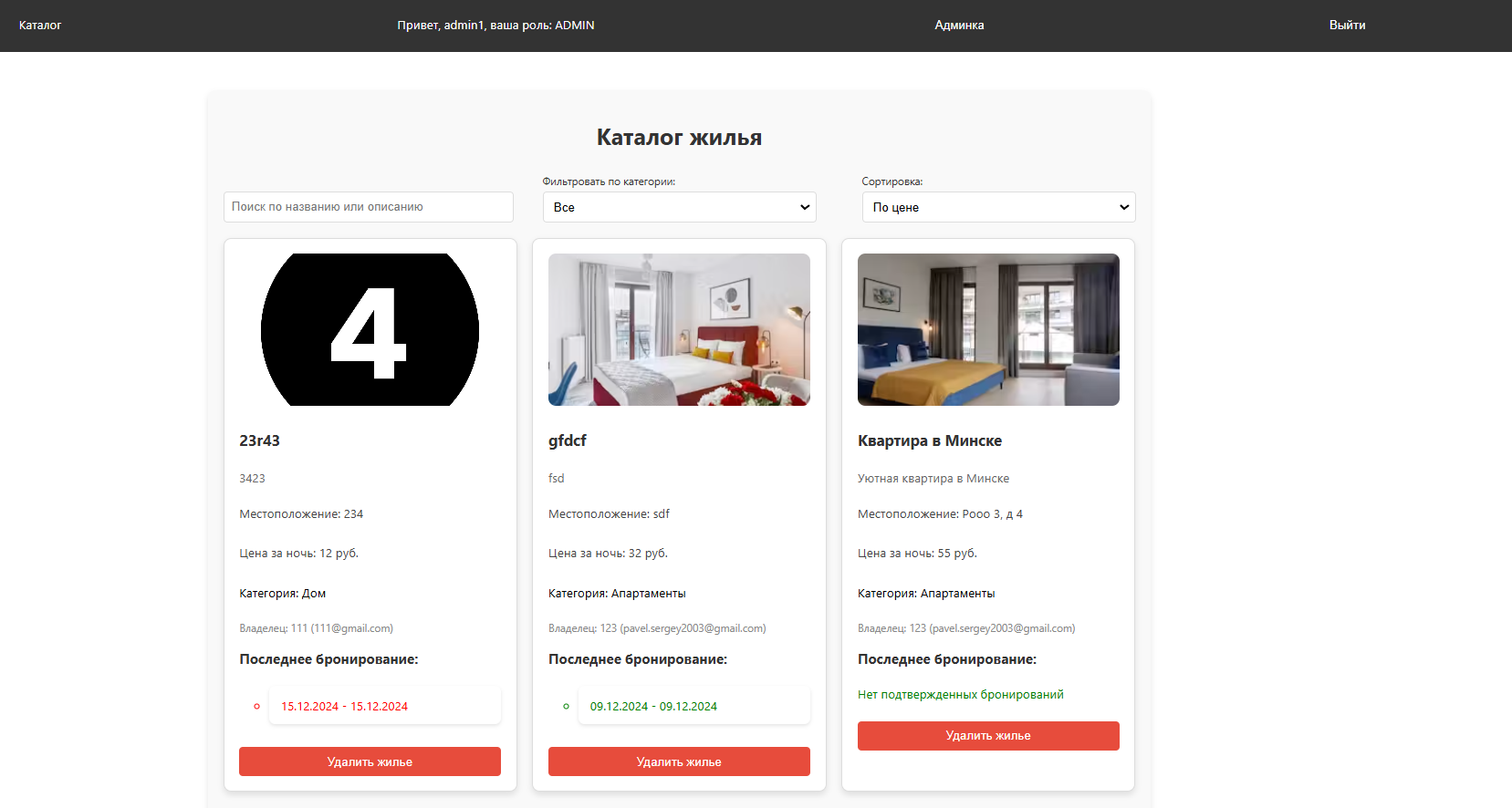


Рисунок 5.14 — Каталог для администратора

Перейдя в панель администратора, можно будет добавлять новые категории и критерии жилья, а также удалять их как показано на рисунке 5.15.

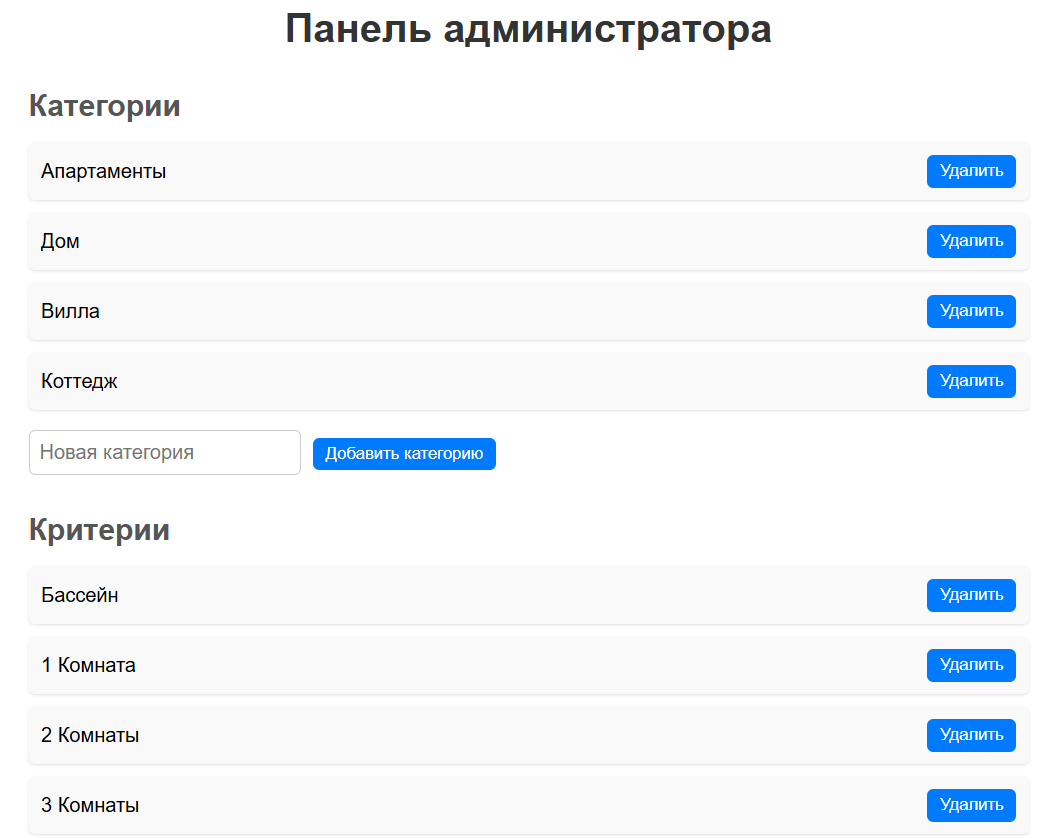


Рисунок 5.15 — Панель администратора

Также тут будут отображаться все пользователи и будет возможность заблокировать и разблокировать пользователя. Вледствии чего, если пользователь заблокирован, после выхода из акаунта заново войти пользователь не сможет, покуда он не будет разблокирован. Данный компонент представлен на рисунке 5.16.

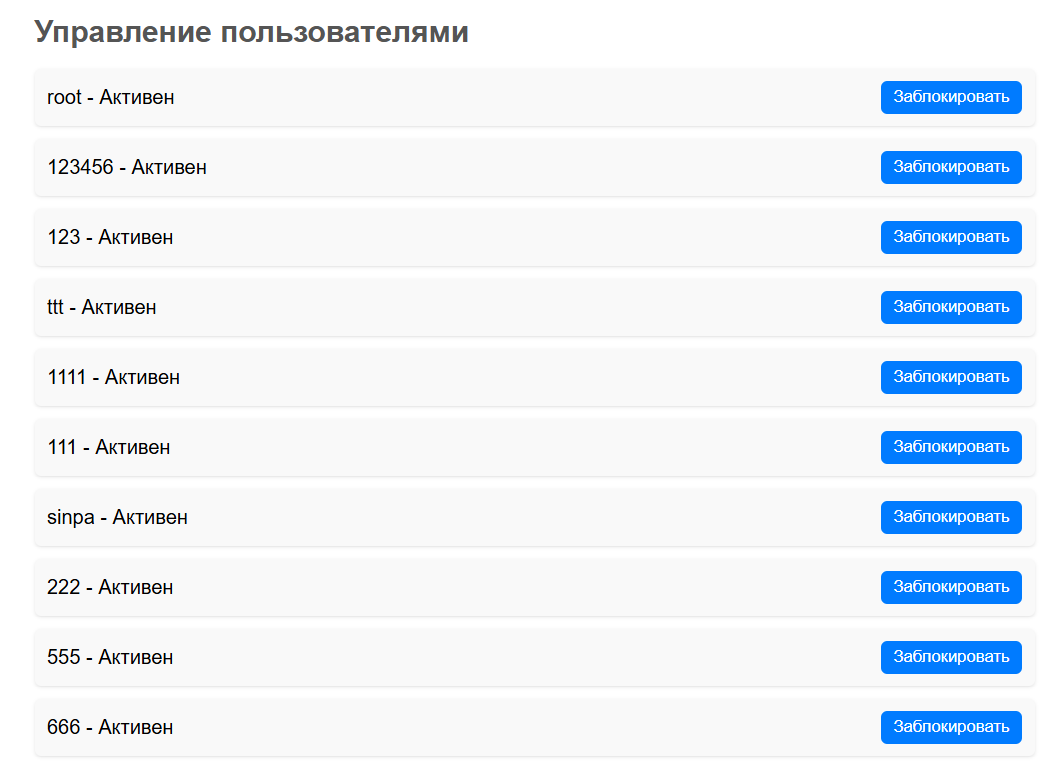


Рисунок 5.16 — Управление пользователями

Помимо этого можно посмотреть статистику, на которой представлено количество пользователей, количество жилья, а также конкретно по каждой категории жилья статистика как показано на рисунке 5.17.

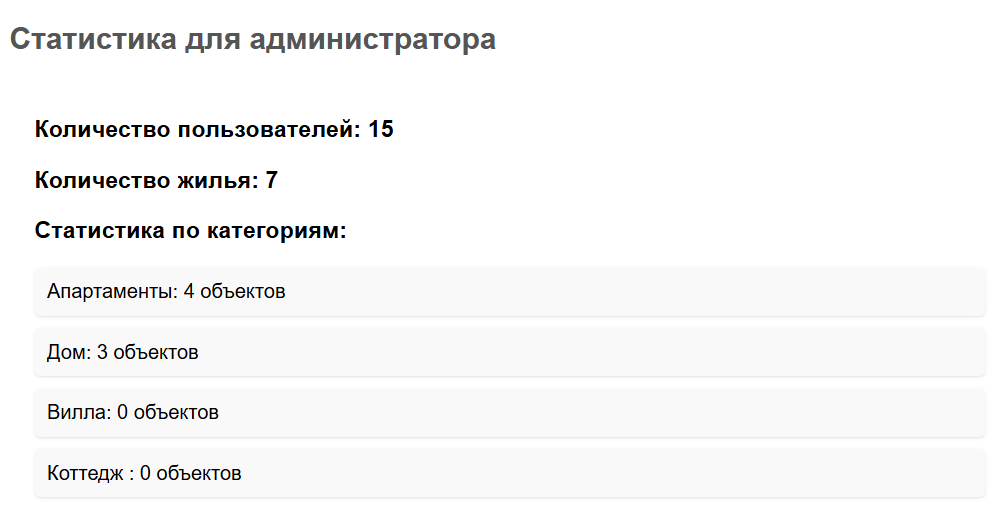


Рисунок 5.17 — Статистика для администратора

Таким образом, выглядит руководство пользователя для администратора.

## 5.2 Установка приложения

Для запуска приложения необходимо выполнить следующие шаги:

1. Запустить серверную часть приложения, которая соединяет базу данных и React приложение. Для этого необходимо запустить скрипт, который настроит соединение с базой данных и запустит сервер.

2. Запустить React приложение, которое будет обрабатывать пользовательские запросы и взаимодействовать с сервером. Для этого необходимо запустить команду для сборки и запуска React приложения.

После выполнения этих шагов приложение будет полностью готово к работе и пользователь сможет начать использовать его функционал.

## 5.3 Выводы по разделу

В данном разделе были рассмотрены руководства пользователя для всех ролей: пользователя, владельца и администратора. Для каждой роли были подробно описаны доступные функции и действия, которые могут быть выполнены на платформе, начиная с регистрации и профиля, заканчивая управлением жильем, бронированием и оставлением отзывов. На примерах с изображениями показано, как правильно использовать приложение и какие действия доступны пользователям в зависимости от их роли. Таким образом, был наглядно продемонстрирован процесс работы с платформой, что делает использование приложения интуитивно понятным и удобным для всех категорий пользователей.

# Заключение

В процессе разработки веб-приложения для аренды жилья была успешно реализована цель создания эффективного инструмента для взаимодействия пользователя и владельца. Для разработки был выбран стек технологий, включающий Node.js, React и PostgreSQL, что обеспечило надежную серверную часть, гибкий и интуитивно понятный интерфейс, а также устойчивое хранение данных.

В ходе выполнения проекта были реализованы следующие ключевые функции:

* регистрация и авторизация пользователей с учетом разных ролей (пользователь, владелец, администратор);
* создание и управление жильем;
* бронирование жилья;
* оставление отзывов после подтверждения бронирования;
* управление категориями и критериями жилья для администратора.

Процесс разработки включал создание базы данных, серверной и клиентской частей приложения, а также интеграцию необходимых функциональных возможностей для обеспечения работы системы. Все функции были тщательно протестированы, и проект прошел все необходимые тесты, подтвердив свою работоспособность и выполнение всех технических требований.

Тестирование показало, что приложение функционирует стабильно, все сценарии работы с вакансиями и профилями пользователей проходят без ошибок. В ходе тестирования не было выявлено проблем, и все поставленные задачи были успешно реализованы.

Разработанное приложение готово к эксплуатации, и его можно использовать для упрощения аренды жилья, обеспечивая удобный и эффективный интерфейс для всех участников процесса.

# Список используемы источников

1. Booking.com [Электронный ресурс] / Режим доступа: https://www.booking.com – Дата доступа: 08.12.2024.
2. **Cian.ru** [Электронный ресурс] / Режим доступа: https://www.cian.ru – Дата доступа: 08.12.2024.
3. Sutochno.ru [Электронный ресурс] / Режим доступа: https://sutochno.ru – Дата доступа: 08.12.2024.
4. PostgreSQL Сайт о программировании [Электронный ресурс] / Режим доступа: https://postgrespro.ru/docs/postgresql.com – Дата доступа: 08.12.2024.
5. Руководство по React [Электронный ресурс] – Режим доступа: https://metanit.com/web/react/– Дата доступа: 08.12.2024.

