МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных Технологий

Кафедра Информационных систем и технологий

Специальность 1-40 01 01 «Программное обеспечение информационных технологий»

Специализация 1-40 01 01 10 «Программное обеспечение информационных технологий (программирование интернет-приложений)»

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**к курсовому проекту на тему:**

Web-приложение «Интернет-магазин электронной техники»

Выполнил студент Коршун Никита Игоревич

(Ф.И.О.)

Руководитель проекта асс. Нистюк О.А.

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

И.о. заведующего кафедрой ст.преп. Блинова Е.А.

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

Курсовой проект защищен с оценкой

Минск 2024

**Содержание**

[Введение 5](#_Toc167211877)

[1 Постановка задачи 6](#_Toc167211878)

[1.1 Аналитический обзор аналогов 6](#_Toc167211879)

[1.1.1. Metacritic 7](#_Toc167211880)

[1.1.2. IGN 7](#_Toc167211881)

[1.1.3. Steam 8](#_Toc167211882)

[1.2 Разработка функциональных требование и вариантов использования 9](#_Toc167211883)

[1.3 Выводы по разделу 10](#_Toc167211884)

[2 Проектирование web-приложения 11](#_Toc167211885)

[2.1 Обобщенная структура управлением приложения 11](#_Toc167211886)

[2.2 Диаграммы UML. 11](#_Toc167211887)

[2.4 Выводы по разделу 12](#_Toc167211888)

[3 Разработка web-приложения 13](#_Toc167211889)

[3.1 Разработка серверной части web-приложения 13](#_Toc167211890)

[3.2 Разработка клиентской части web-приложения 16](#_Toc167211891)

[3.3 Выводы по разделу 17](#_Toc167211892)

[4 Тестирование web-приложения 18](#_Toc167211893)

[5 Руководство пользователя 20](#_Toc167211894)

[5.1 Руководство пользователя 20](#_Toc167211895)

[5.2 Установка приложения 23](#_Toc167211896)

[5.3 Выводы по разделу 24](#_Toc167211897)

[Заключение 25](#_Toc167211898)

[Список используемых источников 26](#_Toc167211899)

[Приложение А 27](#_Toc167211900)

# Введение

Целью работы является разработка кроссплатформенного веб-приложения на тему «Поиск и оценка компьютерных игр». Приложение представляет из себя интернет-магазин, предоставляющий пользователю удобный функционал для ознакомления с ассортиментом магазина и оформления заказов.

Node.js – программная платформа, основанная на движке Chrome V8. Node.js асинхронен и событийно-ориентирован. Данная платформа позволяет JavaScript взаимодействовать с устройствами ввода-вывода через свой API, подключать другие внешние библиотеки. Node.js предназначен для построения масштабируемых сетевых приложений, преимущественно серверов.

В качестве СУБД была выбрана MSSQL, поскольку данная база данных обладает необходимой простотой, производительностью и надежностью.

Написанное приложение строится на клиент-серверной архитектуре. В качестве клиента выступает приложение, написанное на HTML и JavaScript, которое отображается в браузере. Для сервера была выбрана платформа Node.js в связи с кроссплатформенностью, простотой развертки сервера и гибкости работы за счёт npm и пакетов.

В пояснительной записке содержится краткая информация о сопоставимых продуктах, структуре и реализации проекта, а также инструкции по использованию и тестированию приложения.

# 1 Постановка задачи

Важным этапом разработки любого нового продукта является анализ существующих на рынке аналогов. Он позволяет определить, какие решения уже представлены потребителям, выявить их сильные и слабые стороны, а также определиться с направлениями для усовершенствований.

Наше веб-приложение предназначено для поиска и просмотра информации об играх. В отличие от интернет-магазинов, наше приложение фокусируется на предоставлении пользователям информации о видеоиграх, включая обзоры, рейтинги, отзывы и рекомендации. На текущий момент рынок подобных приложений включает множество платформ с различным уровнем функционала и удобства. Однако, многие из них не могут предложить полноценного комплекса инструментов, сочетающего в себе удобство, гибкость и высокие стандарты пользовательского опыта. Это оставляет простор для новых разработок в этой сфере.

Задача проекта: разработать архитектуру приложения, взаимодействие с которой будет понятно любому пользователю. Построить базу данных и выполнить тестирование готового продукта.

Должны быть выполнены следующие требования:

* обеспечивать возможность регистрации и авторизации;
* поддерживать роли модератора и пользователя;
* предоставлять функционал по поиску и фильтрации по различным критериям;
* позволять пользователям оценивать игры и оставлять свои отзывы;
* предоставлять возможность пользователю сохранять игры в списке желаемых игр;
* включать функцию уведомлений на выбранные игры;
* предоставлять возможность пользователю делиться информацией о понравившихся играх;
* поддерживать мультиязычность для удобства пользователей из разных стран;
* предоставлять возможность пользователям создавать свои списки игр;
* обеспечивать функционал по обратной связи;
* предоставлять рекомендации пользователю на основе его предпочтений;

## 1.1 Аналитический обзор аналогов

Были проанализированы цели и задачи, поставленные в данном 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. Metacritic

Одним из самых популярных альтернативных решений является сайт Metacritic. Metacritic предоставляет пользователям агрегированные обзоры и рейтинги для видеоигр, фильмов, телешоу и музыки.

Metacritic предлагает удобный интерфейс для поиска и фильтрации игр по различным критериям, таким как платформа, жанр и дата выпуска. Пользователи могут оставлять свои отзывы и оценки, что помогает другим пользователям в выборе игр. Однако, Metacritic не предоставляет возможности сохранять игры в списке желаемых или получать уведомления о новостях и обновлениях по выбранным играм.

Интерфейс интернет ресурса представлен на рисунке 1.1.

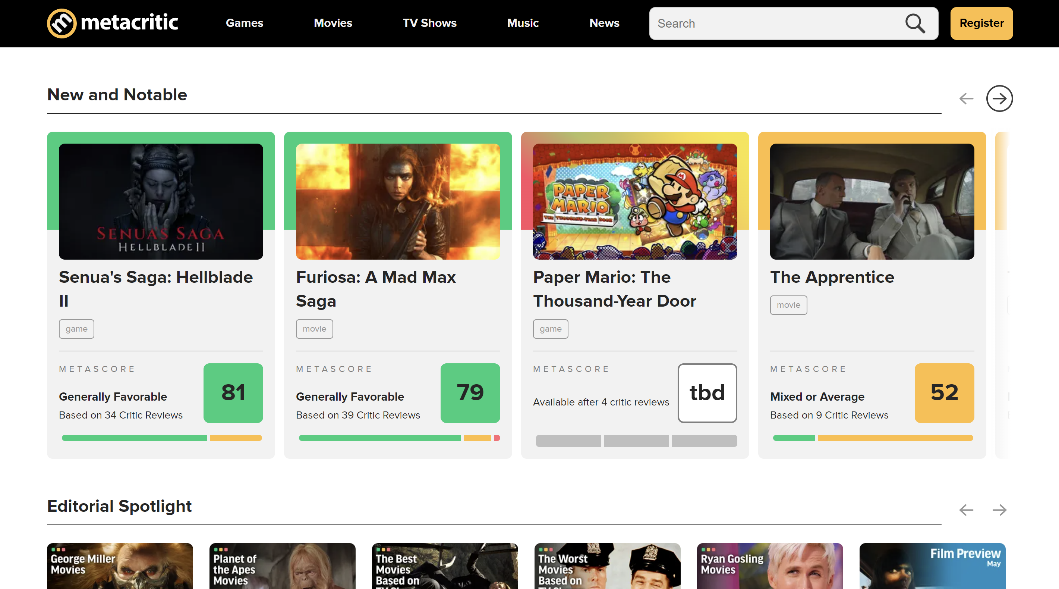


Рисунок 1.1 – Интернет-ресурс «Metacritic»

Из обзора такого аналога, как Metacritic, для нашего приложения можно взять идею агрегирования обзоров и рейтингов, а также функционал по поиску и фильтрации игр.

* + 1. IGN

IGN – это один из ведущих сайтов, предоставляющих новости, обзоры и рейтинги для видеоигр, фильмов и телешоу.

Главная страница IGN отображает последние новости и обзоры, а также предоставляет быстрый доступ к различным разделам сайта. Пользователи могут оставлять свои комментарии и участвовать в обсуждениях, что создает активное сообщество вокруг платформы. Однако, IGN не предоставляет возможности пользователям создавать свои списки игр или получать персонализированные рекомендации.

Интерфейс интернет ресурса представлен на рисунке 1.2.