# Приложение А

import { Request, Response } from 'express';

import prisma from '../db';

// Добавление категории

export const addCategory = async (req: Request, res: Response) : Promise<void> => {

  const { name } = req.body;

  if (!name) res.status(400).json({ message: 'Название категории обязательно' });

  try {

    const category = await prisma.category.create({ data: { name } });

    res.status(201).json(category);

  } catch (error) {

    res.status(500).json({ message: 'Ошибка сервера', error });

  }

};

// Добавление критерия

export const addCriterion = async (req: Request, res: Response) : Promise<void> => {

  const { name } = req.body;

  if (!name) res.status(400).json({ message: 'Название критерия обязательно' });

  try {

    const criterion = await prisma.criterion.create({ data: { name } });

    res.status(201).json(criterion);

  } catch (error) {

    res.status(500).json({ message: 'Ошибка сервера', error });

  }

};

// Удаление категории

export const deleteCategory = async (req: Request, res: Response) => {

  const { id } = req.params;

  try {

    await prisma.category.delete({ where: { id: Number(id) } });

    res.status(200).json({ message: 'Категория удалена' });

  } catch (error) {

    res.status(500).json({ message: 'Ошибка удаления категории', error });

  }

};

// Удаление критерия

export const deleteCriterion = async (req: Request, res: Response) => {

    const { id } = req.params;

    if (!id || isNaN(Number(id))) {

       res.status(400).json({ message: 'Неверный идентификатор критерия' });

    }

    try {

      // Проверка существования критерия

      const existingCriterion = await prisma.criterion.findUnique({ where: { id: Number(id) } });

      if (!existingCriterion) {

         res.status(404).json({ message: 'Критерий не найден' });

      }

      // Удаление связей с этим критерием в промежуточной таблице

      await prisma.propertyCriterion.deleteMany({

        where: { criterionId: Number(id) },

      });

      // Удаление критерия

      await prisma.criterion.delete({ where: { id: Number(id) } });

      res.status(200).json({ message: 'Критерий удален' });

    } catch (error) {

      console.error('Ошибка удаления критерия:', error);

      res.status(500).json({ message: 'Ошибка удаления критерия', error });

    }

  };

// Получение всех категорий

export const getCategories = async (\_req: Request, res: Response) => {

  try {

    const categories = await prisma.category.findMany();

    res.status(200).json(categories);

  } catch (error) {

    res.status(500).json({ message: 'Ошибка загрузки категорий', error });

  }

};

// Получение всех критериев

export const getCriteria = async (\_req: Request, res: Response) => {

  try {

    const criteria = await prisma.criterion.findMany();

    res.status(200).json(criteria);

  } catch (error) {

    res.status(500).json({ message: 'Ошибка загрузки критериев', error });

  }

};

export const getAdminStats = async (req: Request, res: Response): Promise<void> => {

  try {

    // Получаем количество пользователей

    const userCount = await prisma.user.count();

    // Получаем количество жилья

    const propertyCount = await prisma.property.count();

    // Получаем количество объектов для каждой категории

    const categoryStats = await prisma.category.findMany({

      select: {

        name: true, // Название категории

        \_count: {

          select: {

            properties: true, // Количество объектов в категории

          },

        },

      },

    });

    // Формируем статистику

    const stats = {

      userCount,

      propertyCount,

      categories: categoryStats.map((category) => ({

        name: category.name,

        propertyCount: category.\_count.properties,

      })),

    };

    // Отправляем статистику

    res.status(200).json(stats);

  } catch (error) {

    console.error('Ошибка при получении статистики для администратора:', error);

    res.status(500).json({ message: 'Ошибка сервера при получении статистики', error });

  }

};

// Получение списка пользователей, исключая администраторов

export const getUsers = async (req: Request, res: Response) => {

  try {

    const users = await prisma.user.findMany({

      where: {

        role: {

          not: 'ADMIN', // Исключаем пользователей с ролью ADMIN

        },

      },

    });

    res.json(users);

  } catch (error) {

    res.status(500).json({ message: 'Ошибка при получении пользователей' });

  }

};

// Изменение статуса активности пользователя

export const toggleUserStatus = async (req: Request, res: Response) => {

  const { userId } = req.params; // ID пользователя из URL

  const { isActive } = req.body; // Новое значение активности

  try {

    const updatedUser = await prisma.user.update({

      where: { id: Number(userId) },

      data: { isActive },

    });

    res.json(updatedUser);

  } catch (error) {

    res.status(500).json({ message: 'Ошибка при изменении статуса пользователя' });

  }

};

import { Request, Response } from 'express';

import bcrypt from 'bcrypt';

import jwt from 'jsonwebtoken';

import prisma from '../db';

const JWT\_SECRET = 'pahapodoben'; // Используйте безопасное секретное значение и храните его в .env файле

// Регистрация пользователя

export const register = async (req: Request, res: Response): Promise<void> => {

  const { login, password, email } = req.body;

  try {

    const existingUser = await prisma.user.findFirst({

      where: {

        OR: [{ login }, { email }],

      },

    });

    if (existingUser) {

      res.status(400).json({ message: 'Пользователь с таким логином или email уже существует.' });

      return;

    }

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

    const newUser = await prisma.user.create({

      data: {

        login,

        password: hashedPassword,

        email,

        role: 'USER', // Роль по умолчанию

      },

    });

    // Генерация токена

    const token = jwt.sign(

      { userId: newUser.id, role: newUser.role, isActive: newUser.isActive },

      JWT\_SECRET,

      { expiresIn: '1000h' }

    );

    res.status(201).json({

      message: 'Пользователь успешно зарегистрирован',

      token, // Токен с ролью

    });

  } catch (error) {

    console.error("Ошибка при регистрации пользователя:", error);

    res.status(500).json({ message: 'Ошибка при регистрации пользователя', error });

  }

};

// Авторизация пользователя

export const login = async (req: Request, res: Response) : Promise<void> => {

  const { login, password } = req.body;

  try {

    const user = await prisma.user.findUnique({

      where: { login },

    });

    if (!user) {

      res.status(400).json({ message: 'Неверный логин или пароль' });

      return;

    }

    const isPasswordValid = await bcrypt.compare(password, user.password);

    if (!isPasswordValid) {

      res.status(400).json({ message: 'Неверный логин или пароль' });

      return;

    }

    const token = jwt.sign({ userId: user.id, role: user.role, isActive: user.isActive }, JWT\_SECRET, {

      expiresIn: '1000h',

    });

    res.status(200).json({ message: 'Авторизация успешна', token });

  } catch (error) {

    res.status(500).json({ message: 'Ошибка при авторизации', error });

  }

};

export const updateUserInfo = async (req: Request, res: Response): Promise<void> => {

  const { login, email, password, oldPassword } = req.body;

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

  console.log(req.body)

  console.log(oldPassword)

  if (!token) {

    res.status(401).json({ message: 'Не авторизован' });

    return;

  }

  try {

    const decoded = jwt.verify(token, JWT\_SECRET) as { userId: number };

    const user = await prisma.user.findUnique({

      where: { id: decoded.userId },

    });

    if (!user) {

      res.status(404).json({ message: 'Пользователь не найден' });

      return;

    }

    // Если новый пароль передан, проверяем старый пароль

    if (password) {

      if (!oldPassword) {

        res.status(400).json({ message: 'Для изменения пароля необходимо ввести старый пароль' });

        return;

      }

      const isOldPasswordValid = await bcrypt.compare(oldPassword, user.password);

      if (!isOldPasswordValid) {

        res.status(400).json({ message: 'Неверный старый пароль' });

        return;

      }

    }

    const existingUser = await prisma.user.findFirst({

      where: {

        OR: [{ login }, { email }],

        NOT: [{ id: decoded.userId }],

      },

    });

    if (existingUser) {

      res.status(400).json({ message: 'Пользователь с таким логином или email уже существует.' });

      return;

    }

    let updatedPassword = user.password;

    if (password) {

      updatedPassword = await bcrypt.hash(password, 10);

    }

    const updatedUser = await prisma.user.update({

      where: { id: decoded.userId },

      data: {

        login: login || user.login,

        email: email || user.email,

        password: updatedPassword, // Обновляем только если новый пароль был передан

      },

    });

    res.status(200).json({ message: 'Информация пользователя успешно обновлена', user: updatedUser });

  } catch (error) {

    console.error("Ошибка при обновлении информации пользователя:", error);

    res.status(500).json({ message: 'Ошибка при обновлении информации', error });

  }

};

import { Request, Response } from 'express';

import prisma from '../db';

export const deleteHousing = async (req: Request, res: Response): Promise<void> => {

  const { id } = req.params;

  const user = (req as any).user; // Предполагается, что `user` уже добавлен в объект `req` через middleware (например, для авторизации)

  if (!user) {

    res.status(401).json({ message: 'Необходимо авторизоваться' });

    return;

  }

  try {

    const property = await prisma.property.findUnique({

      where: { id: parseInt(id) },

      include: {

        bookings: true, // Добавляем бронирования для проверки

        criteria: true // Добавляем связанные критерии

      },

    });

    if (!property) {

      res.status(404).json({ message: 'Жилье не найдено' });

      return;

    }

    // Проверяем, является ли пользователь владельцем или администратором

    if (user.role !== 'ADMIN' && property.ownerId !== user.userId) {

      res.status(403).json({ message: 'У вас нет прав для удаления этого жилья' });

      return;

    }

    // Удаляем все бронирования, связанные с этим жильем

    await prisma.booking.deleteMany({

      where: {

        propertyId: parseInt(id),

      },

    });

    // Удаляем связи с критериями (если они есть)

    await prisma.propertyCriterion.deleteMany({

      where: {

        propertyId: parseInt(id),

      },

    });

    await prisma.review.deleteMany({

      where: {

        propertyId: parseInt(id),

      },

    });

    // После удаления всех бронирований и связей с критериями, удаляем жилье

    await prisma.property.delete({

      where: { id: parseInt(id) },

    });

    res.status(200).json({ message: 'Жилье и все связанные бронирования успешно удалены' });

  } catch (error) {

    console.error('Ошибка при удалении жилья:', error);

    res.status(500).json({ message: 'Ошибка сервера при удалении жилья', error });

  }

};

export const createHousing = async (req: Request, res: Response): Promise<void> => {

  const { name, description, location, pricePerNight, ownerId, categoryId, criteria } = req.body;

  const imageUrl = req.file ? `/images/${req.file.filename}` : null;

  // Проверка обязательных полей

  if (!name || !description || !location || !pricePerNight || !ownerId || !categoryId) {

    res.status(400).json({ message: 'Все поля, кроме изображения, обязательны' });

    return;

  }

  try {

    // Создание нового жилья

    const newProperty = await prisma.property.create({

      data: {

        name,

        description,

        location,

        pricePerNight: parseFloat(pricePerNight), // Убедитесь, что это число

        owner: {

          connect: { id: parseInt(ownerId) }, // Связь с владельцем

        },

        category: {

          connect: { id: parseInt(categoryId) }, // Связь с категорией

        },

        imageUrl,

      },

    });

    // Если переданы критерии, связываем их с жильем через промежуточную таблицу

    if (criteria && criteria.length > 0) {

      const criterionConnections = criteria.map((criterionId: string) => ({

        propertyId: newProperty.id, // Связь с Property

        criterionId: parseInt(criterionId), // Связь с Criterion

      }));

      // Создаем связи с критериями

      await prisma.propertyCriterion.createMany({

        data: criterionConnections, // Добавляем связи в промежуточную таблицу

      });

    }

    res.status(201).json(newProperty);

  } catch (error) {

    console.error('Ошибка при создании жилья:', error);

    res.status(500).json({ message: 'Ошибка сервера при создании жилья', error });

  }

};

// Получение всех доступных объектов жилья

export const getCatalog = async (req: Request, res: Response): Promise<void> => {

  try {

    // Получаем userId и userRole из query-параметров

    const userId = parseInt(req.query.userId as string); // Получаем userId из query

    const userRole = req.query.userRole as string; // Получаем userRole из query

    // Если роль OWNER, фильтруем жилье по ownerId

    const housing = userRole === 'OWNER' && userId

      ? await prisma.property.findMany({

          where: {

            ownerId: userId, // Фильтруем по ownerId

          },

          include: {

            owner: {

              select: { login: true, email: true, id: true },

            },

            bookings: {

              select: { startDate: true, endDate: true, status: true },

            },

            category: true, // Добавляем информацию о категории жилья

          },

        })

      : userRole !== 'OWNER' && userRole !== 'ADMIN' && userId

      ? await prisma.property.findMany({

          where: {

            ownerId: {

              not: userId, // Фильтруем по ownerId, исключая текущего пользователя

            },

          },

          include: {

            owner: {

              select: { login: true, email: true, id: true },

            },

            bookings: {

              select: { startDate: true, endDate: true, status: true },

            },

            category: true, // Добавляем информацию о категории жилья

          },

        })

        : userRole === 'ADMIN' && userId

      ? await prisma.property.findMany({

          include: {

            owner: {

              select: { login: true, email: true, id: true },

            },

            bookings: {

              select: { startDate: true, endDate: true, status: true },

            },

            category: true, // Добавляем информацию о категории жилья

          },

        })

      : await prisma.property.findMany({

          include: {

            owner: {

              select: { login: true, email: true, id: true },

            },

            bookings: {

              select: { startDate: true, endDate: true, status: true },

            },

            category: true, // Добавляем информацию о категории жилья

          },

        });

    // Если жилье не найдено

    if (housing.length === 0) {

      res.status(404).json({ message: 'Жилье не найдено' });

      return;

    }

    // Отправляем найденное жилье

    res.status(200).json(housing);

  } catch (error) {

    console.error('Ошибка при получении каталога жилья:', error);

    res.status(500).json({ message: 'Ошибка сервера при получении каталога жилья', error });

  }

};

import { PrismaClient } from '@prisma/client';

import { Request, Response } from 'express';

const prisma = new PrismaClient();

// Создание чата (уникальный между ownerId, userId и propertyId)

export const createChat = async (req: Request, res: Response) => {

  const { ownerId, userId, propertyId } = req.body;

  try {

    // Проверяем, существует ли чат

    const existingChat = await prisma.chat.findFirst({

      where: {

        ownerId: Number(ownerId),

        userId: Number(userId),

        propertyId: Number(propertyId), // Учитываем также propertyId

      },

    });

    if (existingChat) {

      // Если чат существует, возвращаем его

      res.status(200).json(existingChat);

      return;

    }

    // Создаём новый чат

    const chat = await prisma.chat.create({

      data: {

        ownerId: Number(ownerId),

        userId: Number(userId),

        propertyId: Number(propertyId), // Добавляем propertyId

      },

    });

    res.status(201).json(chat);

  } catch (error) {

    res.status(500).json({ error: (error as Error).message });

  }

};