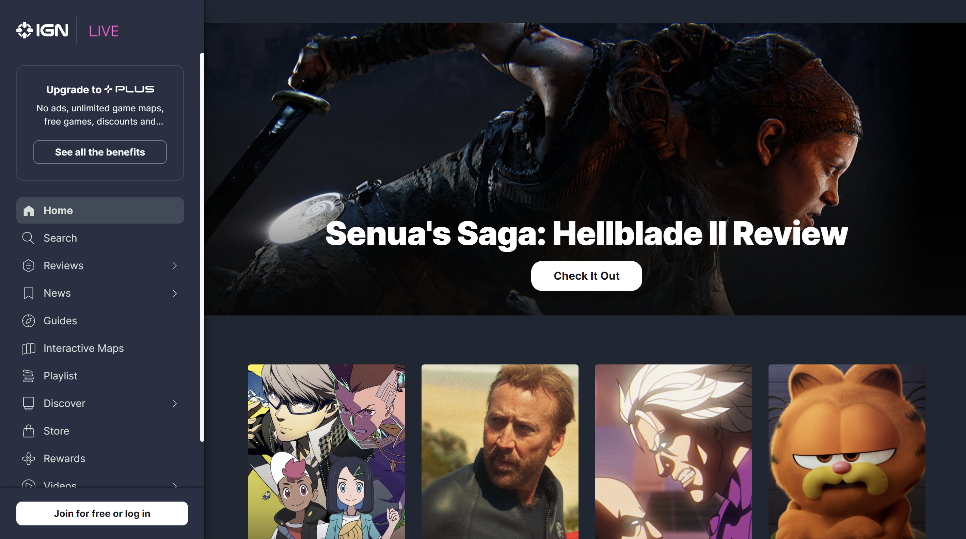


Рисунок 1.2 – Интернет-ресурс «IGN»

Из обзора такого аналога, как IGN, для нашего приложения можно взять идею активного сообщества и функционал по добавлению комментариев.

* + 1. Steam

Steam – это платформа для цифровой дистрибуции видеоигр, разработанная компанией Valve.

Steam предлагает пользователям широкий ассортимент игр, возможность оставлять отзывы и оценки, а также функционал по созданию списков желаемых игр. Платформа также предоставляет персонализированные рекомендации на основе предпочтений пользователя и его игровой активности. Однако, интерфейс Steam может быть перегружен для новых пользователей, и не всегда интуитивно понятен.

Интерфейс интернет ресурса представлен на рисунке 1.3.

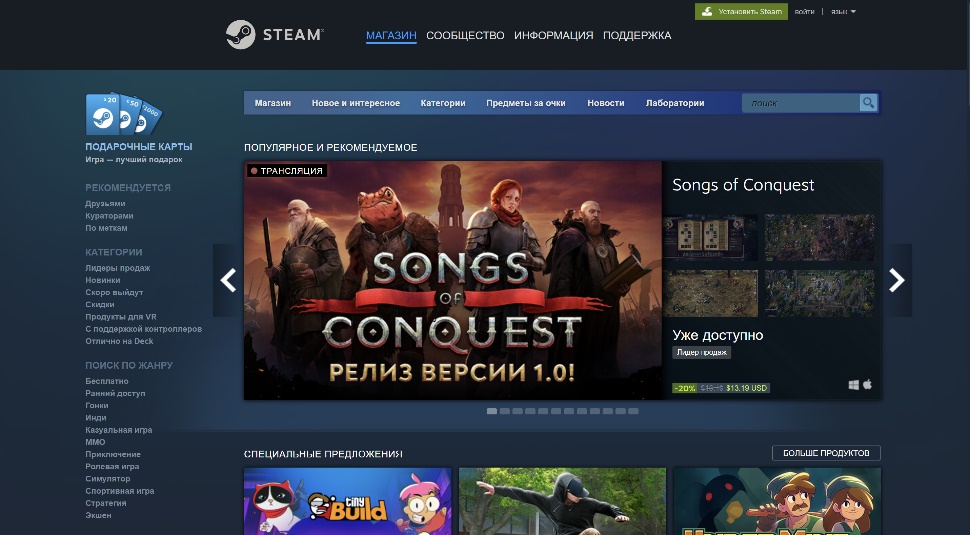


Рисунок 1.3 – Интернет-ресурс «Steam»

Из обзора такого аналога, как Steam, для нашего приложения можно взять идею по созданию списков желаемых игр.

Анализ аналогичных решений, таких как Metacritic, IGN и Steam, позволил выявить ключевые функции и возможности, которые могут быть полезны для нашего веб-приложения. Внедрение агрегированных обзоров и рейтингов, активного сообщества, функционала по созданию списков желаемых игр обеспечит высокое качество пользовательского опыта и удовлетворит потребности широкой аудитории.

## 1.2 Разработка функциональных требование и вариантов использования

Приложение должно обеспечивать функционал для регистрации и авторизации, который позволит идентифицировать пользователей. Также приложение должно обеспечить функционал просмотра информации об играх с возможностью фильтрации по различным критериям. Для работы с играми должны быть созданы функции добавления, редактирования и удаления информации об играх. Удаление должно быть полным, чтобы информация полностью удалялась из базы данных. Администратор должен иметь функции для контроля как за информацией об играх, так и за всеми пользователями. Диаграмма вариантов использования представлена на рисунке 1.4.



Рисунок 1.4 – Диаграмма вариантов использования

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

## 1.3 Выводы по разделу

В данном разделе мы рассмотрели аналогичные веб-приложения. Исходя из этих аналогов можно взять пару идей для своего веб-приложения. Например, данное приложение должно обеспечивать возможность добавление игр в список желаемого, как и Steam, данное приложение должно предоставлять функционал для добавления комментариев к играм, как и IGN.

Также в данном разделе рассмотрели весь основной функционал, который должен присутствовать в конечном приложении. Данный функционал рассчитан на то, что он позволит довольно-таки просто начать пользоваться этим веб-приложением, а также что этот функционал позволит просматривать и оценивать игры. Была приведена UML диаграмма, на которой все и приведено.

# 2 Проектирование web-приложения

## 2.1 Обобщенная структура управлением приложения

Для обеспечения управления приложением с использованием базы данных необходимо разработать удобный и интуитивно понятный интерфейс, который позволит пользователю взаимодействовать с базой данных и эффективно управлять данными. Это может включать в себя разработку оптимизированных запросов для вставки, обновления и удаления данных, а также разработку механизмов для извлечения и обработки информации из базы данных.

В функциональность приложения интернет-магазина должны входить функции для удобного поиска игр в приложении, функции добавления игр в список желаемого и другие подобные функции.

## 2.2 Диаграммы UML.

Диаграмма базы данных таблиц (Database Table Diagram) – это визуальное представление структуры базы данных и отношений между таблицами, которые хранятся в этой базе данных. Диаграмма базы данных представлена на рисунке 2.1.

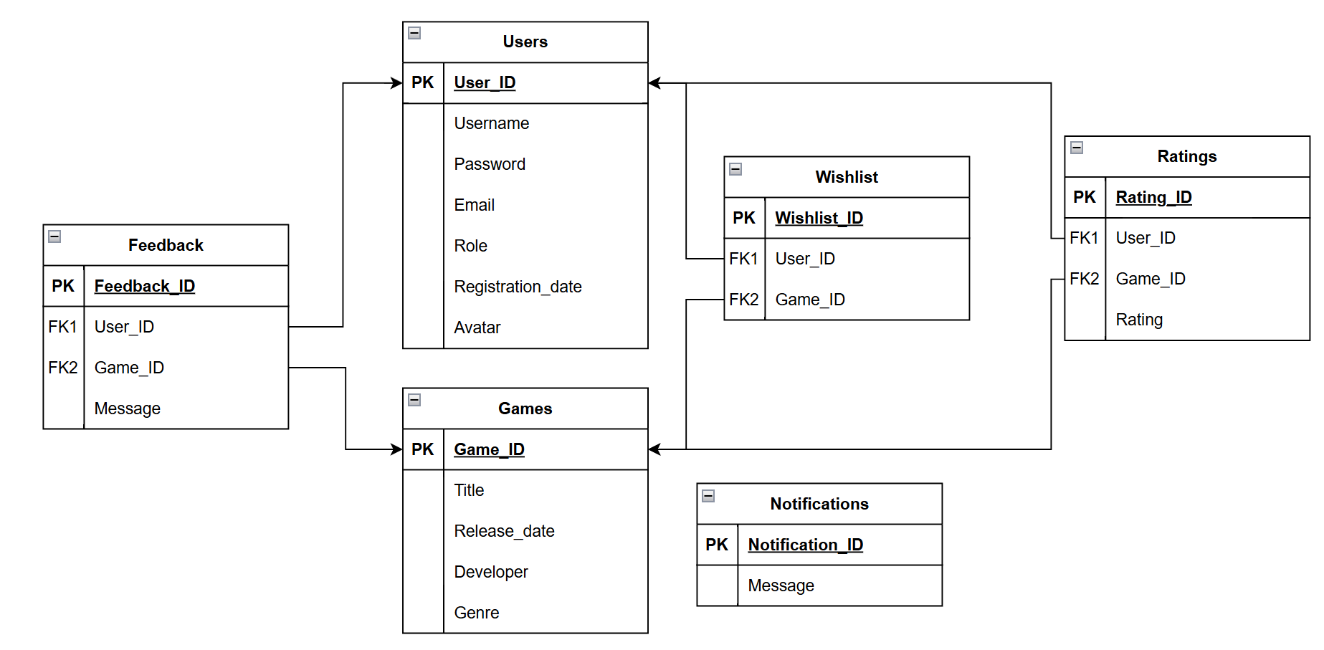


Рисунок 2.1 – Логическая схема базы данных

В базе данных находится 6 таблиц. В таблице Users хранятся данные для идентификации пользователей, зарегистрированных в приложении, столбец username содержит никнейм пользователя, который используется как логин в процессе авторизации, password – хешированный алгоритмом SHA-256 пароль, email – почта пользователя, role – роль пользователя, registration-date – дата регистрации пользователя, avatar – фото профиля пользователя.

Feedback – отзывы пользователей об играх, столбец feedback\_id содержит идентификатор отзыва, которому соответствует запись в таблице, user\_id – содержит идентификатор пользователя, который добавил отзыв, game\_id – идентификатор игры, которой был добавлен отзыв, message – комментарий.

Games – информация об играх, столбец game\_id – идентификатор игры, title – название игры, release\_date – дата выхода игры, developer – разработчики, genre – жанр игры.

Notifications – данные об уведомлениях, столбец notification\_id – идентификатор уведомления, message – текст уведомления.

Wishlist – данные об играх, добавленных в список желаемого, столбец wishlist\_id – идентификатор избранного элемента, user\_id – идентификатор пользователя, game\_id – идентификатор игры.

Ratings – данные об оценках игры, столбец rating\_id – идентификатор элемента рейтинга, user\_id – идентификатор пользователя, game\_id – идентификатор игры, rating – оценка игры.

## 2.4 Выводы по разделу

Разработка архитектуры проекта необходима для определения структуры и функциональности приложения. Обобщенная структура управления приложения позволяет определить, какие компоненты необходимы для реализации приложения и как они должны взаимодействовать между собой.

В данном разделе рассмотрели структуру базы данных MSSQL, ее основные сущности и как они связаны.

# 3 Разработка web-приложения

## 3.1 Разработка серверной части web-приложения

Серверная часть написана на фреймворке Express для платформы NodeJS, а также были использованы различные пакеты, такие как ws, cheerio, axios, path, crypto, jsonwebtoken, tedious и sequelize, winston. Непосредственно для взаимодействия с MSSQL нам необходимы только пакеты tedious и sequelize. Схема взаимодействия Node.js с базой данных представлена на рисунке 3.1.

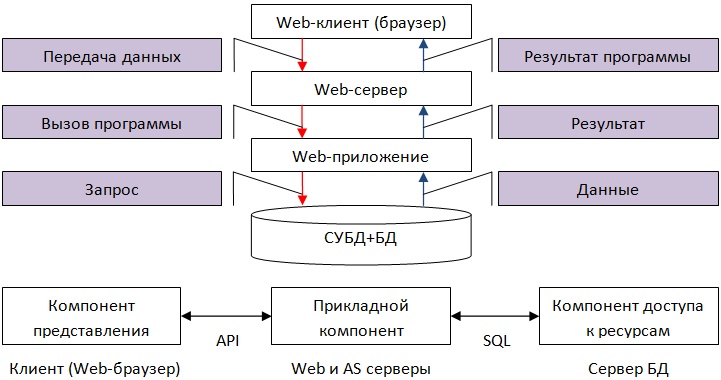


Рисунок 3.1 – Схема взаимодействия NodeJS и базы данных

Структура сервера была разработана соответствии с архитектурным паттерном Model-View-Controller (далее - MVC), что делает её читабельной и легко расширяемой.

MVC подразумевает под собой разделение проекта на три независимые компоненты:

Model – модель, предоставляет данные и реагирует на команды контроллера, изменяя своё состояние.

View – представление, отвечает за отображение данных модели пользователю, реагируя на изменения модели.

Controller – контроллер, интерпретирует действия пользователя, оповещая модель о необходимости изменений.

В данном проекте модель работает с базой данных. Каждый объект базы данных имеет свой класс-сущность, сформированный при помощи Sequelize. Эти классы реализованы в отдельных модулях, что позволяет при необходимости вызвать какой-либо метод репозитория в контроллере.

Контроллер выполняет функцию маршрутизатора запросов к серверу. Контроллер (как часть MVC) состоит из двух частей:

* маршрутизатора index, который принимает запросы и передает их определенным контроллерам;
* контроллеров репозитория controllers, каждый из которых обрабатывает свой определенный путь маршрутизатора.

Дерево маршрутизации можно увидеть в листинге 3.1.

app.use(routes)

app.use(comments)

app.use(ratings)

app.use(user)

app.use(wishlist)

app.use(adminPanel)

Листинг 3.1 – Дерево маршрутизации

Архитектуру файлов контроллера MVC можно увидеть на рисунке 3.2.

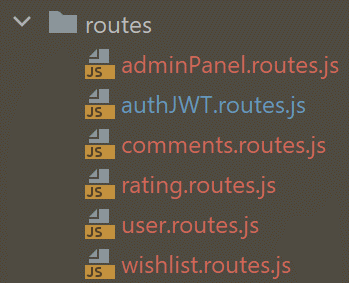


Рисунок 3.2 – Архитектура файлов контроллера MVC

Проект представляет собой веб-приложение, разработанное с использованием Node.js и Express. Каждый контроллер выполняет определенные операции над моделью, формирует ответ и отправляет данные клиентской части приложения. HTML-разметка динамически формируется на сервере и отправляется клиенту, обеспечивая интерактивное взаимодействие с пользователем.

Так как приложение является браузерным, нужно позаботиться о механизме авторизации пользователей. В приложении реализована JWT-авторизация с использованием пакета jsonwebtoken и crypto для шифрования пароля пользователя. Элемент конвейера обработки запросов, который отвечает за проверку пользователя на факт авторизации представлен в листинге 3.2.

const authenticateToken = (req, res, next) => {

const token = req.cookies.accessToken;

if (!token) return res.sendStatus(401);

jwt.verify(token, 'access\_secret', (err, user) => {

if (err) return res.sendStatus(403);

req.user = user;

next();

});

};

Листинг 3.2 – Реализация middleware проверки авторизации

В приложении также реализован функционал получения уведомлений на основании протокола WebSocket. Клиентская часть при получении сообщений от сервера создает обьект Notification и отображает уведомление на экране. Пример функции для создания уведомления представлен на в листинге 3.3.

function showNotification(message) {

// Проверка, поддерживается ли API уведомлений в браузере

if ('Notification' in window && Notification.permission === 'granted') {

console.log(message)

new Notification('Новое уведомление', { body: message });

} else if ('Notification' in window && Notification.permission !== 'denied') {

// Запрос разрешения на показ уведомлений

Notification.requestPermission().then((permission) => {

if (permission === 'granted') {

// Создание уведомления

new Notification('Новое уведомление', { body: message });

}

});

} else {

console.log('Уведомления не поддерживаются в вашем браузере');

}

}

Листинг 3.3 – Пример функции создания уведомления

Для получения изображений игр использовались запросы на удаленный API.

При загрузке страницы происходит запрос на API и отображение пришедших изображений. Пример функции получения изображений показан на листинге 3.4.

async function getGameImages(gameTitle) {

const formattedTitle = formatGameTitle(gameTitle);

const url = `https://www.gamespot.com/games/${formattedTitle}/`;

try {

const response = await axios.get(url);

const $ = cheerio.load(response.data);

let images = extractImageUrls($);

if (images.length === 0) {

const words = gameTitle.split(' ');

if (words.length > 1) {

const newTitle = words.slice(0, -1).join(' ');

return await getGameImages(newTitle);

} else {

throw new Error('No images found');

}

}

return images;

} catch (error) {

winston.error('Error fetching game images:', error);

return [];

}

}

Листинг 3.4 – Пример функции получения изображений с удаленного сервера

## 3.2 Разработка клиентской части web-приложения

Так как проект является кроссплатформенным, клиентская часть была разработана для использования в браузере. Чтобы клиент не видел задержки при запросе у сервера необходимых данных, клиент запрашивает данные асинхронно с использованием promise-ориентированных AJAX и Axios, что позволяет отправить данные при помощи определенного метода на строго фиксированный путь. Пример реализации AJAX-запросов представлен в листинге 3.5.

fetch("/login", {

method: "POST",

headers: {

"Content-Type": "application/json"

},

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

})

.then(function(response) {

if (response.status === 200) {

window.location.href = "/";

} else {

return response.text();

}

})

Листинг 3.5 – Пример функции с AJAX-запросом

После каждого успешно выполненного запроса на сервере и получения данных, происходит изменение значений элементов в соответствии с содержимым ответа.

На листинге 3.6 представлен пример применения функции запроса и использования данных изменения значений полей.

fetch("/getComments?game\_id=" + game\_id)

.then(function(comments) {

var commentsContainer = document.getElementById("comments-container");

commentsContainer.innerHTML = "";

if (comments.length === 0) {

var noCommentsBlock = document.createElement("div");

noCommentsBlock.classList.add("no-comments-block");

noCommentsBlock.textContent = "Нет комментариев";

commentsContainer.appendChild(noCommentsBlock);

Листинг 3.6 – Пример использования запроса комментариев

## 3.3 Выводы по разделу

В данном разделе мы рассмотрели основные момент разработки веб-приложения. Рассмотрели, как построено взаимодействие клиента с сервером по протоколу HTTP. Рассмотрели, как хранится информация и как отображается на клиенте.