// Получение всех чатов пользователя

export const getChats = async (req: Request, res: Response) => {

  const { userId } = req.params;

  try {

    const chats = await prisma.chat.findMany({

      where: {

        OR: [

          { ownerId: parseInt(userId) },

          { userId: parseInt(userId) },

        ],

      },

      include: { messages: true, user: true, owner: true, property: true },

    });

    res.json(chats);

  } catch (error) {

    res.status(500).json({ error: (error as Error).message });

  }

};

// Получение чатов по propertyId (если нужно)

export const getChatsByProperty = async (req: Request, res: Response) => {

  const { userId, propertyId } = req.params;

  try {

    const chats = await prisma.chat.findMany({

      where: {

        propertyId: Number(propertyId),

        OR: [

          { ownerId: parseInt(userId) },

          { userId: parseInt(userId) },

        ],

      },

      include: { messages: true, user: true, owner: true, property: true },

    });

    res.json(chats);

  } catch (error) {

    res.status(500).json({ error: (error as Error).message });

  }

};

// Отправка сообщения

export const sendMessage = async (req: Request, res: Response) => {

  const { chatId, senderId, content } = req.body;

  try {

    const message = await prisma.message.create({

      data: {

        chatId: parseInt(chatId),

        senderId: parseInt(senderId),

        content,

      },

    });

    res.status(201).json(message);

  } catch (error) {

    res.status(500).json({ error: (error as Error).message });

  }

};

// Получение сообщений чата

export const getMessagesByChatId = async (req: Request, res: Response) => {

  const { chatId } = req.params;

  try {

    const messages = await prisma.message.findMany({

      where: { chatId: parseInt(chatId) },

      orderBy: { createdAt: 'asc' }, // Сортировка сообщений по времени

    });

    res.status(200).json(messages);

  } catch (error) {

    res.status(500).json({ error: (error as Error).message });

  }

};

import { Request, Response } from 'express';

import prisma from '../db'; // Подключаем prisma клиент

export const getOwnerStatistics = async (req: Request, res: Response): Promise<void> => {

  const { userId } = req.params;

  try {

    // Получаем статистику по жилью

    const housingCount = await prisma.property.count({

      where: {

        ownerId: Number(userId),

      },

    });

    // Получаем статистику по бронированиям

    const bookingsCount = await prisma.booking.count({

      where: {

        property: {

          ownerId: Number(userId),

        },

      },

    });

    // Получаем статистику по статусам бронирований

    const completedBookingsCount = await prisma.booking.count({

      where: {

        property: {

          ownerId: Number(userId),

        },

        status: 'CONFIRMED',

      },

    });

    // Отправляем статистику владельцу

    res.status(200).json({

      housingCount,

      bookingsCount,

      completedBookingsCount,

    });

  } catch (error) {

    console.error('Ошибка при получении статистики владельца:', error);

    res.status(500).json({ message: 'Ошибка сервера при получении статистики', error });

  }

};

import { Request, Response } from 'express';

import prisma from '../db';

// Получение подробной информации о жилье по ID

export const getPropertyDetails = async (req: Request, res: Response): Promise<void> => {

  const { id } = req.params;

  try {

    const property = await prisma.property.findUnique({

      where: {

        id: parseInt(id), // ID жилья

      },

      include: {

        owner: true, // Включаем владельца

        category: true, // Включаем категорию

        bookings: true, // Включаем бронирования

        criteria: {

          include: {

            criterion: true, // Включаем данные о критериях через промежуточную таблицу

          },

        },

      },

    });

    if (!property) {

      res.status(404).json({ message: 'Жилье не найдено' });

      return;

    }

    // Преобразуем данные для удобного отображения критериев

    const propertyWithCriteria = {

      ...property,

      criteria: property.criteria.map((pc) => pc.criterion), // Извлекаем только критерии

    };

    res.status(200).json(propertyWithCriteria);

  } catch (error) {

    console.error('Ошибка при получении жилья:', error);

    res.status(500).json({ message: 'Ошибка сервера', error });

  }

};

// Получение всех бронирований для владельца

export const getOwnerBookings = async (req: Request, res: Response): Promise<void> => {

  const { userId } = req.params; // Получаем ID владельца из токена (например, из middleware, если он аутентифицирован)

  try {

    const properties = await prisma.property.findMany({

      where: {

        ownerId: parseInt(userId), // Фильтруем по ID владельца

      },

      include: {

        bookings: {

          where: {

            status: 'PENDING', // Фильтруем бронирования по статусу PENDING

          },

        },

      },

    });

    // Если нет жилья или бронирований с PENDING статусом, возвращаем ошибку

    if (!properties || properties.length === 0) {

      res.status(404).json({ message: 'Нет бронирований для вашего жилья' });

      return;

    }

    // Возвращаем только те объекты, которые имеют хотя бы одно бронирование со статусом PENDING

    const filteredProperties = properties.filter(property => property.bookings.length > 0);

    res.status(200).json(filteredProperties); // Возвращаем все фильтрованные объекты

  } catch (error) {

    console.error('Ошибка при получении бронирований:', error);

    res.status(500).json({ message: 'Ошибка сервера', error });

  }

};

// Получить историю бронирований для пользователя

export const getBookingHistory = async (req: Request, res: Response): Promise<void> => {

  try {

    // Извлекаем userId из параметров запроса (например, из JWT или сессии)

    const { userId } = req.params;

    // Получаем все бронирования пользователя с детальной информацией о жилье

    const bookings = await prisma.booking.findMany({

      where: {

        userId: Number(userId), // Преобразуем в число для фильтрации

      },

      include: {

        property: true, // Включаем связанные данные о жилье

      },

    });

    if (bookings.length === 0) {

      res.status(404).json({ message: 'История бронирований не найдена' });

    }

    // Отправляем данные о бронированиях

    res.status(200).json(bookings);

  } catch (error) {

    console.error('Ошибка при получении истории бронирований:', error);

    res.status(500).json({ message: 'Ошибка при получении данных' });

  }

};

// Создание бронирования

export const createBooking = async (req: Request, res: Response): Promise<void> => {

  const { propertyId, userId, startDate, endDate } = req.body;

  try {

    // Проверяем, не пересекаются ли даты с уже существующими бронированиями

    const existingBookings = await prisma.booking.findMany({

        where: {

          propertyId: propertyId,

          status: 'CONFIRMED',

          OR: [

            // Проверяем, пересекаются ли даты бронирования с выбранным диапазоном

            {

              startDate: { gte: new Date(startDate) },

              endDate: { lte: new Date(endDate) } // Пересечение с окончанием выбранного диапазона

            },

            {

              startDate: { lte: new Date(endDate) },

              endDate: { gte: new Date(startDate) } // Пересечение с началом выбранного диапазона

            }

          ],

        },

      });

    if (existingBookings.length > 0) {

      res.status(400).json({ message: 'Этот период уже занят' });

      return;

    }

    // Создаем бронирование

    const newBooking = await prisma.booking.create({

      data: {

        userId,

        propertyId,

        startDate: new Date(startDate),

        endDate: new Date(endDate),

        status: 'PENDING', // Статус бронирования пока что ожидающий

      },

    });

    res.status(201).json(newBooking);

  } catch (error) {

    console.error('Ошибка при создании бронирования:', error);

    res.status(500).json({ message: 'Ошибка сервера при создании бронирования', error });

  }

};

// Контроллер для принятия бронирования

export const acceptBooking = async (req: Request, res: Response): Promise<void> => {

  const { bookingId } = req.params; // ID бронирования из параметров

  try {

    // Находим бронирование по ID

    const booking = await prisma.booking.findUnique({

      where: { id: Number(bookingId) },

    });

    if (!booking) {

      res.status(404).json({ message: 'Бронирование не найдено' });

      return;

    }

    // Проверка, что бронирование еще не подтверждено

    if (booking.status === 'CONFIRMED') {

      res.status(400).json({ message: 'Бронирование уже подтверждено' });

      return;

    }

    // Получаем все подтвержденные бронирования для данного объекта недвижимости

    const existingBookings = await prisma.booking.findMany({

      where: {

        propertyId: booking.propertyId,

        status: 'CONFIRMED', // Только подтвержденные бронирования

      },

    });

    // Проверяем, пересекаются ли даты с уже подтвержденными бронированиями

    const newStartDate = new Date(booking.startDate);

    const newEndDate = new Date(booking.endDate);

    for (const existingBooking of existingBookings) {

      const existingStartDate = new Date(existingBooking.startDate);

      const existingEndDate = new Date(existingBooking.endDate);

      // Проверка на пересечение дат

      if (

        (newStartDate >= existingStartDate && newStartDate < existingEndDate) ||

        (newEndDate > existingStartDate && newEndDate <= existingEndDate) ||

        (newStartDate <= existingStartDate && newEndDate >= existingEndDate)

      ) {

        res.status(400).json({ message: 'Данные даты уже заняты другим бронированием.' });

        return;

      }

    }

    // Если даты не пересекаются, обновляем статус бронирования

    const updatedBooking = await prisma.booking.update({

      where: { id: Number(bookingId) },

      data: {

        status: 'CONFIRMED',

      },

    });

    // Отправляем ответ

    res.status(200).json({ message: 'Бронирование подтверждено', booking: updatedBooking });

    return;

  } catch (error) {

    console.error('Ошибка при подтверждении бронирования:', error);

    res.status(500).json({ message: 'Ошибка при подтверждении бронирования' });

    return;

  }

};

// Контроллер для отмены бронирования

export const cancelBooking = async (req: Request, res: Response): Promise<void> => {

  const { bookingId } = req.params; // ID бронирования из параметров

  try {

    // Находим бронирование по ID

    const booking = await prisma.booking.findUnique({

      where: { id: Number(bookingId) },

    });

    if (!booking) {

      res.status(404).json({ message: 'Бронирование не найдено' });

      return;

    }

    if (booking.status === 'CANCELLED') {

      res.status(400).json({ message: 'Бронирование уже отменено' });

      return;

    }

    // Обновляем статус бронирования

    const updatedBooking = await prisma.booking.update({

      where: { id: Number(bookingId) },

      data: {

        status: 'CANCELLED',

      },

    });

    // Отправляем ответ

    res.status(200).json({ message: 'Бронирование отменено', booking: updatedBooking });

    return;

  } catch (error) {

    console.error('Ошибка при отмене бронирования:', error);

    res.status(500).json({ message: 'Ошибка при отмене бронирования' });

    return;

  }

};

export const updateProperty = async (req: Request, res: Response): Promise<void> => {

  const { id } = req.params;

  const { name, description, location, pricePerNight, categoryId, criteria } = req.body;

  const imageUrl = req.file ? `/images/${req.file.filename}` : null;

  // Инициализируем массив для валидных критериев

  let validCriteria: number[] = [];

  // Если criteria это строка, пробуем распарсить её в массив

  if (typeof criteria === 'string') {

    try {

      // Преобразуем строку в массив

      const parsedCriteria = JSON.parse(criteria);

      // Проверяем, что результат это массив

      if (Array.isArray(parsedCriteria)) {

        // Отфильтровываем только валидные числа

        validCriteria = parsedCriteria

          .map((id: any) => Number(id)) // Преобразуем в числа

          .filter((id: number) => !isNaN(id)); // Отфильтровываем некорректные значения

      }

    } catch (error) {

      console.error('Ошибка при парсинге строки criteria:', error);

      // В случае ошибки можем вернуть пустой массив или какие-то данные об ошибке

    }

  } else if (Array.isArray(criteria)) {

    // Если criteria уже является массивом, то просто фильтруем его

    validCriteria = criteria.filter((id: any) => !isNaN(id)).map(Number);

  }

  try {

    const property = await prisma.property.findUnique({

      where: { id: Number(id) },

      include: { criteria: true }, // Включаем связанные с жильём критерии

    });

    if (!property) {

      res.status(404).json({ message: 'Жилье не найдено' });

      return;

    }

    const updatedProperty = await prisma.property.update({

      where: { id: Number(id) },

      data: {

        name,

        description,

        location,

        pricePerNight: parseFloat(pricePerNight),

        category: categoryId ? { connect: { id: parseInt(categoryId) } } : undefined,

        ...(imageUrl && { imageUrl }), // Обновляем изображение, если оно передано

      },

    });

    // Если были переданы корректные критерии, обновляем их

    if (validCriteria.length > 0) {

      // Удаляем старые связи с критериями

      await prisma.propertyCriterion.deleteMany({

        where: { propertyId: Number(id) },

      });

      // Создаем новые связи с критериями

      const criterionConnections = validCriteria.map((criterionId: number) => ({

        propertyId: updatedProperty.id,

        criterionId,

      }));

      // Создаём связи с критериями

      await prisma.propertyCriterion.createMany({

        data: criterionConnections,

      });

    }

    res.status(200).json(updatedProperty);

  } catch (error) {

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

    res.status(500).json({ message: 'Ошибка сервера при обновлении жилья', error });

  }

};

// Получить все отзывы для конкретного жилья

export const getReviewsByProperty = async (req: Request, res: Response): Promise<void> => {

  try {

    const { propertyId } = req.params;

    // Проверка на правильность формата propertyId

    if (isNaN(Number(propertyId))) {

      res.status(400).json({ message: 'Неверный формат propertyId' });

      return;

    }

    // Получаем все отзывы для данного жилья

    const reviews = await prisma.review.findMany({

      where: {

        propertyId: Number(propertyId), // Фильтрация по propertyId

      },

      include: {

        user: {

          select: {

            id: true,

            login: true, // Отображаем имя пользователя, оставившего отзыв

          },

        },

      },

    });

    // Отправляем отзывы

    res.status(200).json(reviews);

  } catch (error) {

    console.error('Ошибка при получении отзывов:', error);

    res.status(500).json({ message: 'Ошибка при получении данных' });

  }

};

// Добавить новый отзыв

export const addReview = async (req: Request, res: Response): Promise<void> => {

  try {

    const { propertyId, userId, rating, comment } = req.body;

    // Проверка на существование отзыва для данного propertyId и userId

    const existingReview = await prisma.review.findUnique({

      where: {

        propertyId\_userId: {

          propertyId: Number(propertyId),

          userId: Number(userId),

        },

      },

    });

    if (existingReview) {

      // Если отзыв существует, обновляем его

      const updatedReview = await prisma.review.update({

        where: {

          id: existingReview.id, // Обновляем существующий отзыв по его id

        },

        data: {

          rating,

          comment,

        },

        include: { user: true }, // Подключаем данные пользователя

      });

      // Отправляем успешный ответ с обновленным отзывом

       res.status(200).json(updatedReview);

    }else {

      // Если отзыва нет, создаем новый

      const newReview = await prisma.review.create({

        data: {

          propertyId: Number(propertyId),

          userId: Number(userId),

          rating,

          comment,

        },

        include: { user: true }, // Подключаем данные пользователя

      });

      res.status(201).json(newReview);  // Отправляем новый отзыв

    }

  } catch (error) {

    console.error('Ошибка при добавлении/обновлении отзыва:', error);

    res.status(500).json({ message: 'Ошибка при добавлении/обновлении отзыва' });