# 4 Тестирование web-приложения

Вначале нужно протестировать форму регистрации, введя некорректные данные. Результат проделанного текста показан на рисунке 4.1.

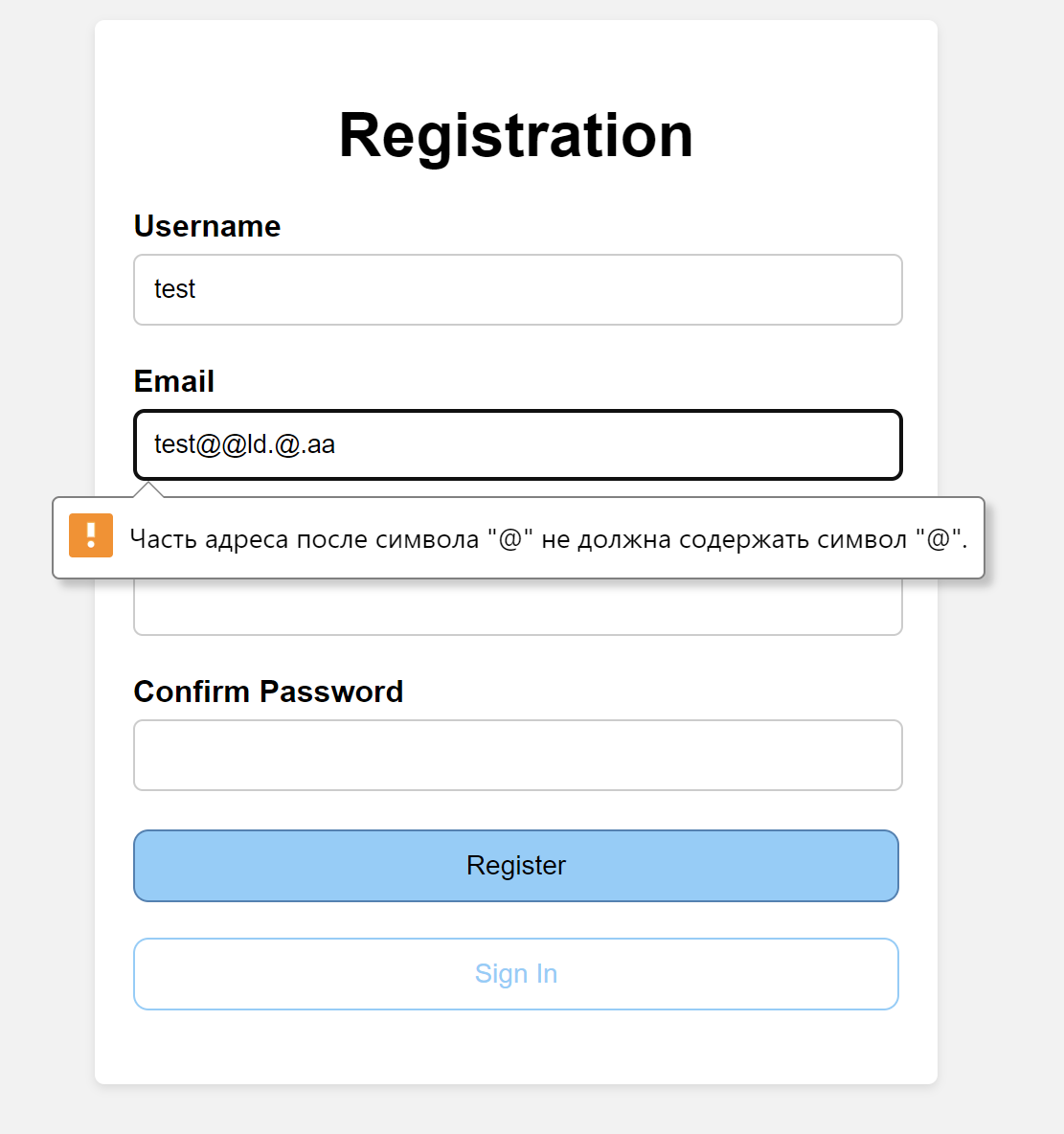


Рисунок 4.1 – Результат проделывания первого теста

По картинке 4.1 видно, что система отработала правильно и указала на ошибку.

Второй тест будет связан с проверкой на поле «Пароль». Если отправить пустое пароль, пользователь не должен зарегистрироваться. Результат данного теста показан на рисунке 4.2.

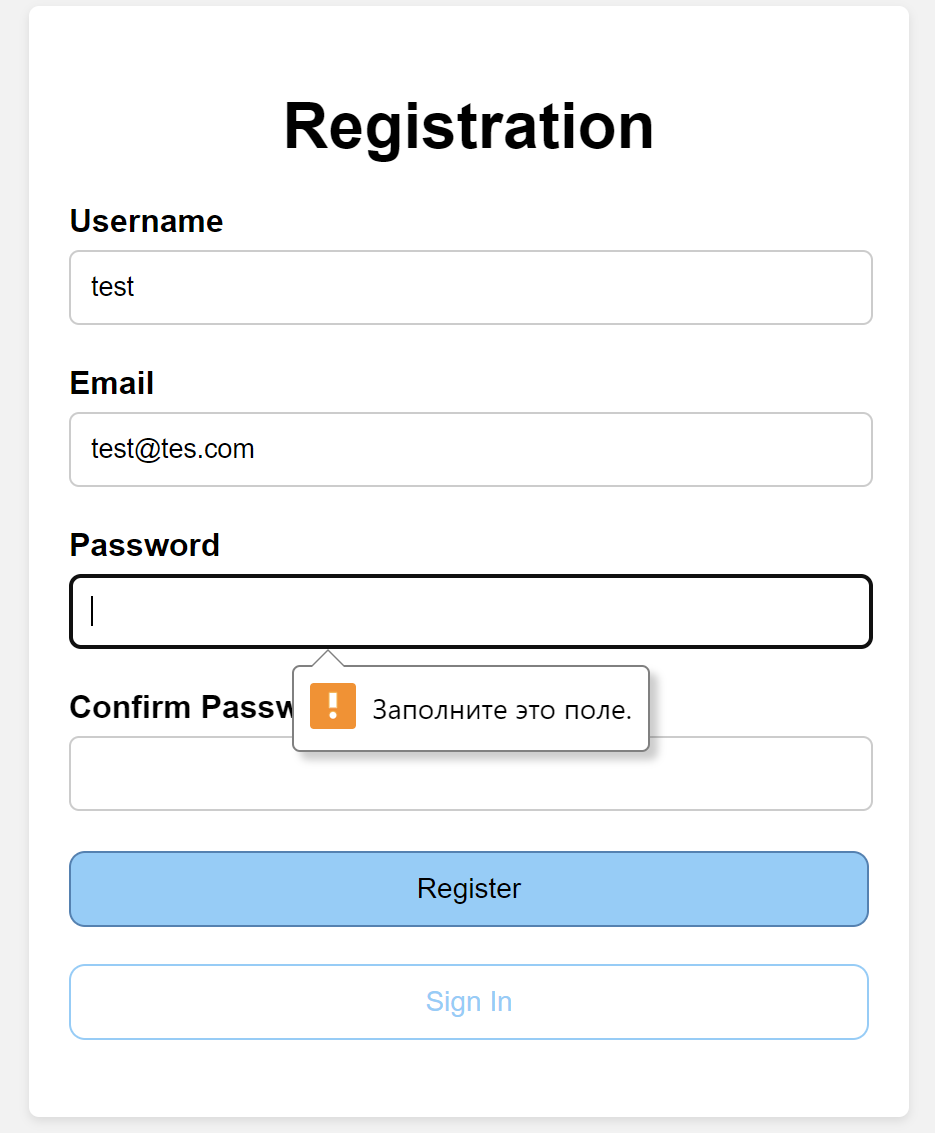


Рисунок 4.2 – Результат проделывания второго теста

По результатам второго теста, видно, что приложение отработало верно.

Третий тест будет заключатся в проверке регистрации по существующим данным. Мы воспользуемся уже использованной электронной почтой и попробуем зарегистрироваться снова. Ожидаемый результат – это вывод сообщении о ошибке. Результат проделывания теста три показан на рисунке 4.3.