  }

};

import { Request, Response } from 'express';

import prisma from '../db';

// Обновить роль пользователя

export const updateUserRole = async (req: Request, res: Response): Promise<void> => {

  const { userId } = req.params;

  const { newRole } = req.body;

  try {

    // Проверяем, что роль новая корректная

    if (!['USER', 'OWNER'].includes(newRole)) {

      res.status(400).json({ message: 'Неверная роль' });

      return;

    }

    // Обновляем роль в базе данных

    const updatedUser = await prisma.user.update({

      where: { id: parseInt(userId) },

      data: { role: newRole }, // Роль будет изменена на 'USER' или 'OWNER'

    });

    res.status(200).json({ message: 'Роль обновлена', updatedUser });

  } catch (error) {

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

    res.status(500).json({ message: 'Ошибка сервера при обновлении роли' });

  }

};

import { Router } from 'express';

import {

  addCategory,

  addCriterion,

  deleteCategory,

  deleteCriterion,

  getAdminStats,

  getCategories,

  getCriteria,

  getUsers,

  toggleUserStatus

} from '../controllers/adminController';

const router = Router();

// Роуты для работы с категориями

router.post('/categories', addCategory); // Добавить категорию

router.get('/categories', getCategories); // Получить все категории

router.delete('/categories/:id', deleteCategory); // Удалить категорию

// Роуты для работы с критериями

router.post('/criteria', addCriterion); // Добавить критерий

router.get('/criteria', getCriteria); // Получить все критерии

router.delete('/criteria/:id', deleteCriterion); // Удалить критерий

// Маршрут для получения статистики администратора

router.get('/stats', getAdminStats);

// Получение всех пользователей

router.get('/users', getUsers);

// Изменение статуса активности пользователя

router.patch('/users/:userId/status', toggleUserStatus);

export default router;

import { Router } from 'express';

import { register, login, updateUserInfo } from '../controllers/authController';

const router = Router();

router.post('/register', register);

router.post('/login', login);

// Маршрут для обновления личных данных пользователя

router.put('/user/update', updateUserInfo);

export default router;

// src/routes/catalogRoutes.ts

import { Router } from 'express';

import { getCatalog, createHousing, deleteHousing} from '../controllers/catalogController';

import { authenticate } from '../middleware/authMiddleware'; // Импортируем middleware

const router = Router();

router.get('/', getCatalog); // Получить все объекты жилья

router.post('/', createHousing); // Создать новый объект жилья

router.delete('/:id', authenticate, deleteHousing); // Удалить объект жилья по id с проверкой авторизации

export default router;

import express from 'express';

import { createChat, getChats, getChatsByProperty, getMessagesByChatId, sendMessage } from '../controllers/chatController';

const router = express.Router();

router.post('/chats', createChat);        // Создание чата

// Получение чатов для конкретного пользователя и жилья

router.get('/chats/:userId/:propertyId', getChatsByProperty);

router.get('/chats/:userId', getChats);  // Получение чатов пользователя

router.post('/messages', sendMessage);   // Отправка сообщения

router.get('/messages/:chatId', getMessagesByChatId); // Новый маршрут

export default router;

import express from 'express';

import { getOwnerStatistics } from '../controllers/ownerController';

const router = express.Router();

router.get('/statistics/:userId', getOwnerStatistics);

export default router;

import express from 'express';

import { getPropertyDetails, createBooking, acceptBooking, cancelBooking, updateProperty, getOwnerBookings, getBookingHistory, getReviewsByProperty, addReview } from '../controllers/propertyController';

import { authenticate } from '../middleware/authMiddleware';

import multer from 'multer';

import path from 'path';

// Создание хранилища для файлов

const storage = multer.diskStorage({

  destination: (req, file, cb) => {

    cb(null, 'images/'); // Папка для сохранения изображений

  },

  filename: (req, file, cb) => {

    const uniqueSuffix = Date.now() + '-' + Math.round(Math.random() \* 1e9);

    cb(null, file.fieldname + '-' + uniqueSuffix + path.extname(file.originalname)); // Генерация уникального имени файла

  },

});

const upload = multer({ storage }); // Настройка multer

const router = express.Router();

// Получение подробной информации о жилье

router.get('/:id', getPropertyDetails);

// Получение подробной информации о жилье

router.get('/owner/:userId', getOwnerBookings);

// Маршрут для получения истории бронирований пользователя

router.get('/user-history/:userId', getBookingHistory);

// Создание бронирования

router.post('/booking', createBooking);

// Принять бронирование

router.put('/booking/:bookingId/accept', authenticate, acceptBooking);

// Отменить бронирование

router.put('/booking/:bookingId/cancel', authenticate, cancelBooking);

// Редактирование информации о жилье с загрузкой изображения

router.put('/:id', upload.single('image'), updateProperty);

router.get('/:propertyId/reviews', getReviewsByProperty);

router.post('/reviews', addReview);

export default router;

import { Router } from 'express';

import { updateUserRole } from '../controllers/userController';

const router = Router();

// Обновить роль пользователя

router.put('/:userId/role', updateUserRole);

export default router;

import express from 'express';

import cors from 'cors';

import multer from 'multer';

import path from 'path';

import { createHousing } from './controllers/catalogController'; // Импорт контроллера

import authRoutes from './routes/authRoutes';

import catalogRoutes from './routes/catalogRoutes';

import prisma from './db';

import userRoutes from './routes/userRoutes';

import propertyRoutes from './routes/propertyRoutes';

import adminRoutes from './routes/adminRoutes';

import chatRoutes from './routes/chatRoutes';

import ownerRoutes from './routes/ownerRoutes';

const app = express();

const PORT = process.env.PORT || 3000;

app.use(cors({

  origin: 'http://localhost:3001',

}));

app.use(express.json());

const storage = multer.diskStorage({

  destination: (req, file, cb) => {

    cb(null, 'images/'); // Папка для сохранения изображений

  },

  filename: (req, file, cb) => {

    const uniqueSuffix = Date.now() + '-' + Math.round(Math.random() \* 1e9);

    cb(null, file.fieldname + '-' + uniqueSuffix + path.extname(file.originalname));

  },

});

const upload = multer({ storage });

app.post('/housing', upload.single('image'), createHousing); // Маршрут для создания жилья с загрузкой изображения

app.use('/images', express.static(path.join(\_\_dirname, '../images')));

app.use('/auth', authRoutes);

app.use('/users', userRoutes); // Применяем маршруты для работы с пользователями

app.use('/catalog', catalogRoutes);

app.use('/property', propertyRoutes);

app.use('/admin', adminRoutes);

app.use('/api', chatRoutes);

app.use('/stat', ownerRoutes);

// Запуск сервера

app.listen(PORT, async () => {

  await prisma.$connect();

  console.log(`Сервер запущен на http://localhost:${PORT}`);

});

Листинг 1 – Серверная часть

# Приложение Б

|  |
| --- |
| import React, { useState, useEffect } from 'react';  import { useSelector } from 'react-redux';  import { useNavigate } from 'react-router-dom';  import '../styles/AdminPanel.css';  import AdminStats from './AdminStats';  import UserManagement from './UserManagement'; // Импортируем компонент UserManagement  import '../styles/UserManagement.css'; // Импортируем стили для UserManagement  const AdminPanel: React.FC = () => {    const [categories, setCategories] = useState<{ id: number; name: string }[]>([]);    const [criteria, setCriteria] = useState<{ id: number; name: string }[]>([]);    const [newCategory, setNewCategory] = useState<string>('');    const [newCriterion, setNewCriterion] = useState<string>('');    const [error, setError] = useState<string | null>(null);    const navigate = useNavigate();    const { role: userRole } = useSelector((state: any) => state.user);    // Проверка роли пользователя    useEffect(() => {      if (userRole !== 'ADMIN') {        alert('Доступ запрещен: требуется роль ADMIN');        navigate('/catalog'); // Перенаправление на главную или другую страницу      }    }, [userRole, navigate]);    // Загрузка существующих данных    const fetchData = async () => {      try {        const [categoriesRes, criteriaRes] = await Promise.all([          fetch('http://localhost:3000/admin/categories'),          fetch('http://localhost:3000/admin/criteria'),        ]);        if (!categoriesRes.ok || !criteriaRes.ok) throw new Error('Ошибка загрузки данных');        setCategories(await categoriesRes.json());        setCriteria(await criteriaRes.json());      } catch (err) {        setError('Не удалось загрузить данные.');      }    };    useEffect(() => {      fetchData();    }, []);    // Добавление категории    const addCategory = async () => {      if (!newCategory.trim()) return;      try {        const res = await fetch('http://localhost:3000/admin/categories', {          method: 'POST',          headers: { 'Content-Type': 'application/json' },          body: JSON.stringify({ name: newCategory }),        });        if (!res.ok) throw new Error('Ошибка добавления категории');        const addedCategory = await res.json();        setCategories((prev) => [...prev, addedCategory]);        setNewCategory('');      } catch {        setError('Не удалось добавить категорию.');      }    };    // Добавление критерия    const addCriterion = async () => {      if (!newCriterion.trim()) return;      try {        const res = await fetch('http://localhost:3000/admin/criteria', {          method: 'POST',          headers: { 'Content-Type': 'application/json' },          body: JSON.stringify({ name: newCriterion }),        });        if (!res.ok) throw new Error('Ошибка добавления критерия');        const addedCriterion = await res.json();        setCriteria((prev) => [...prev, addedCriterion]);        setNewCriterion('');      } catch {        setError('Не удалось добавить критерий.');      }    };    // Удаление категории или критерия    const deleteItem = async (id: number, type: 'category' | 'criterion') => {      try {        const endpoint = type === 'category' ? 'categories' : 'criteria';        const res = await fetch(`http://localhost:3000/admin/${endpoint}/${id}`, { method: 'DELETE' });        if (!res.ok) throw new Error('Ошибка удаления');        if (type === 'category') setCategories((prev) => prev.filter((c) => c.id !== id));        if (type === 'criterion') setCriteria((prev) => prev.filter((c) => c.id !== id));      } catch {        setError('Не удалось удалить элемент.');      }    };    return (      <div className="admin-panel">        <h1>Панель администратора</h1>        {error && <p className="error">{error}</p>}        <section>          <h2>Категории</h2>          <ul>            {categories.map((category) => (              <li key={category.id}>                {category.name}{' '}                <button onClick={() => deleteItem(category.id, 'category')}>Удалить</button>              </li>            ))}          </ul>          <input            value={newCategory}            onChange={(e) => setNewCategory(e.target.value)}            placeholder="Новая категория"          />          <button onClick={addCategory}>Добавить категорию</button>        </section>        <section>          <h2>Критерии</h2>          <ul>            {criteria.map((criterion) => (              <li key={criterion.id}>                {criterion.name}{' '}                <button onClick={() => deleteItem(criterion.id, 'criterion')}>Удалить</button>              </li>            ))}          </ul>          <input            value={newCriterion}            onChange={(e) => setNewCriterion(e.target.value)}            placeholder="Новый критерий"          />          <button onClick={addCriterion}>Добавить критерий</button>        </section>          {/\* Вставляем компонент UserManagement \*/}        <section>          <h2>Управление пользователями</h2>          <UserManagement /> {/\* Вставка компонента \*/}        </section>        <section>          <h2>Статистика для администратора</h2>          <AdminStats />        </section>      </div>    );  };  export default AdminPanel;  import React, { useEffect, useState } from 'react';  import '../styles/AdminPanel.css'; // Подключаем стили  const AdminStats: React.FC = () => {    const [userCount, setUserCount] = useState<number>(0);    const [propertyCount, setPropertyCount] = useState<number>(0);    const [categoryStats, setCategoryStats] = useState<{ name: string; propertyCount: number }[]>([]);    const [loading, setLoading] = useState<boolean>(true);    const [error, setError] = useState<string | null>(null);    useEffect(() => {      // Функция для получения статистики      const fetchStats = async () => {        try {          const response = await fetch('http://localhost:3000/admin/stats'); // Путь к вашему API для статистики          // Проверяем, если статус ответа не OK          if (!response.ok) {            throw new Error('Не удалось загрузить статистику');          }          const data = await response.json();          setUserCount(data.userCount);          setPropertyCount(data.propertyCount);          setCategoryStats(data.categories); // Заполняем статистику по категориям        } catch (err) {          setError('Не удалось загрузить статистику');        } finally {          setLoading(false);        }      };      fetchStats();    }, []);    if (loading) {      return <div className="admin-panel"><div>Загрузка статистики...</div></div>;    }    if (error) {      return <div className="admin-panel"><div>{error}</div></div>;    }    return (      <div className="admin-panel">        <div>          <h3>Количество пользователей: {userCount}</h3>          <h3>Количество жилья: {propertyCount}</h3>        </div>        <h3>Статистика по категориям:</h3>        <ul>          {categoryStats.map((category) => (            <li key={category.name}>              {category.name}: {category.propertyCount} объектов            </li>          ))}        </ul>      </div>    );  };  export default AdminStats;  import React, { useState } from 'react';  interface BookingFormProps {    propertyId: number;    userId: number;    onBookingSuccess: (message: string) => void;  }  const BookingForm: React.FC<BookingFormProps> = ({ propertyId, userId, onBookingSuccess }) => {    const [startDate, setStartDate] = useState('');    const [endDate, setEndDate] = useState('');    const [message, setMessage] = useState(''); // Оставим это, чтобы отображать сообщение на экране    const handleBooking = async () => {      if (!startDate || !endDate) {        const errorMessage = 'Пожалуйста, выберите даты.';        setMessage(errorMessage); // Сохраняем сообщение об ошибке        onBookingSuccess(errorMessage); // Передаем сообщение через onBookingSuccess        return;      }      const bookingData = {        propertyId,        userId,        startDate,        endDate,      };      try {        const response = await fetch('http://localhost:3000/property/booking', {          method: 'POST',          headers: {            'Content-Type': 'application/json',          },          body: JSON.stringify(bookingData),        });        if (response.ok) {          const successMessage = 'Бронирование успешно создано!';          setMessage(successMessage); // Сохраняем сообщение об успехе          onBookingSuccess(successMessage); // Передаем сообщение через onBookingSuccess        } else {          const data = await response.json();          const errorMessage = data.message || 'Ошибка при создании бронирования';          setMessage(errorMessage); // Сохраняем сообщение об ошибке          onBookingSuccess(errorMessage); // Передаем сообщение через onBookingSuccess        }      } catch (error) {        console.error('Ошибка при создании бронирования:', error);        const errorMessage = 'Ошибка при создании бронирования';        setMessage(errorMessage); // Сохраняем сообщение об ошибке        onBookingSuccess(errorMessage); // Передаем сообщение через onBookingSuccess      }    };    return (      <div>        <h2>Забронировать жилье</h2>        <input          type="date"          value={startDate}          onChange={(e) => setStartDate(e.target.value)}          min={new Date().toISOString().split('T')[0]} // Ограничение на сегодняшнюю дату        />        <input          type="date"          value={endDate}          onChange={(e) => setEndDate(e.target.value)}          min={startDate}        />        <button onClick={handleBooking}>Забронировать</button>        {message && <p>{message}</p>} {/\* Отображение сообщения о статусе \*/}      </div>    );  };  export default BookingForm;  import React, { useEffect, useState, useCallback } from 'react';  import { useSelector } from 'react-redux';  import '../styles/BookingHistory.css';  interface Property {    id: number;    name: string;    description: string;    location: string;    pricePerNight: number;    imageUrl: string;  }  interface Booking {    id: number;    startDate: string;    endDate: string;    status: string;    property: Property;  }  const BookingHistory: React.FC = () => {    const [bookings, setBookings] = useState<Booking[]>([]);    const [error, setError] = useState<string | null>(null);    const userId = useSelector((state: any) => state.user.userId);    const statusTranslations: Record<string, string> = {      PENDING: 'В ожидании',      CONFIRMED: 'Подтверждено',      CANCELLED: 'Отменено',    };    const fetchBookingHistory = useCallback(async () => {      try {        const response = await fetch(`http://localhost:3000/property/user-history/${userId}`);        if (!response.ok) {          throw new Error('Не удалось загрузить историю бронирований');        }        const data = await response.json();        setBookings(data);        setError(null);      } catch (error) {        setError('Ошибка при загрузке данных');        console.error('Ошибка при загрузке данных:', error);      }    }, [userId]);    useEffect(() => {      fetchBookingHistory();      const intervalId = setInterval(() => {        fetchBookingHistory();      }, 1000);      return () => clearInterval(intervalId);    }, [fetchBookingHistory]);    if (error) {      return <div className="error-message">{error}</div>;    }    return (      <div className="container">        <h2 className="title">История бронирований</h2>        {bookings.length === 0 ? (          <p className="no-bookings">Нет истории бронирований</p>        ) : (          <ul className="booking-list">            {bookings.map((booking) => {              const startDate = new Date(booking.startDate).toLocaleDateString('ru-RU');              const endDate = new Date(booking.endDate).toLocaleDateString('ru-RU');              return (                <li key={booking.id} className="booking-item">                  <h3>{booking.property.name}</h3>                  <p>                    {startDate} - {endDate} ({statusTranslations[booking.status] || booking.status})                  </p>                  <p>Цена за ночь: {booking.property.pricePerNight} руб.</p>                  <p>{booking.property.description}</p>                </li>              );            })}          </ul>        )}      </div>    );  };  export default BookingHistory;  import React, { useEffect, useState } from 'react';  import { useSelector } from 'react-redux';  import { useNavigate } from 'react-router-dom';  import '../styles/Catalog.css';  interface Property {    id: number;    name: string;    description: string;    location: string;    pricePerNight: number;    imageUrl: string;    owner: {      login: string;      email: string;      id: number;    };    bookings: {      startDate: string;      endDate: string;      status: string;    }[];    category: {      name: string;    };  }  interface Category {    name: string;  }  const Catalog: React.FC = () => {    const [properties, setProperties] = useState<Property[]>([]);    const [categories, setCategories] = useState<Category[]>([]); // Состояние для категорий    const [loading, setLoading] = useState(true);    const [sortType, setSortType] = useState<string>('price');    const [filterCategory, setFilterCategory] = useState<string>('');    const [searchQuery, setSearchQuery] = useState<string>('');    const userRole = useSelector((state: any) => state.user.role);    const userId = useSelector((state: any) => state.user.userId);    const navigate = useNavigate();    useEffect(() => {      const fetchCategories = async () => {        try {          const response = await fetch('http://localhost:3000/admin/categories');          const data = await response.json();          setCategories(data); // Сохраняем категории        } catch (error) {          console.error('Ошибка при загрузке категорий:', error);        }      };      const fetchCatalog = async () => {        try {          let url = 'http://localhost:3000/catalog';          if (userId) {            url = `http://localhost:3000/catalog?userRole=${userRole}&userId=${userId}`;          }          const response = await fetch(url);          const data = await response.json();          if (Array.isArray(data)) {            setProperties(data);          } else {            console.error('Полученные данные не являются массивом:', data);            setProperties([]);          }        } catch (error) {          console.error('Ошибка при получении каталога:', error);        } finally {          setLoading(false);        }      };      fetchCategories();      fetchCatalog();    }, [userRole, userId]);    const handleDelete = async (id: number) => {      if (userRole !== 'OWNER' && userRole !== 'ADMIN') {        alert('Вы должны быть владельцем жилья для удаления!');        return;      }      if (window.confirm('Вы уверены, что хотите удалить это жилье?')) {        try {          const response = await fetch(`http://localhost:3000/catalog/${id}`, {            method: 'DELETE',            headers: {              'Content-Type': 'application/json',              'Authorization': `Bearer ${localStorage.getItem('token')}`,            },          });          if (response.ok) {            setProperties(properties.filter((property) => property.id !== id));            alert('Жилье успешно удалено');          } else {            const errorData = await response.json();            console.error('Ошибка при удалении жилья:', errorData);            alert('Ошибка при удалении жилья');          }        } catch (error) {          console.error('Ошибка при удалении жилья:', error);          alert('Ошибка при удалении жилья');        }      }    };    const handleAddHousing = () => {      navigate('/create');    };    const handlePropertyClick = (id: number) => {      navigate(`/property/${id}`);    };    const formatDate = (dateString: string) => {      const date = new Date(dateString);      const day = String(date.getDate()).padStart(2, '0');      const month = String(date.getMonth() + 1).padStart(2, '0');      const year = date.getFullYear();      return `${day}.${month}.${year}`;    };    const isBookingActive = (booking: { startDate: string, endDate: string }) => {      const currentDate = new Date();      const endDate = new Date(booking.endDate);      return endDate > currentDate;    };    const sortedProperties = [...properties]      .sort((a, b) => {        if (sortType === 'price') {          return a.pricePerNight - b.pricePerNight;        }        return a.name.localeCompare(b.name);      })      .filter((property) => {        if (filterCategory && property.category.name !== filterCategory) {          return false;        }        if (          property.name.toLowerCase().includes(searchQuery.toLowerCase())          ) {          return true;        }        return false;      });    if (loading) return <div>Загрузка...</div>;    return (      <div className="catalog-container">        <h1 className="catalog-header">Каталог жилья</h1>        {(userRole === 'OWNER') && (          <button onClick={handleAddHousing} className="add-housing-button">            Добавить жилье          </button>        )}  <div className="filters-container">    <div className="search-container">      <input        type="text"        placeholder="Поиск по названию или описанию"        value={searchQuery}        onChange={(e) => setSearchQuery(e.target.value)}        className="search-input"      />    </div>    <div className="filter-container">      <label>Фильтровать по категории: </label>      <select onChange={(e) => setFilterCategory(e.target.value)} className="filter-select">        <option value="">Все</option>        {categories.map((category, index) => (          <option key={index} value={category.name}>{category.name}</option>        ))}      </select>    </div>    <div className="sort-container">      <label>Сортировка: </label>      <select onChange={(e) => setSortType(e.target.value)} className="sort-select">        <option value="price">По цене</option>        <option value="alphabet">По алфавиту</option>      </select>    </div>  </div>        {sortedProperties.length === 0 ? (          <p>Жилье не найдено</p>        ) : (          <ul className="property-list">            {sortedProperties.map((property) => {              const latestConfirmedBooking = property.bookings                .filter((booking) => booking.status === 'CONFIRMED')                .sort((a, b) => new Date(b.endDate).getTime() - new Date(a.endDate).getTime())[0];              return (                <li                  key={property.id}                  className="property-card"                  onClick={() => handlePropertyClick(property.id)}                >                  <img                    src={`http://localhost:3000${property.imageUrl}`}                    alt={property.name}                    className="property-image"                  />                  <h2 className="property-name">{property.name}</h2>                  <p className="property-description">{property.description}</p>                  <p className="property-location">Местоположение: {property.location}</p>                  <p className="property-price">Цена за ночь: {property.pricePerNight} руб.</p>                  <p className="property-category">Категория: {property.category.name}</p>                  <div className="owner-info">                    Владелец: {property.owner.login} ({property.owner.email})                  </div>                  <div>                    <h4 className="bookings-header">Последнее бронирование:</h4>                    {latestConfirmedBooking ? (                      <ul>                        <li className={`booking-item ${isBookingActive(latestConfirmedBooking) ? 