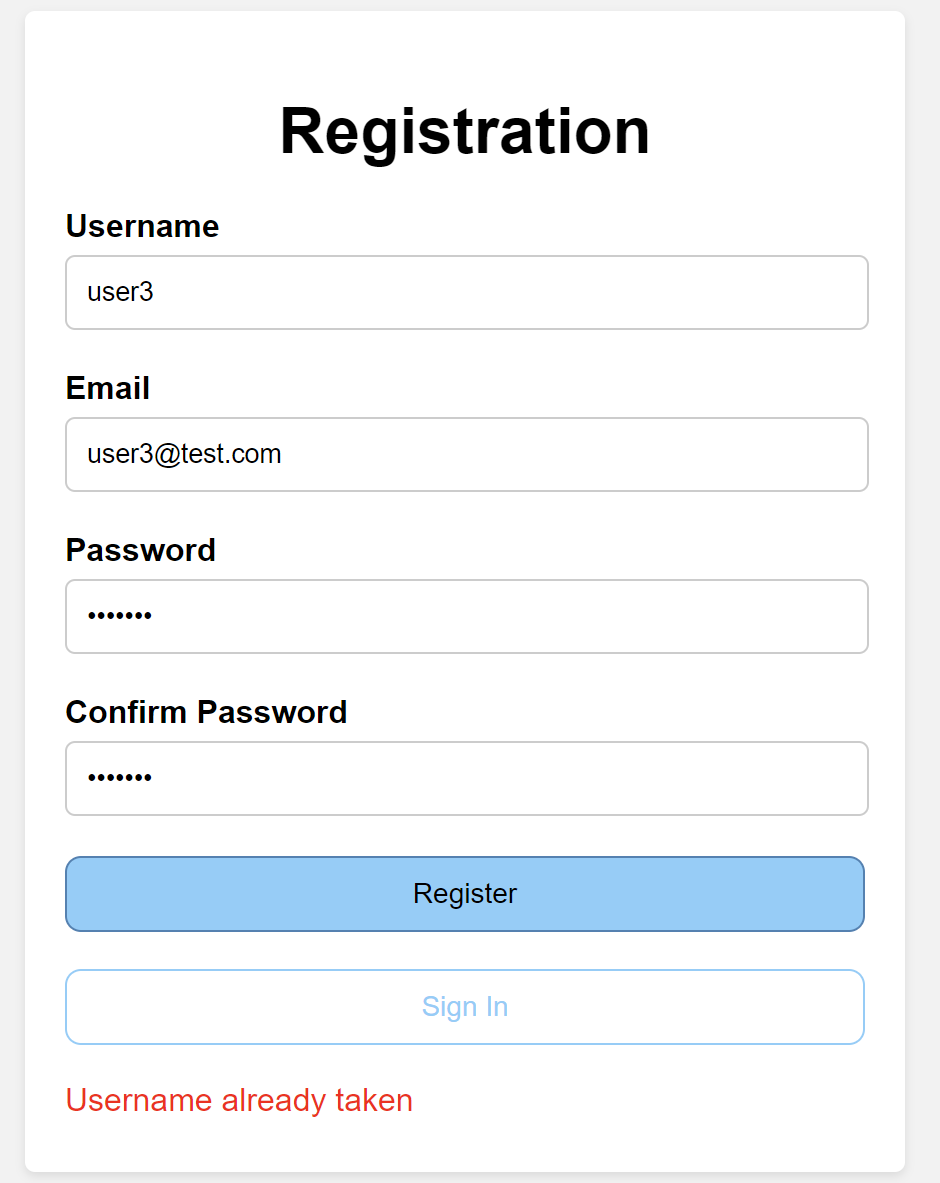


Рисунок 4.3 – Результат проделывания третьего теста

По результатам третьего теста, видно, что приложение отработало верно.

Четвертый тест будет заключатся в том, что мы попробуем изменить имя пользователя на пустую строку. Ожидаемый результат – редактирование не пройдет. Результат данного тест показан на рисунке 4.4.

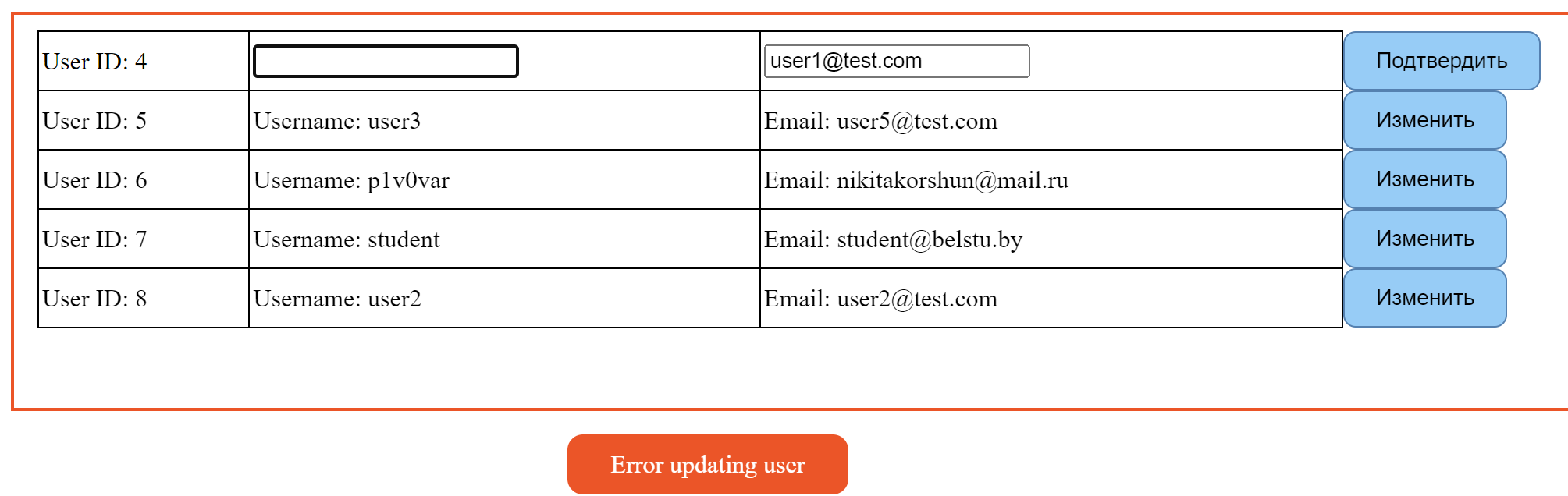


Рисунок 4.4 – Результат проделования четвертого теста

Результат четвертого теста соответствует ожидаемому результату. Данное приложение готово к публикации и работе в сети Интернет.

# 5 Руководство пользователя

## 5.1 Руководство пользователя

При открытии веб-приложения вас будет встречать главная страница, на которой вы можете увидеть список игр и категорий, а также поле для поиска. Рисунок главной страницы представлен на рисунке 5.1.

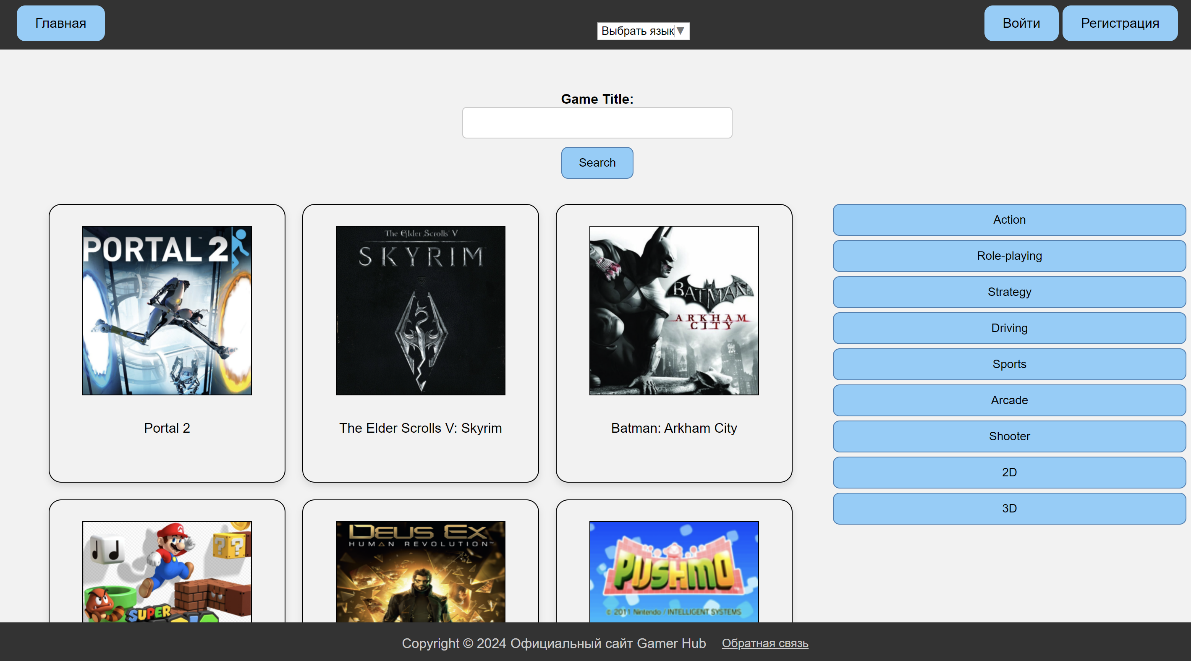


Рисунок 5.1 – Главная страницы

При нажатии на кнопку входа или регистрации пользователь попадает на страницу авторизации/регистрации соответственно. Рисунок страницы авторизации представлен на рисунке 5.2.

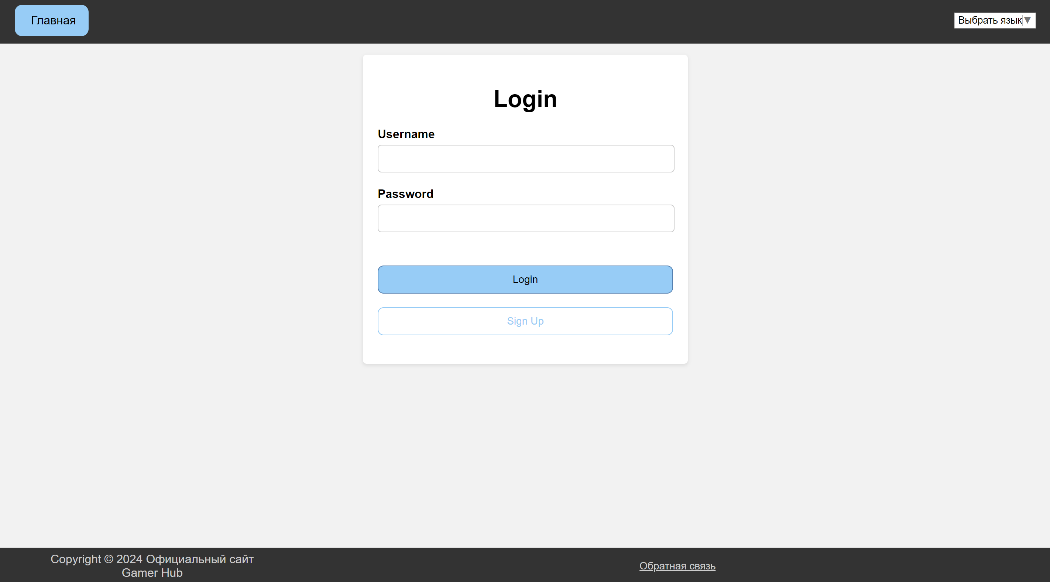


Рисунок 5.2 – Страница авторизации

После входа под определенным пользователем при переходе на страницу с игрой будут доступны кнопки для оценки игры и блок ввода для написания комментария к игре. Страница игры представлена на рисунке 5.3.

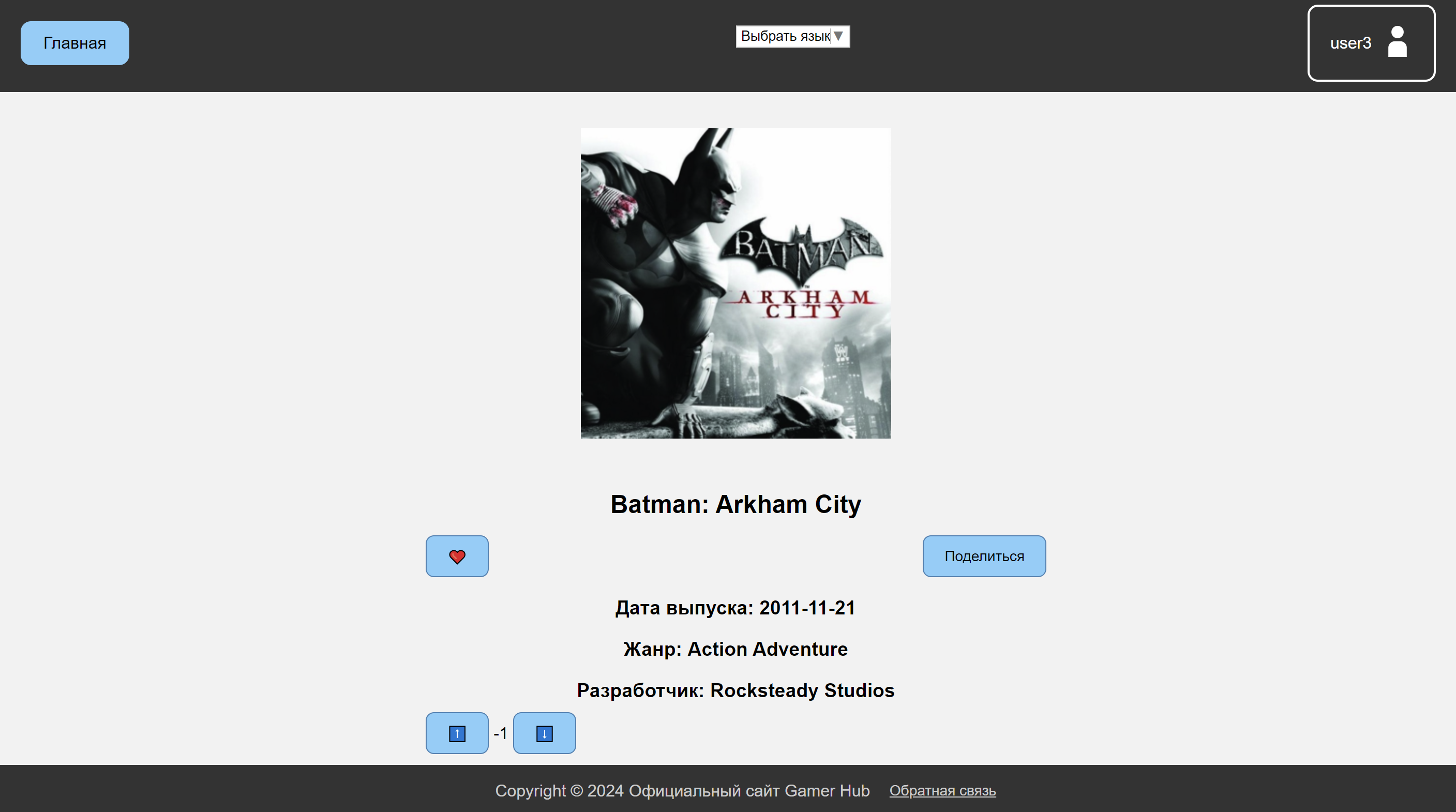


Рисунок 5.3 – Страница игры

Пользователь может создать новый аккаунт. Для этого ему надо перейти на страницу регистрации, где ему будет доступна интуитивно понятная форма. Рисунок формы регистрации представлен на рисунке 5.4.

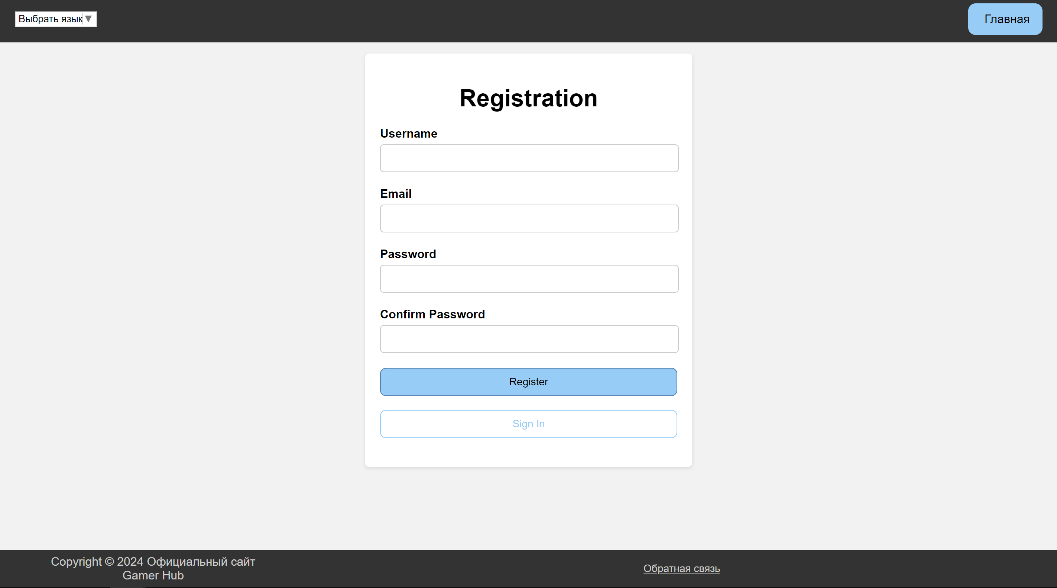


Рисунок 5.4 – Страница регистрации

При нажатии на имя пользователя после авторизации, отображается меню в котором можно перейти в личный кабинет, список желаний или выйти с аккаунта. В личном кабинете пользователь может просмотреть свою информацию и изменить данные. Страница профиля пользователя представлена на рисунке 5.5.