'active-booking' : 'free-booking'}`}>                          {formatDate(latestConfirmedBooking.startDate)} - {formatDate(latestConfirmedBooking.endDate)}                        </li>                      </ul>                    ) : (                      <p className="free-booking">Нет подтвержденных бронирований</p>                    )}                  </div>                  {(userRole === 'ADMIN' || String(userId) === String(property.owner.id)) && (                    <button                      className="delete-button"                      onClick={(e) => {                        e.stopPropagation();                        handleDelete(property.id);                      }}                    >                      Удалить жилье                    </button>                  )}                </li>              );            })}          </ul>        )}      </div>    );  };  export default Catalog;  import React, { useState, useEffect } from 'react';  import '../styles/ChatComponent.css'  import { useSelector } from 'react-redux';  interface Chat {    id: number;    ownerId: number;    userId: number;    propertyId: number;    messages: Message[];    user: {      id: number;      role: string;      login: string;      email: string;    };    owner: {      id: number;      role: string;      login: string;      email: string;    };    property: {      ownerId: number;    };  }  interface Message {    id: number;    chatId: number;    senderId: number;    content: string;    createdAt: string;  }  const ChatComponent: React.FC<{ ownerId: number; userRole: string; propertyId: number; ownerName: string }> = ({ ownerId, userRole, propertyId, ownerName }) => {    const [chats, setChats] = useState<Chat[]>([]);    const [currentChat, setCurrentChat] = useState<Chat | null>(null);    const [chatExists, setChatExists] = useState(false);    const [message, setMessage] = useState('');    const [loading, setLoading] = useState(false);    const userName = useSelector((state: any) => state.user.userName);    const userId = useSelector((state: any) => state.user.userId);    const [error, setError] = useState<string | null>(null);    // Получение списка чатов по propertyId    useEffect(() => {      const fetchChats = async () => {        setLoading(true);        try {          const response = await fetch(`http://localhost:3000/api/chats/${userId}/${propertyId}`);          if (!response.ok) {            throw new Error(`Error fetching chats: ${response.statusText}`);          }          const data: Chat[] = await response.json();          console.log(data)          setChats(data || []);          const existingChat = data.find(            (chat) =>              (Number(chat.ownerId) === Number(ownerId) && Number(chat.userId) === Number(userId) && chat.propertyId === propertyId) ||              (Number(chat.ownerId) === Number(userId) && Number(chat.userId) === Number(ownerId) && chat.propertyId === propertyId)          );          setChatExists(!!existingChat);          // Для владельца открыть первый существующий чат, если доступен          if (existingChat) {            setCurrentChat(existingChat);          } else if (userRole === 'OWNER' && data.length > 0) {            setCurrentChat(data[0]);          }        } catch (err) {          setError((err as Error).message);        } finally {          setLoading(false);        }      };      fetchChats();    }, [userId, ownerId, propertyId, userRole]);    // Создание нового чата    const createChat = async () => {      setLoading(true);      setError(null);      try {        const response = await fetch('http://localhost:3000/api/chats', {          method: 'POST',          headers: {            'Content-Type': 'application/json',          },          body: JSON.stringify({            ownerId,            userId,            propertyId,          }),        });        if (!response.ok) {          throw new Error(`Error creating chat: ${response.statusText}`);        }        const newChat = await response.json();          // Добавление нового чата в список чатов        setChats((prevChats) => [...prevChats, newChat]);          // Переключаемся на новый чат сразу        setCurrentChat(newChat);          // Устанавливаем, что чат существует        setChatExists(true);      } catch (err) {        setError((err as Error).message);      } finally {        setLoading(false);      }    };      // Обновление сообщений текущего чата    const fetchMessages = async (chatId: number) => {      try {        const response = await fetch(`http://localhost:3000/api/messages/${chatId}`);        if (!response.ok) {          throw new Error(`Error fetching messages: ${response.statusText}`);        }        const messages: Message[] = await response.json();        setCurrentChat((prevChat) => {          if (prevChat) {            return { ...prevChat, messages };          }          return prevChat;        });      } catch (err) {        setError((err as Error).message);      }    };    // Счётчик для обновления сообщений    useEffect(() => {      if (currentChat) {        const intervalId = setInterval(() => {          fetchMessages(currentChat.id);        }, 1000);        return () => clearInterval(intervalId); // Очистка интервала при размонтировании      }    }, [currentChat]);    // Отправка сообщения    const sendMessage = () => {      if (currentChat && message.trim()) {        setError(null);        fetch('http://localhost:3000/api/messages', {          method: 'POST',          headers: {            'Content-Type': 'application/json',          },          body: JSON.stringify({            chatId: currentChat.id,            senderId: userId,            content: message,          }),        })          .then((response) => {            if (!response.ok) {              throw new Error(`Error sending message: ${response.statusText}`);            }            return response.json();          })          .then((newMessage) => {            // Обновляем чат, убедившись, что сообщения всегда массив            setCurrentChat((prevChat) => {              if (prevChat) {                return {                  ...prevChat,                  messages: Array.isArray(prevChat.messages) ? [...prevChat.messages, newMessage] : [newMessage],                };              }              return prevChat; // Если чата нет, возвращаем предыдущий стейт            });            setMessage('');          })          .catch((err) => setError((err as Error).message));      }    };    return (      <div className="chat-component">        <h1 className="chat-component\_\_title">Чаты</h1>        {error && <p className="chat-component\_\_error">Error: {error}</p>}        <div>          {loading ? (            <p>Loading chats...</p>          ) : (            <ul className="chat-component\_\_list">              {chats.length > 0 ? (                chats.map((chat) => (                  <li key={chat.id} onClick={() => setCurrentChat(chat)} className="chat-component\_\_item">                    Чат с{' '}                    {chat.owner?.login && Number(chat.userId) === Number(userId)                      ? `владельцем ${chat.owner.login}`                      : chat.user?.login && `пользователем ${chat.user.login}`}                  </li>                ))              ) : (                userRole === 'USER' && <p>Нет доступных чатов. Начни новый чат!</p>              )}            </ul>          )}        </div>        {!currentChat && userRole !== 'OWNER' && !chatExists && Number(ownerId) !== Number(userId) && (          <div>            <button              onClick={createChat}              disabled={loading}              className="chat-component\_\_button"            >              {loading ? 'Creating chat...' : 'Start a chat'}            </button>          </div>        )}        {currentChat && (          <div>            <h2 className="chat-component\_\_title">Чат</h2>            <div className="chat-component\_\_messages">              {currentChat?.messages?.length > 0 ? (                currentChat.messages.map((msg) => (                  <p key={msg.id} className="chat-component\_\_message">                    <b>                      {Number(msg.senderId) === Number(userId)                        ? 'Вы'                        : Number(msg.senderId) === Number(currentChat.owner?.id)                        ? currentChat.owner?.login                        : currentChat.user?.login}                    </b>{' '}                    {msg.content}                  </p>                ))              ) : (                <p>Сообщений пока нет. Начни диалог!</p>              )}            </div>            <input              value={message}              onChange={(e) => setMessage(e.target.value)}              placeholder="Type your message..."              className="chat-component\_\_input"            />            <button              onClick={sendMessage}              className="chat-component\_\_button"            >              Отправить            </button>          </div>        )}      </div>    );  };  export default ChatComponent;  import React, { useState, useEffect } from 'react';  import { useSelector } from 'react-redux';  import { useNavigate } from 'react-router-dom';  import '../styles/UserChatsComponent.css';  // Подключаем CSS  interface Chat {    id: number;    ownerId: number;    userId: number;    propertyId: number;    messages: Message[];    user: {      id: number;      role: string;      login: string;      email: string;    };    owner: {      id: number;      role: string;      login: string;      email: string;    };    property: {      name: string;    }  }  interface Message {    id: number;    chatId: number;    senderId: number;    content: string;    createdAt: string;  }  const UserChatsComponent: React.FC = () => {    const [chats, setChats] = useState<Chat[]>([]);    const [loading, setLoading] = useState(false);    const [error, setError] = useState<string | null>(null);    const userId = useSelector((state: any) => state.user.userId);    const navigate = useNavigate();    // Получение списка чатов    useEffect(() => {      const fetchChats = async () => {        setLoading(true);        setError(null);        try {          const response = await fetch(`http://localhost:3000/api/chats/${userId}`);          if (!response.ok) {            throw new Error(`Error fetching chats: ${response.statusText}`);          }          const data: Chat[] = await response.json();          setChats(data || []);        } catch (err) {          setError((err as Error).message);        } finally {          setLoading(false);        }      };      fetchChats();    }, [userId]);    // Переход к жилью и продолжение чата    const handleChatClick = (chat: Chat) => {      navigate(`/property/${chat.propertyId}`, { state: { chatId: chat.id } });    };    return (      <div className="container">        <h1 className="header">Ваши чаты</h1>        {/\* Отображение ошибки \*/}        {error && <p className="error">Error: {error}</p>}        {/\* Список чатов \*/}        <div>          {loading ? (            <p className="loading">Загрузка чатов...</p>          ) : (            <ul className="chatList">              {chats.length > 0 ? (                chats.map((chat) => (                  <li key={chat.id} className="chatItem" onClick={() => handleChatClick(chat)}>                    <span className="chatText">                      Чат с {chat.owner?.login && Number(chat.userId) === Number(userId)                        ? chat.owner.login                        : chat.user?.login} о жилье "{chat.property.name}"                    </span>                    <button className="button">Перейти к жилью</button>                  </li>                ))              ) : (                <p className="noChats">Нет доступных чатов</p>              )}            </ul>          )}        </div>      </div>    );  };  export default UserChatsComponent;  import React, { useState, useEffect } from 'react';  import { useSelector } from 'react-redux';  // Для доступа к данным Redux  import '../styles/CreateHousing.css';  // Импортируем стили  const CreateHousing: React.FC = () => {    const [formData, setFormData] = useState({      name: '',      description: '',      location: '',      pricePerNight: '',      categoryId: '', // Новый параметр для категории      criteria: [] as string[], // Новый параметр для критериев (массив)    });    const [image, setImage] = useState<File | null>(null); // для хранения выбранного изображения    const [categories, setCategories] = useState<any[]>([]); // Состояние для списка категорий    const [criteriaList, setCriteriaList] = useState<any[]>([]); // Состояние для списка критериев    // Получаем данные пользователя из Redux    const user = useSelector((state: any) => state.user);    // Получаем ID владельца из данных пользователя    const ownerId = user.userId;    // Проверяем, является ли пользователь администратором или владельцем    const canCreateHousing = user.role === 'ADMIN' || user.role === 'OWNER';    // Загрузка категорий и критериев с сервера    useEffect(() => {      // Загрузка категорий      fetch('http://localhost:3000/admin/categories')        .then(response => response.json())        .then(data => setCategories(data))        .catch(error => console.error('Ошибка при загрузке категорий:', error));      // Загрузка критериев      fetch('http://localhost:3000/admin/criteria')        .then(response => response.json())        .then(data => setCriteriaList(data))        .catch(error => console.error('Ошибка при загрузке критериев:', error));    }, []);    const handleChange = (e: React.ChangeEvent<HTMLInputElement | HTMLTextAreaElement>) => {      setFormData({        ...formData,        [e.target.name]: e.target.value,      });    };    const handleCategoryChange = (e: React.ChangeEvent<HTMLSelectElement>) => {      setFormData({        ...formData,        categoryId: e.target.value,      });    };    const handleCriteriaChange = (e: React.ChangeEvent<HTMLInputElement>) => {      const { value, checked } = e.target;        setFormData(prevFormData => {        const updatedCriteria = checked          ? [...prevFormData.criteria, value]  // Добавляем ID критерия в массив          : prevFormData.criteria.filter(criterionId => criterionId !== value); // Убираем ID критерия из массива          return {          ...prevFormData,          criteria: updatedCriteria,        };      });    };      const handleImageChange = (e: React.ChangeEvent<HTMLInputElement>) => {      if (e.target.files) {        setImage(e.target.files[0]); // сохраняем выбранный файл      }    };    const handleSubmit = async (e: React.FormEvent) => {      e.preventDefault();      const formDataToSend = new FormData();      formDataToSend.append('name', formData.name);      formDataToSend.append('description', formData.description);      formDataToSend.append('location', formData.location);      formDataToSend.append('pricePerNight', formData.pricePerNight);      formDataToSend.append('ownerId', ownerId.toString()); // Используем ID владельца из контекста      formDataToSend.append('categoryId', formData.categoryId); // Добавляем categoryId      if (formData.criteria.length > 0) {        formData.criteria.forEach(criterionId => {          formDataToSend.append('criteria[]', criterionId); // Добавляем каждый ID критерия        });      }      if (image) {        formDataToSend.append('image', image); // добавляем изображение      }        formDataToSend.forEach((value, key) => {        console.log(key, value); // выводим ключ и значение      });      try {        const response = await fetch('http://localhost:3000/housing', {          method: 'POST',          body: formDataToSend,        });        if (response.ok) {          alert('Жилье успешно создано!');          const newProperty = await response.json();          console.log(newProperty.imageUrl);          setFormData({            name: '',            description: '',            location: '',            pricePerNight: '',            categoryId: '',            criteria: [],          });          setImage(null);        } else {          alert('Ошибка при создании жилья');        }      } catch (error) {        console.error('Ошибка:', error);        alert('Ошибка при создании жилья');      }    };    if (!canCreateHousing) {      return <p className="error-message">У вас нет прав для создания жилья. Только администратор или владелец может создать жилье.</p>;    }    return (      <div className="create-housing-container">        <h2 className="create-housing-header">Создать новое жилье</h2>        <form onSubmit={handleSubmit} className="create-housing-form">          <input            name="name"            placeholder="Название"            value={formData.name}            onChange={handleChange}            required          />          <textarea            name="description"            placeholder="Описание"            value={formData.description}            onChange={handleChange}            required          />          <input            name="location"            placeholder="Местоположение"            value={formData.location}            onChange={handleChange}            required          />          <input            name="pricePerNight"            type="number"            placeholder="Цена за ночь"            value={formData.pricePerNight}            onChange={handleChange}            required          />          <select            name="categoryId"            value={formData.categoryId}            onChange={handleCategoryChange}            required          >            <option value="">Выберите категорию</option>            {categories.map(category => (              <option key={category.id} value={category.id}>                {category.name}              </option>            ))}          </select>          <div>              <p>Выберите критерии:</p>              {criteriaList.map(criterion => (                <label key={criterion.id}>                  <input                    type="checkbox"                    value={criterion.id.toString()} // Преобразуем в строку, если id числовой                    checked={formData.criteria.includes(criterion.id.toString())} // Убедитесь, что это строка                    onChange={handleCriteriaChange}                  />                  {criterion.name}                </label>              ))}            </div>          <input            type="file"            accept="image/\*"            onChange={handleImageChange}            required          />          <button type="submit">Создать</button>        </form>      </div>    );  };  export default CreateHousing;  import React, { useState, useEffect } from 'react';  import { useParams, useNavigate } from 'react-router-dom';  import '../styles/EditProperty.css';  interface Property {    id: number;    name: string;    description: string;    location: string;    pricePerNight: number;    imageUrl: string;    category: { id: number, name: string };    criteria: { id: number, name: string }[];  }  const EditProperty: React.FC = () => {    const { id } = useParams();    const navigate = useNavigate();    const [property, setProperty] = useState<Property | null>(null);    const [name, setName] = useState('');    const [description, setDescription] = useState('');    const [location, setLocation] = useState('');    const [pricePerNight, setPricePerNight] = useState('');    const [image, setImage] = useState<File | null>(null);    const [categories, setCategories] = useState<{ id: number, name: string }[]>([]);    const [selectedCategory, setSelectedCategory] = useState<number | null>(null);    const [selectedCriteria, setSelectedCriteria] = useState<number[]>([]);    const [criteriaList, setCriteriaList] = useState<{ id: number, name: string }[]>([]);    const [message, setMessage] = useState('');    useEffect(() => {      const fetchProperty = async () => {        try {          const response = await fetch(`http://localhost:3000/property/${id}`);          const data = await response.json();          setProperty(data);          setName(data.name);          setDescription(data.description);          setLocation(data.location);          setPricePerNight(data.pricePerNight.toString());          setSelectedCategory(data.category.id);          setSelectedCriteria(data.criteria.map((criterion: { id: number }) => criterion.id));          setImage(null); // Сбросим изображение после загрузки        } catch (error) {          console.error('Ошибка при загрузке данных жилья:', error);        }      };      const fetchCategories = async () => {        try {          const response = await fetch('http://localhost:3000/admin/categories');          const data = await response.json();          setCategories(data);        } catch (error) {          console.error('Ошибка при загрузке категорий:', error);        }      };      const fetchCriteria = async () => {        try {          const response = await fetch('http://localhost:3000/admin/criteria');          const data = await response.json();          setCriteriaList(data);        } catch (error) {          console.error('Ошибка при загрузке критериев:', error);        }      };      fetchProperty();      fetchCategories();      fetchCriteria();    }, [id]);    const handleSubmit = async (e: React.FormEvent) => {      e.preventDefault();      const formData = new FormData();      formData.append('name', name);      formData.append('description', description);      formData.append('location', location);      formData.append('pricePerNight', pricePerNight);      formData.append('categoryId', String(selectedCategory));      formData.append('criteria', JSON.stringify(selectedCriteria));      // Добавляем изображение (если оно выбрано или если старое изображение доступно)      if (image instanceof File) {        formData.append('image', image); // Добавляем новый файл      } else if (image && typeof image === 'string') {        formData.append('image', image); // Отправляем старое изображение, если оно есть      }      try {        const response = await fetch(`http://localhost:3000/property/${id}`, {          method: 'PUT',          headers: {            'Authorization': `Bearer ${localStorage.getItem('token')}`,          },          body: formData,        });        formData.forEach((value, key) => {          if (value instanceof File) {            console.log(`${key}:`, value.name);          } else {            console.log(`${key}:`, value);          }        });        if (response.ok) {          setMessage('Жилье успешно обновлено!');          navigate(`/property/${id}`);        } else {          const data = await response.json();          setMessage(data.message || 'Ошибка при обновлении жилья');        }      } catch (error) {        console.error('Ошибка при отправке данных:', error);        setMessage('Ошибка при обновлении жилья');      }    };    if (!property || !categories.length || !criteriaList.length) {      return <div>Загрузка...