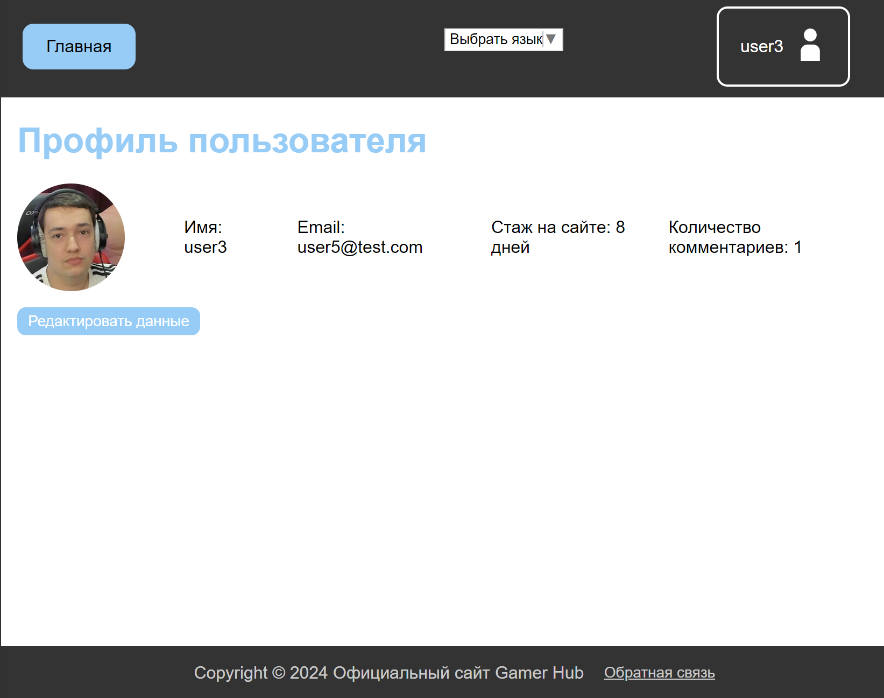


Рисунок 5.5 – Профиль пользователя

Если пользователь является администратором, то в профиле в меню пользователя появится опция админ-панели. Страница админ-панели представлена на рисунке 5.6.

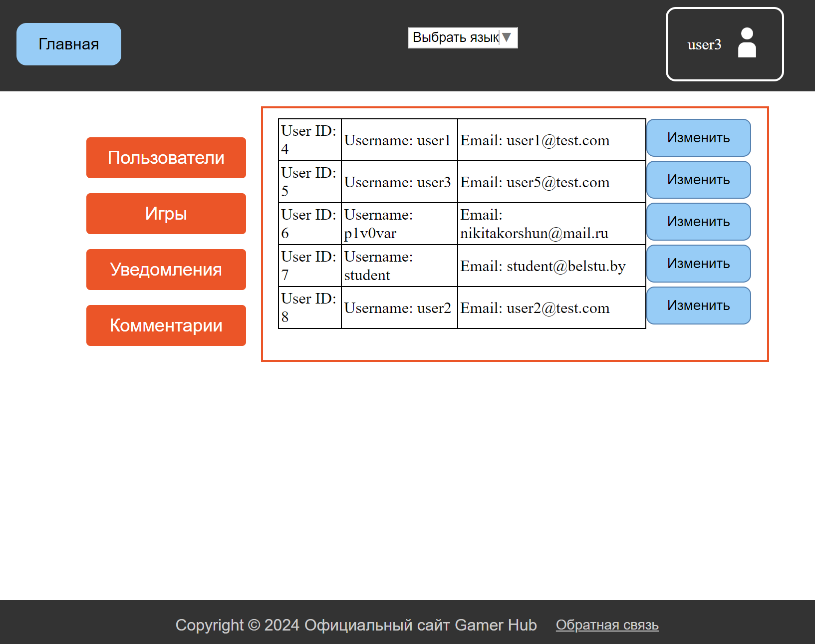


Рисунок 5.6 – Админ-панель

При добавлении игры в избранное она появляется в списке желаемого пользователя. Доступ к этой странице можно получить из меню пользователя. Список желаемого представлен на рисунке 5.7.

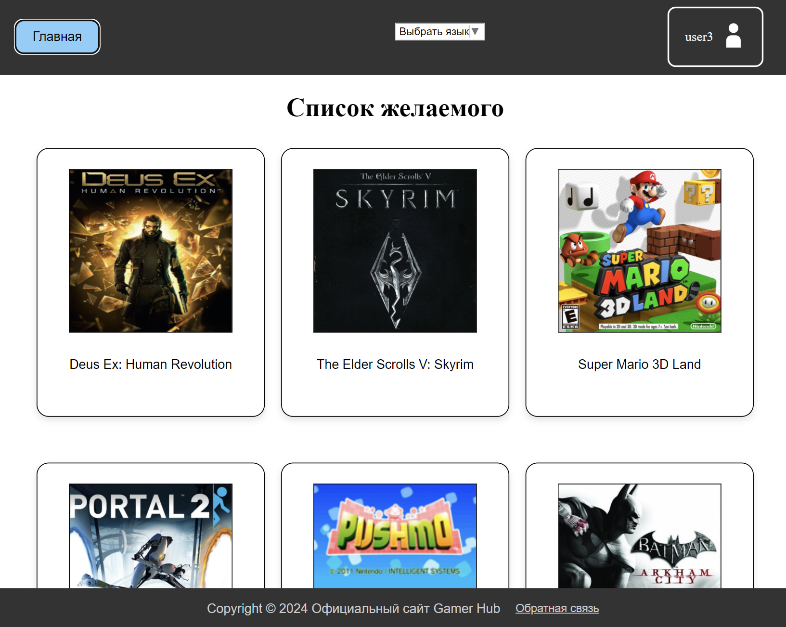


Рисунок 5.7 – Список желаемого

Если пользователь захочет связаться с поддержкой он может оставить свою почту, нажав на кнопку обратная связь. Окно ввода почты для обратной связи изображено на рисунке 5.8.

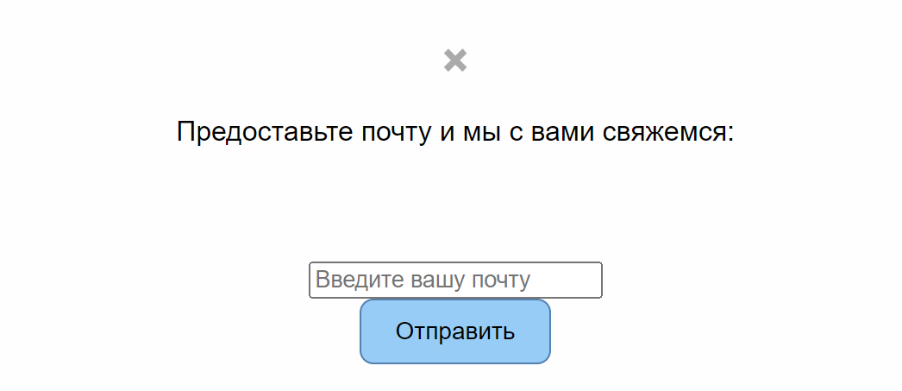


Рисунок 5.8 – Окно обратной связи

Мы рассмотрели основной функционал данного приложения. Он должен помочь освоится новому пользователю. Мы увидели, что интерфейс более-менее понятный и отзывчивый. Дизайн приложения довольно-таки привлекательный.

## 5.2 Установка приложения

Для запуска приложения необходимо выполнить следующие шаги:

1. Запустить серверную часть приложения, которая соединяет базу данных и Node приложение. Для этого необходимо запустить сервер Redis.

2. Запустить Node приложение, которое будет обрабатывать пользовательские запросы и взаимодействовать с сервером. Для этого необходимо запустить команду для сборки и запуска Node 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 мы рассмотрели, как пользоваться приложением. Рассмотрели, какими функциями обладает обычный пользователь, а какими функционалом обладает администратор. Рассмотрели, как администратор может редактировать элементы, где администратор может посмотреть содержимое всей базы данных данного приложения. Рассмотрели способ связи пользователя с поддержкой. Также рассмотрели, как пользователь может добавлять игры в список желаемого.

# Заключение

В процессе решения поставленной задачи была достигнута поставленная цель по созданию приложения «Сайт игр». Основой целью курсового проекта стало проектирование кроссплатформенного приложение, которое помогло облегчить взаимодействие с пользователем. Это было достигнуто за счёт гибкой программной платформы сервера на Node.js и адаптивности сайта, в качестве клиентской части.

При разработке выполнены следующие пункты:

* авторизация и регистрация пользователя;
* создание новых учетных записей пользователя;
* просмотр и оценка игр;
* предоставление возможности для создания заявки в поддержку;
* предоставление возможности доступа к таблицам БД администратору;

В курсовом проекте были реализованы следующие задачи:

* создание базы данных;
* создание серверной части приложения;
* создание клиенткой части с пользовательским интерфейсом;
* реализация функций приложения;
* тестирование программного продукта.

Данный проект является показательным примером программного продукта, реализующего интернет-магазин.

В соответствии с полученным результатом работы программы можно сделать вывод, что разработанная программа работает верно, а требования технического задания выполнены в полном объёме.

Тестирование проекта показало, что приложение функционирует правильно и все поставленные тесты прошло. В ходе тестирования ошибок не было найдено.

Данной приложение готово к эксплуатации и использованию в сети Интернет.