</div>;    }    return (      <div className="edit-property">        <h2>Редактировать жилье</h2>        <form onSubmit={handleSubmit} className="property-form" encType="multipart/form-data">          <div className="form-group">            <label htmlFor="name">Название</label>            <input              id="name"              type="text"              value={name}              onChange={(e) => setName(e.target.value)}              required              className="form-control"            />          </div>          <div className="form-group">            <label htmlFor="description">Описание</label>            <textarea              id="description"              value={description}              onChange={(e) => setDescription(e.target.value)}              required              className="form-control"            />          </div>          <div className="form-group">            <label htmlFor="location">Местоположение</label>            <input              id="location"              type="text"              value={location}              onChange={(e) => setLocation(e.target.value)}              required              className="form-control"            />          </div>          <div className="form-group">            <label htmlFor="pricePerNight">Цена за ночь</label>            <input              id="pricePerNight"              type="number"              value={pricePerNight}              onChange={(e) => setPricePerNight(e.target.value)}              required              className="form-control"            />          </div>          <div className="form-group">            <label htmlFor="category">Категория</label>            <select              id="category"              value={selectedCategory ?? ''}              onChange={(e) => setSelectedCategory(Number(e.target.value))}              required              className="form-control"            >              <option value="">Выберите категорию</option>              {categories.map((category) => (                <option key={category.id} value={category.id}>                  {category.name}                </option>              ))}            </select>          </div>          <div className="form-group">            <label>Критерии</label>            <div className="criteria-list">              {criteriaList.map((criterion) => (                <div key={criterion.id} className="form-check">                  <input                    type="checkbox"                    id={`criterion-${criterion.id}`}                    checked={selectedCriteria.includes(criterion.id)}                    onChange={() => {                      if (selectedCriteria.includes(criterion.id)) {                        setSelectedCriteria(selectedCriteria.filter((id) => id !== criterion.id));                      } else {                        setSelectedCriteria([...selectedCriteria, criterion.id]);                      }                    }}                    className="form-check-input"                  />                  <label htmlFor={`criterion-${criterion.id}`} className="form-check-label">                    {criterion.name}                  </label>                </div>              ))}            </div>          </div>          <div className="form-group">            <label htmlFor="image">Изображение</label>            {property?.imageUrl && !image && (              <div>                <img src={`http://localhost:3000${property.imageUrl}`} alt="Текущее изображение" width="100" />                <p>Изображение не изменено</p>              </div>            )}            <input              id="image"              type="file"              accept="image/\*"              onChange={(e) => setImage(e.target.files ? e.target.files[0] :null)}              className="form-control"            />          </div>          <button type="submit" className="btn-submit">Сохранить</button>        </form>        {message && <p className="message">{message}</p>}      </div>    );  };  export default EditProperty;  import React, { useState, FormEvent } from 'react';  import { useDispatch } from 'react-redux';  import { setUserRole, setUserName, setUserId, setUserActiveStatus } from '../slices/userSlice';  import { decodeJwt } from 'jose';  import '../styles/Login.css';  const Login: React.FC = () => {    const dispatch = useDispatch();    const [login, setLogin] = useState<string>('');    const [password, setPassword] = useState<string>('');    const [message, setMessage] = useState<string>('');    const handleSubmit = async (e: FormEvent) => {      e.preventDefault();      try {        const response = await fetch('http://localhost:3000/auth/login', {          method: 'POST',          headers: { 'Content-Type': 'application/json' },          body: JSON.stringify({ login, password }),        });        const data = await response.json();        if (response.ok && data.token) {          localStorage.setItem('token', data.token);          // Декодируем токен          const decodedToken: any = decodeJwt(data.token);          if (decodedToken) {            const { role, userId, isActive } = decodedToken;            // Сохраняем данные в localStorage и Redux            if (role) {              dispatch(setUserRole(role));            }            if (userId) {              dispatch(setUserId(userId));            }            if (isActive !== undefined) {              dispatch(setUserActiveStatus(isActive)); // Сохраняем статус активности            }          }          dispatch(setUserName(login)); // Обновляем имя в Redux          setMessage('Авторизация успешна');          // Перенаправление          if (decodedToken.isActive === false) {            setMessage('Ваш аккаунт заблокирован');          } else {            window.location.href = '/catalog';          }        } else {          setMessage(data.message || 'Ошибка при авторизации');        }      } catch (error) {        console.error('Ошибка при авторизации:', error);        setMessage('Ошибка при авторизации');      }    };    return (      <div className="login-container">        <h2 className="login-header">Вход</h2>        <form onSubmit={handleSubmit} className="login-form">          <input            type="text"            placeholder="Логин"            value={login}            onChange={(e) => setLogin(e.target.value)}            required            className="input-field"          />          <input            type="password"            placeholder="Пароль"            value={password}            onChange={(e) => setPassword(e.target.value)}            required            className="input-field"          />          <button type="submit" className="submit-button">Войти</button>        </form>        {message && <p className="message">{message}</p>}      </div>    );  };  export default Login;  import React, { useState, useEffect, useCallback } from 'react';  import { Link, useNavigate } from 'react-router-dom';  import { useDispatch, useSelector } from 'react-redux';  import { setUserRole, logout } from '../slices/userSlice';  import "../styles/Navbar.css";  const Navbar: React.FC = () => {    const dispatch = useDispatch();    const navigate = useNavigate();    // Данные пользователя из Redux    const { role: userRole, userName, userId, isActive } = useSelector((state: any) => state.user);    const [pendingBookings, setPendingBookings] = useState<number>(0); // Количество ожидающих бронирований    // Функция для получения количества ожидающих бронирований    const fetchPendingBookings = useCallback(async () => {      try {        if (userRole === 'OWNER') {          const response = await fetch(`http://localhost:3000/property/owner/${userId}`, {            method: 'GET',            headers: {              'Content-Type': 'application/json',              'Authorization': `Bearer ${localStorage.getItem('token')}`,            },          });          if (response.ok) {            const data = await response.json();            const pendingCount = data.flatMap((property: any) => property.bookings)              .filter((booking: any) => booking.status === 'PENDING')              .length;            setPendingBookings(pendingCount);          } else {            console.error('Ошибка при получении бронирований');          }        }      } catch (error) {        console.error('Ошибка при получении бронирований:', error);      }    }, [userRole, userId]);    // Выполнение запроса при изменении userId или userRole    useEffect(() => {      if (userRole === 'OWNER') {        fetchPendingBookings();        // Настроим обновление каждую секунду        const intervalId = setInterval(() => {          fetchPendingBookings();        }, 1000);        // Очищаем интервал при размонтировании компонента        return () => clearInterval(intervalId);      }    }, [fetchPendingBookings, userRole, userId]);    // Проверка блокировки пользователя    useEffect(() => {      if (isActive === false) {        alert('Ваш аккаунт заблокирован. Пожалуйста, свяжитесь с администратором.');        navigate('/login'); // Перенаправление на страницу входа      }    }, [isActive, navigate]);    // Общая функция для смены роли    const handleRoleSwitch = async (newRole: string, successMessage: string) => {      if (!userId) {        alert('Ошибка: не найден ID пользователя');        return;      }      try {        const response = await fetch(`http://localhost:3000/users/${userId}/role`, {          method: 'PUT',          headers: { 'Content-Type': 'application/json' },          body: JSON.stringify({ newRole }),        });        const data = await response.json();        if (response.ok) {          dispatch(setUserRole(newRole));          alert(successMessage);          navigate('/catalog');        } else {          alert(data.message || 'Ошибка при смене роли');        }      } catch (error) {        console.error('Ошибка при смене роли:', error);        alert('Ошибка при смене роли');      }    };    // Выход пользователя    const handleLogout = () => {      ['token', 'role', 'userName', 'userId', 'isActive'].forEach((key) => localStorage.removeItem(key));      dispatch(logout());      navigate('/catalog');    };    // Проверка на гостя    const isGuest = !userRole;    return (      <nav className="navbar">        <ul className="navbar-list">          <li><Link to="/catalog">Каталог</Link></li>          {isGuest ? (            <>              <li><Link to="/login">Войти</Link></li>              <li><Link to="/register">Зарегистрироваться</Link></li>            </>          ) : (            <>              <li>Привет, {userName}, ваша роль: {userRole}</li>              {userRole === 'USER' && isActive && (                <>                  <li>                    <button onClick={() => navigate(`/user-history/${userId}`)}>                      История бронирований                    </button>                  </li>                  <li>                    <button onClick={() => handleRoleSwitch('OWNER', 'Вы стали владельцем!')}>                      Стать владельцем                    </button>                  </li>                  <li>                    <button onClick={() => navigate(`/user-settings/${userId}`)}>                      Настройки пользователя                    </button>                  </li>                  {/\* Добавляем переход к чату \*/}                  <li>                    <button onClick={() => navigate(`/user-chats/${userId}`)}>                      Мои чаты                    </button>                  </li>                </>              )}              {userRole === 'OWNER' && isActive && (                <>                  {pendingBookings > 0 && (                    <li>                      <button onClick={() => navigate(`/owner/${userId}`)}>                        У вас есть {pendingBookings} новых запросов на бронирование                      </button>                    </li>                  )}                  <li>                    <button onClick={() => handleRoleSwitch('USER', 'Вы стали пользователем!')}>                      Стать пользователем                    </button>                  </li>                  {/\* Добавляем переход к чату \*/}                  <li>                    <button onClick={() => navigate(`/user-chats/${userId}`)}>                      Мои чаты                    </button>                  </li>                  {/\* Добавляем ссылку на статистику владельца \*/}                  <li>                    <button onClick={() => navigate('/statistics')}>                      Статистика владельца                    </button>                  </li>                </>              )}              {userRole === 'ADMIN' && (                <li>                  <button onClick={() => navigate('/admin')}>                    Админка                  </button>                </li>              )}              <li><button onClick={handleLogout}>Выйти</button></li>            </>          )}        </ul>      </nav>    );  };  export default Navbar;  import React, { useState, useEffect, useCallback } from 'react';  import { useSelector } from 'react-redux';  import { useNavigate } from 'react-router-dom';  import '../styles/OwnerBookingsPage.css';  interface Booking {    id: number;    startDate: string;    endDate: string;    status: string;    property: {      id: number;      name: string;      imageUrl: string;      description: string;      location: string;      pricePerNight: number;    };  }  const OwnerBookingsPage: React.FC = () => {    const [bookings, setBookings] = useState<Booking[]>([]);    const navigate = useNavigate();    const userId = useSelector((state: any) => state.user.userId);    const fetchBookings = useCallback(async () => {      try {        const response = await fetch(`http://localhost:3000/property/owner/${userId}`, {          method: 'GET',          headers: {            'Content-Type': 'application/json',            'Authorization': `Bearer ${localStorage.getItem('token')}`,          },        });        if (response.ok) {          const data = await response.json();          const allBookings = data.flatMap((property: any) =>            property.bookings.filter((booking: any) => booking.status === 'PENDING')              .map((booking: any) => ({                ...booking,                property: property              }))          );            setBookings(allBookings);        } else {          console.error('Ошибка при получении бронирований');        }      } catch (error) {        console.error('Ошибка при получении бронирований:', error);      }    }, [userId]);    useEffect(() => {      if (userId) {        fetchBookings();        const intervalId = setInterval(() => {          fetchBookings();        }, 1000);        return () => clearInterval(intervalId);      }    }, [fetchBookings, userId]);    const goToPropertyPage = (propertyId: number) => {      navigate(`/property/${propertyId}`);    };    if (bookings.length === 0) {      return <div className="message">Нет бронирований для обработки</div>;    }    return (      <div className="container">        <h1 className="title">Запросы на бронирование</h1>        <ul className="list">          {bookings.map((booking) => {            const property = booking.property;            const propertyName = property?.name || 'Без имени';            const startDate = new Date(booking.startDate).toLocaleDateString('ru-RU');            const endDate = new Date(booking.endDate).toLocaleDateString('ru-RU');            return (              <li key={booking.id} className="list-item">                <div className="booking-details">                  <strong>Имя жилья: {propertyName}</strong>                  <p>Даты: {startDate} - {endDate} (ОЖИДАНИЕ)</p>                </div>                <button className="go-to-button" onClick={() => goToPropertyPage(property?.id || 0)}>                  Перейти к жилью                </button>              </li>            );          })}        </ul>      </div>    );  };  export default OwnerBookingsPage;  import React, { useEffect, useState } from 'react';  import { useSelector } from 'react-redux';  import '../styles/OwnerStatistics.css'; // Подключаем стили  interface Statistics {    housingCount: number;    bookingsCount: number;    completedBookingsCount: number;  }  const OwnerStatistics: React.FC = () => {    const [statistics, setStatistics] = useState<Statistics | null>(null);    const userId = useSelector((state: any) => state.user.userId);    useEffect(() => {      const fetchStatistics = async () => {        try {          const response = await fetch(`http://localhost:3000/stat/statistics/${userId}`);          if (response.ok) {            const data = await response.json();            setStatistics(data);          } else {            console.error('Не удалось получить статистику');          }        } catch (error) {          console.error('Ошибка при получении статистики:', error);        }      };      fetchStatistics();    }, [userId]);    if (!statistics) {      return <div className="loading">Загрузка...</div>;    }    return (      <div className="owner-statistics">        <h2>Статистика владельца</h2>        <ul>          <li><strong>Количество жилья:</strong> {statistics.housingCount}</li>          <li><strong>Количество бронирований:</strong> {statistics.bookingsCount}</li>          <li><strong>Количество подтвержденных бронирований:</strong> {statistics.completedBookingsCount}</li>        </ul>      </div>    );  };  export default OwnerStatistics;  import React, { useState, useEffect, useCallback } from 'react';  import { useParams, useNavigate } from 'react-router-dom';  import { useSelector } from 'react-redux';  import '../styles/PropertyPage.css';  import BookingForm from './BookingForm';  import Reviews from './Review'; // Импортируем компонент Reviews  import { useBookingActions } from './useBookingActions';  import ChatComponent from './ChatComponent';  interface Property {    id: number;    name: string;    description: string;    location: string;    pricePerNight: number;    imageUrl: string;    owner: {      id: number;      role: string;      login: string;      email: string;    };    category: {      name: string; // Категория жилья    };    criteria: {      name: string; // Название критерия    }[]; // Список критериев    bookings: {      id: number;      startDate: string;      endDate: string;      status: string;      userId: number;    }[];  }  const PropertyPage: React.FC = () => {    const [property, setProperty] = useState<Property | null>(null);    const [error, setError] = useState<string | null>(null); // Добавляем состояние для ошибки    const { id } = useParams();    const userId = useSelector((state: any) => state.user.userId);    const userRole = useSelector((state: any) => state.user.role);    const navigate = useNavigate();      const { acceptBooking, cancelBooking, setMessage } = useBookingActions();    const [isBookingConfirmed, setIsBookingConfirmed] = useState<boolean>(false); // Состояние подтверждения бронирования    const fetchProperty = useCallback(async () => {      try {        const response = await fetch(`http://localhost:3000/property/${id}`);        if (!response.ok) {          throw new Error('Жилье не найдено');        }        const data = await response.json();        setProperty(data);        setError(null);      } catch (error) {        setError('Жилье было удалено или не найдено');        setProperty(null);      }    }, [id]);    // Функция для проверки, подтверждено ли бронирование текущего пользователя    const checkBookingStatus = useCallback((bookings: any[]) => {      const userBooking = bookings.find((booking) => String(booking.status) === 'CONFIRMED' && Number(booking.userId) === Number(userId));      if (userBooking) {        setIsBookingConfirmed(true); // Если найдено подтвержденное бронирование, обновляем состояние      } else {        setIsBookingConfirmed(false);      }    }, [userId]);    useEffect(() => {      fetchProperty();      const intervalId = setInterval(() => {        fetchProperty();      }, 1000); // 1 секунда      return () => clearInterval(intervalId);    }, [fetchProperty]);    useEffect(() => {      if (property) {        checkBookingStatus(property.bookings);      }    }, [property, checkBookingStatus]);    const handleDeleteProperty = async () => {      if (!window.confirm('Вы уверены, что хотите удалить это жилье?')) return;      try {        const response = await fetch(`http://localhost:3000/catalog/${id}`, {          method: 'DELETE',          headers: {            'Content-Type': 'application/json',            Authorization: `Bearer ${localStorage.getItem('token')}`,          },        });        if (response.ok) {          navigate('/catalog');        } else {          const data = await response.json();          console.error(data.message || 'Ошибка при удалении жилья');        }      } catch (error) {        console.error('Ошибка при удалении жилья:', error);      }    };    const updateBookingStatus = (bookingId: number, newStatus: string) => {      setProperty((prevProperty) => {        if (!prevProperty) return null;        const updatedBookings = prevProperty.bookings.map((booking) =>          booking.id === bookingId ? { ...booking, status: newStatus } : booking        );        return { ...prevProperty, bookings: updatedBookings };      });    };    if (error) {      return (        <div>          {error}. <button onClick={() => navigate('/catalog')}>Вернуться в каталог</button>        </div>      );    }    const statusTranslations: { [key: string]: string } = {      PENDING: 'В ожидании',      CONFIRMED: 'Подтверждено',      CANCELLED: 'Отменено',    };    if (!property) {      return <div>Загрузка...</div>;    }    return (      <div className="property-page">        <h1>{property.name}</h1>        <img src={`http://localhost:3000${property.imageUrl}`} alt={property.name} className="property-image" />        <p><strong>Описание:</strong> {property.description}</p>        <p><strong>Местоположение:</strong> {property.location}</p>        <p><strong>Цена за ночь:</strong> {property.pricePerNight} руб.</p>        <p><strong>Категория:</strong> {property.category.name}</p>        <p><strong>Владелец:</strong> {property.owner.login} ({property.owner.email})</p>        <div>          <strong>Критерии:</strong>          <ul>            {property.criteria.map((criterion) => (              <li key={criterion.name}>{criterion.name}</li>            ))}          </ul>        </div>        {userRole === 'OWNER' && (          <div>            <button onClick={() => navigate(`/property/${id}/edit`)}>Редактировать жилье</button>            <button onClick={handleDeleteProperty}>Удалить жилье</button>          </div>        )}        <h2>Доступные бронирования</h2>        <ul>          {property.bookings.map((booking) => {            const startDate = new Date(booking.startDate).toLocaleDateString('ru-RU');            const endDate = new Date(booking.endDate).toLocaleDateString('ru-RU');            return (              <li key={booking.id}>                {startDate} - {endDate} ({statusTranslations[booking.status] || booking.status})                {userRole === 'OWNER' && booking.status === 'PENDING' && (                  <div>                    <button                      onClick={async () => {                        await acceptBooking(booking.id);                        updateBookingStatus(booking.id, 'CONFIRMED');                      }}                    >                      Принять                    </button>                    <button                      onClick={async () => {                        await cancelBooking(booking.id);                        updateBookingStatus(booking.id, 'CANCELLED');                      }}                    >                      Отменить                    </button>                  </div>                )}              </li>            );          })}        </ul>        {userRole === 'USER' && (          <BookingForm propertyId={property.id} userId={Number(userId)} onBookingSuccess={setMessage} />        )}        <Reviews propertyId={property.id} isConfirmed={isBookingConfirmed} /> {/\* Передаем параметр isBookingConfirmed \*/}        {/\* Добавляем компонент чата \*/}        <ChatComponent ownerId={property.owner.id} userRole={userRole} propertyId={property.id} ownerName={property.owner.login}/>      </div>    );  };  export default PropertyPage;  import React, { useState, FormEvent } from 'react';  import { useDispatch } from 'react-redux';  import { decodeJwt } from 'jose';  import { setUserRole, setUserName, setUserId, setUserActiveStatus } from '../slices/userSlice';  import '../styles/Register.css';  const Register: React.FC = () => {    const [login, setLogin] = useState<string>('');    const [email, setEmail] = useState<string>('');    const [password, setPassword] = useState<string>('');    const [message, setMessage] = useState<string>('');    const dispatch = useDispatch();    const handleSubmit = async (e: FormEvent) => {      e.preventDefault();      try {        const response = await fetch('http://localhost:3000/auth/register', {          method: 'POST',          headers: { 'Content-Type': 'application/json' },          body: JSON.stringify({ login, email, password }),        });        const data = await response.json();        setMessage(data.message || 'Регистрация успешна');        if (data.token) {          localStorage.setItem('token', data.token);          // Декодируем токен          const decodedToken: any = decodeJwt(data.token);          // Обновляем данные в localStorage и Redux          if (decodedToken) {            const { role, userId, isActive } = decodedToken;            if (role) {              localStorage.setItem('role', role);              dispatch(setUserRole(role));            }            if (userId) {              localStorage.setItem('userId', userId.toString());              dispatch(setUserId(userId.toString()));            }            if (isActive !== undefined) {              localStorage.setItem('isActive', String(isActive));              dispatch(setUserActiveStatus(isActive)); // Сохраняем статус активности            }            localStorage.setItem('userName', login);            dispatch(setUserName(login));          }          // Перенаправление на каталог          if (decodedToken.isActive === false) {            setMessage('Ваш аккаунт заблокирован');          } else {            window.location.href = 'http://localhost:3001/catalog';          }        }      } catch (error) {        console.error('Ошибка при регистрации:', error);        setMessage('Ошибка при регистрации');      }    };    return (      <div className="register-container">        <h2 className="register-header">Регистрация</h2>        <form onSubmit={handleSubmit} className="register-form">          <input            type="text"            placeholder="Логин"            value={login}            onChange={(e) => setLogin(e.target.value)}            required            className="input-field"          />          <input            type="email"            placeholder="Email"            value={email}            onChange={(e) => setEmail(e.target.value)}            required            className="input-field"          />          <input            type="password"            placeholder="Пароль"            value={password}            onChange={(e) => setPassword(e.target.value)}            required            className="input-field"          />          <button type="submit" className="submit-button">Зарегистрироваться</button>        </form>        {message && <p className="message">{message}</p>}      </div>    );  };  export default Register;  import React, { useState, useEffect } from 'react';  import { useSelector } from 'react-redux';  import '../styles/Reviews.css'; // Подключаем стили  interface Review {    id: number;    user: {      id: number;  // Добавляем поле id для пользователя      login: string;    };    rating: number;    comment: string;  }  interface ReviewsProps {    propertyId: number; // Теперь propertyId передается как пропс    isConfirmed: boolean; // Статус подтверждения бронирования  }  const Reviews: React.FC<ReviewsProps> = ({ propertyId, isConfirmed }) => {    const [reviews, setReviews] = useState<Review[]>([]);    const [newReview, setNewReview] = useState({ rating: 1, comment: '' });    const [error, setError] = useState<string | null>(null);    const userId = useSelector((state: any) => state.user.userId);    const userRole = useSelector((state: any) => state.user.role); // Получаем роль пользователя    // Функция для загрузки отзывов с сервера    const fetchReviews = async () => {      try {        setError(null);        const response = await fetch(`http://localhost:3000/property/${propertyId}/reviews`);        if (!response.ok) {          throw new Error('Не удалось загрузить отзывы');        }        const data = await response.json();        setReviews(data);      } catch (err) {        console.error(err);        setError('Ошибка при загрузке отзывов');      }    };    useEffect(() => {      if (!propertyId) return;      fetchReviews();    }, [propertyId, userId]);    // Обрабатываем логику для установки отзыва пользователя, если он существует    useEffect(() => {      const userReview = reviews.find((review) => review.user.id == userId);      if (userReview) {        // Если отзыв есть, заполняем форму этими данными        setNewReview({          rating: userReview.rating,          comment: userReview.comment,        });      }    }, [reviews, userId]);    const handleSubmit = async (e: React.FormEvent) => {      e.preventDefault();      setError(null);      try {        const response = await fetch(`http://localhost:3000/property/reviews`, {          method: 'POST',          headers: {            'Content-Type': 'application/json',            Authorization: `Bearer ${localStorage.getItem('token')}`,          },          body: JSON.stringify({            propertyId,            userId,            rating: newReview.rating,            comment: newReview.comment,          }),        });        if (!response.ok) {          const errorData = await response.json();          throw new Error(errorData.error || 'Ошибка при добавлении отзыва');        }        await fetchReviews(); // Обновляем отзывы с сервера      } catch (err: any) {        console.error(err);        setError(err.message || 'Неизвестная ошибка');      }    };    const renderStars = (rating: number) => {      return Array.from({ length: 5 }, (\_, index) => (        <span key={index} className={`star ${index < rating ? 'filled' : ''}`}>          ★        </span>      ));    };    return (      <div className="reviews-section">        <h2>Отзывы</h2>        {reviews.length === 0 ? (          <p className="no-reviews">Отзывов пока нет</p>        ) : (          <ul className="reviews-list">            {reviews.map((review) => (              <li key={review.id} className="review-item">                <div className="review-header">                  <strong>{review.user?.login}</strong>                  <div className="stars">{renderStars(review.rating)}</div>                </div>                {review.comment && <p className="review-comment">{review.comment}</p>}              </li>            ))}          </ul>        )}        {userRole !== 'ADMIN' && userRole && userRole !== 'OWNER' && isConfirmed && (          <form onSubmit={handleSubmit} className="review-form">            <h3>Добавить отзыв</h3>            <label>              Рейтинг:              <select                value={newReview.rating}                onChange={(e) =>                  setNewReview({ ...newReview, rating: Number(e.target.value) })                }                className="rating-select"              >                {[1, 2, 3, 4, 5].map((val) => (                  <option key={val} value={val}>                    {val}                  </option>                ))}              </select>            </label>            <label>              Комментарий:              <textarea                value={newReview.comment}                onChange={(e) =>                  setNewReview({ ...newReview, comment: e.target.value })                }                className="comment-textarea"              />            </label>            <button type="submit" className="submit-button">              Добавить отзыв            </button>          </form>        )}        {!isConfirmed && <p>Вы можете оставить отзыв только после подтверждения бронирования.</p>}      </div>    );  };  export default Reviews;  import { useState } from 'react';  export const useBookingActions = () => {    const [message, setMessage] = useState<string>('');    // Функция для принятия бронирования    const acceptBooking = async (bookingId: number) => {      const token = localStorage.getItem('token'); // Получаем токен из localStorage (или другого хранилища)      if (!token) {        setMessage('Токен не найден. Пожалуйста, войдите в систему.');        return;      }      try {        const response = await fetch(`http://localhost:3000/property/booking/${bookingId}/accept`, {          method: 'PUT',          headers: {            'Content-Type': 'application/json',            'Authorization': `Bearer ${token}`, // Передаем токен в заголовке          },        });        if (response.ok) {          setMessage('Бронирование принято!');        } else {          const data = await response.json();          setMessage(data.message || 'Ошибка при принятии бронирования');        }      } catch (error) {        console.error('Ошибка при принятии бронирования:', error);        setMessage('Ошибка при принятии бронирования');      }    };    // Функция для отмены бронирования    const cancelBooking = async (bookingId: number) => {      const token = localStorage.getItem('token'); // Получаем токен из localStorage (или другого хранилища)      if (!token) {        setMessage('Токен не найден. Пожалуйста, войдите в систему.');        return;      }      try {        const response = await fetch(`http://localhost:3000/property/booking/${bookingId}/cancel`, {          method: 'PUT',          headers: {            'Content-Type': 'application/json',            'Authorization': `Bearer ${token}`, // Передаем токен в заголовке          },        });        if (response.ok) {          setMessage('Бронирование отменено!');        } else {          const data = await response.json();          setMessage(data.message || 'Ошибка при отмене бронирования');        }      } catch (error) {        console.error('Ошибка при отмене бронирования:', error);        setMessage('Ошибка при отмене бронирования');      }    };    return { message, acceptBooking, cancelBooking, setMessage };  };  import React, { useState, useEffect } from 'react';  const UserManagement: React.FC = () => {    const [users, setUsers] = useState<any[]>([]);    const [error, setError] = useState<string | null>(null);    useEffect(() => {      // Загрузка всех пользователей      const fetchUsers = async () => {        try {          const response = await fetch('http://localhost:3000/admin/users');          if (!response.ok) {            throw new Error('Не удалось загрузить пользователей');          }          const data = await response.json();          setUsers(data);        } catch (err) {          setError('Не удалось загрузить пользователей');        }      };      fetchUsers();    }, []);    // Изменение статуса активности пользователя    const toggleStatus = async (userId: number, isActive: boolean) => {      try {        const response = await fetch(`http://localhost:3000/admin/users/${userId}/status`, {          method: 'PATCH',          headers: {            'Content-Type': 'application/json',          },          body: JSON.stringify({ isActive: !isActive }),        });        if (!response.ok) {          throw new Error('Ошибка при изменении статуса');        }        const updatedUser = await response.json();        setUsers((prev) =>          prev.map((user) =>            user.id === updatedUser.id ? { ...user, isActive: updatedUser.isActive } : user          )        );      } catch (err) {        setError('Не удалось изменить статус');      }    };    if (error) {      return <div>{error}</div>;    }    return (      <div>        <ul>          {users.map((user) => (            <li key={user.id}>              <span>{user.login} - {user.isActive ? 'Активен' : 'Заблокирован'}</span>              <button onClick={() => toggleStatus(user.id, user.isActive)}>                {user.isActive ? 'Заблокировать' : 'Разблокировать'}              </button>            </li>          ))}        </ul>      </div>    );  };  export default UserManagement;  import React, { useState } from 'react';  import { useDispatch } from 'react-redux';  import { setUserRole, setUserName, setUserId } from '../slices/userSlice'; // Импортируем actions  import '../styles/UserSettings.css'; // Импортируем глобальный CSS  const UserSettings: React.FC = () => {    const dispatch = useDispatch();  // Подключаем dispatch    const [login, setLogin] = useState('');    const [email, setEmail] = useState('');    const [password, setPassword] = useState('');    const [oldPassword, setOldPassword] = useState('');    const [error, setError] = useState<string | null>(null);    const [success, setSuccess] = useState<string | null>(null);    const handleSubmit = async (e: React.FormEvent) => {      e.preventDefault();      const token = localStorage.getItem('token');      if (!token) {        setError('Вы не авторизованы');        return;      }      if (password && !oldPassword) {        setError('Для изменения пароля необходимо ввести старый пароль');        return;      }      const requestData: any = {        login,        email,        oldPassword,      };      if (password) {        requestData.password = password; // Добавляем новый пароль только если он задан      }      try {        const response = await fetch('http://localhost:3000/auth/user/update', {          method: 'PUT',          headers: {            'Content-Type': 'application/json',            Authorization: `Bearer ${token}`,          },          body: JSON.stringify(requestData),        });        if (!response.ok) {          const data = await response.json();          setError(data.message || 'Ошибка при обновлении информации');          setSuccess(null);          return;        }        const data = await response.json();        setSuccess('Информация успешно обновлена');        setError(null);        // Обновляем данные в Redux        dispatch(setUserName(data.user.login)); // Обновляем имя пользователя        dispatch(setUserRole(data.user.role)); // Обновляем роль пользователя        dispatch(setUserId(data.user.id)); // Обновляем id пользователя      } catch (error) {        setError('Ошибка при обновлении информации');        setSuccess(null);      }    };    return (      <div className="user-settings">        <h1>Настройки пользователя</h1>        {error && <div className="error">{error}</div>}        {success && <div className="success">{success}</div>}        <form onSubmit={handleSubmit} className="form">          <div>            <label htmlFor="login" className="label">Логин</label>            <input              type="text"              id="login"              className="input"              value={login}              onChange={(e) => setLogin(e.target.value)}              placeholder="Введите новый логин"            />          </div>          <div>            <label htmlFor="email" className="label">Email</label>            <input              type="email"              id="email"              className="input"              value={email}              onChange={(e) => setEmail(e.target.value)}              placeholder="Введите новый email"            />          </div>          <div>            <label htmlFor="oldPassword" className="label">Старый пароль</label>            <input              type="password"              id="oldPassword"              className="input"              value={oldPassword}              onChange={(e) => setOldPassword(e.target.value)}              placeholder="Введите старый пароль"            />          </div>          <div>            <label htmlFor="password" className="label">Новый пароль (если хотите изменить)</label>            <input              type="password"              id="password"              className="input"              value={password}              onChange={(e) => setPassword(e.target.value)}              placeholder="Введите новый пароль"            />          </div>          <button type="submit" className="button">Сохранить изменения</button>        </form>      </div>    );  };  export default UserSettings; |

Листинг 2 – Клиентская часть