# Список используемых источников

1. MSSQL Сайт о программировании [Электронный ресурс] / Режим доступа: https://learn.microsoft.com/en-us/sql/?view=sql-server-ver16 – Дата доступа: 18.03.2024.
2. Все, что вы хотели знать о Sequelize [Электронный ресурс] / Режим доступа: https://habr.com/ru/post/566036/ – Дата доступа: 19.03.2024.
3. Model Instances [Электронный ресурс] – Режим доступа: https://sequelize.org/master/manual/model-instances.html – Дата доступа: 26.03.2024.
4. How to Build and Structure a Node.js MVC Application [Электронный ресурс] – Режим доступа: <https://www.sitepoint.com/node-js-mvc-application/> – Дата доступа: 01.04.2024.
5. How to Set Up a Websocket Server with Node.js and Express [Электронный ресурс] – Режим доступа: https://cheatcode.co/tutorials/how-to-set-up-a-websocket-server-with-node-js-and-express – Дата доступа: 28.04.2024.

# Приложение А

const { Sequelize, Op } = require('sequelize');

const sequelize = new Sequelize('GamerHub', 'student', 'fitfit', {

host: 'localhost',

dialect: 'mssql',

logging: false

});

sequelize

.authenticate()

.then(() => {

console.log('Database connection has been established successfully');

})

.catch((error) => {

console.error('Unable to connect to the database:', error);

});

const User = sequelize.define('Users', {

user\_id: {

type: Sequelize.INTEGER,

primaryKey: true,

autoIncrement: true

},

username: {

type: Sequelize.STRING,

allowNull: false

},

password: {

type: Sequelize.STRING,

allowNull: false

},

email: {

type: Sequelize.STRING,

allowNull: false

},

role: {

type: Sequelize.STRING,

allowNull: false

},

registration\_date: {

type: Sequelize.DATE,

allowNull: false,

defaultValue: Sequelize.NOW

},

avatar: {

type: Sequelize.BLOB('long'),

allowNull: true

}

}, {

tableName: 'Users',

timestamps: false

});

const Game = sequelize.define('Games', {

game\_id: {

type: Sequelize.INTEGER,

primaryKey: true,

autoIncrement: true

},

title: {

type: Sequelize.STRING,

allowNull: false

},

release\_date: {

type: Sequelize.DATE,

allowNull: false

},

developer: {

type: Sequelize.STRING,

allowNull: false

},

genre: {

type: Sequelize.STRING,

allowNull: false

}

}, {

tableName: 'Games', // Название таблицы

timestamps: false // Если нет столбцов created\_at и updated\_at

});

const Rating = sequelize.define('Rating', {

rating\_id: {

type: Sequelize.INTEGER,

primaryKey: true,

autoIncrement: true

},

game\_id: {

type: Sequelize.INTEGER,

allowNull: false

},

user\_id: {

type: Sequelize.INTEGER,

allowNull: false

},

rating: {

type: Sequelize.INTEGER,

allowNull: false

},

}, {

tableName: 'Ratings',

timestamps: false

});

const Wishlist = sequelize.define('Wishlist', {

wishlist\_id: {

type: Sequelize.INTEGER,

primaryKey: true,

autoIncrement: true

},

user\_id: Sequelize.INTEGER,

game\_id: Sequelize.INTEGER

}, {

tableName: 'Wishlist',

timestamps: false

});

const Notification = sequelize.define('Notifications', {

notification\_id: {

type: Sequelize.INTEGER,

primaryKey: true,

autoIncrement: true

},

message: {

type: Sequelize.TEXT,

allowNull: false

}

}, {

tableName: 'Notifications',

timestamps: false

});

const Feedback = sequelize.define('Feedback', {

feedback\_id: {

type: Sequelize.INTEGER,

primaryKey: true,

autoIncrement: true

},

message: {

type: Sequelize.TEXT,

allowNull: false

},

game\_id: {

type: Sequelize.INTEGER, // Тип поля может отличаться в зависимости от вашей базы данных

allowNull: false

}

}, {

tableName: 'Feedback',

timestamps: false,

associate: function(models) {

Feedback.belongsTo(models.User, { foreignKey: 'user\_id' });

}

});

User.hasMany(Rating, { foreignKey: 'user\_id' });

Rating.belongsTo(User, { foreignKey: 'user\_id' });

Game.hasMany(Rating, { foreignKey: 'game\_id' });

Rating.belongsTo(Game, { foreignKey: 'game\_id' });

User.hasMany(Wishlist, { foreignKey: 'user\_id' });

Wishlist.belongsTo(User, { foreignKey: 'user\_id' });

Game.hasMany(Wishlist, { foreignKey: 'game\_id' });

Wishlist.belongsTo(Game, { foreignKey: 'game\_id' });

User.hasMany(Feedback, { foreignKey: 'user\_id' });

Feedback.belongsTo(User, { foreignKey: 'user\_id' });

// Синхронизация моделей с базой данных

async function syncDatabase() {

try {

await sequelize.sync({ alter: true });

console.log('Таблицы успешно созданы и синхронизированы с базой данных');

} catch (error) {

console.error('Ошибка создания и синхронизации таблиц:', error);

}

}

module.exports = {

sequelize,

syncDatabase,

User,

Game,

Rating,

Wishlist,

Notification,

Feedback,

Op

};

Листинг 1 – Пример описания структуры базы данных

const redisClient = redis.createClient();

redisClient.connect().then(() => {

console.log('connect Redis');

}).catch((err) => {

console.log('connection error Redis:', err);

});

router.use(bodyParser.urlencoded({ extended: true }));

router.use(bodyParser.json());

router.use(cookieParser());

redisClient.on("ready", () => {

console.log("Connected!");

});

redisClient.on("error", (err) => {

console.log("Error in the Connection");

});

const generateAccessToken = (username, email, \_expiresIn = '24m', \_secret = secret) => {

const payload = {

username,

email

};

return jwt.sign(payload, \_secret, { expiresIn: \_expiresIn });

}

function decodeToken(token) {

try {

const decoded = jwt.decode(token);

return decoded;

} catch (error) {

console.error('Error decoding token:', error.message);

return null;

}

}

const authenticateToken = (req, res, next) => {

const token = req.cookies.accessToken;

if (!token) return res.sendStatus(401);

jwt.verify(token, 'access\_secret', (err, user) => {

if (err) return res.sendStatus(403);

req.user = user;

next();

});

};

const hashPassword = (password) => {

return crypto.createHash('sha256').update(password).digest('hex');

};

router.get('/logout', (req, res) => {

// Очистка куки с токенами

res.clearCookie('accessToken', { httpOnly: true, sameSite: 'strict' });

res.clearCookie('refreshToken', { httpOnly: true, sameSite: 'strict', path: '/' });

// Перенаправление на страницу входа

res.redirect('/login');

});

router.get('/resource', authenticateToken, async (req, res) => {

try {

const { username } = req.user;

const { refreshToken } = req.cookies;

jwt.verify(refreshToken, 'refresh\_secret', async (err, user) => {

if (err) {

return res.status(401).send('Unauthorized');

}

});

const result2 = await redisClient.get(`key${refreshToken}`);

if (`${username}` === result2) {

return res.status(401).send('Refresh token is in the BLACKLIST :)');

}

res.send(`Resource page. You're authorized. <br/> Username: ${username} Info: ${req.user.email}`);

} catch (error) {

console.error('Error accessing resource:', error);

return res.status(500).send('Internal Server Error');

}

});

router.get('/user-info', authenticateToken, (req, res) => {

const { username, email } = req.user;

User.findOne({ where: { username: username, email: email } })

.then(user => {

if (user) {

res.json(user);

} else {

res.status(404).send('User not found');

}

})

.catch(error => {

console.error('Error fetching user data:', error);

res.status(500).send('Server Error');

});

});

router.get('/refresh-token', async (req, res) => {

const existRefreshToken = req.cookies.refreshToken;

const decodedToken = decodeToken(existRefreshToken);

if (existRefreshToken) {

jwt.verify(existRefreshToken, 'refresh\_secret', async (err, user) => {

if (err) {

return res.status(401).send('Unauthorized');

} else if (user) {

const usrname = user.username;

const Banned = await redisClient.get(`key${existRefreshToken}`);

if (`value ${usrname}` === Banned) {

return res.status(401).send('Refresh token in black list');

}

const newAccessToken = generateAccessToken(decodedToken.username, decodedToken.email, '10m', 'access\_secret');

const newRefreshToken = generateAccessToken(decodedToken.username, decodedToken.email, '24h', 'refresh\_secret');

res.clearCookie('accessToken');

res.clearCookie('refreshToken');

res.cookie('accessToken', newAccessToken, { httpOnly: true, sameSite: 'strict' });

res.cookie('refreshToken', newRefreshToken, { httpOnly: true, sameSite: 'strict', path: '/' });

await redisClient.set(`key${existRefreshToken}`, `${usrname}`);

console.log('ADD refresh token in black list', await redisClient.get(`key${existRefreshToken}`));

console.log('NEW refreshToken ' + newRefreshToken);

res.status(200).json({ accessToken: newAccessToken });

}

});

} else {

res.status(400).send('No refresh token provided');

}

});

router.post('/login', (req, res) => {

const { username, password } = req.body;

const hashedPassword = hashPassword(password);

User.findOne({ where: { username: username } })

.then(user => {

if (user && user.password === hashedPassword) {

const accessToken = generateAccessToken(user.username, user.email, '10m', 'access\_secret');

const refreshToken = generateAccessToken(user.username, user.email, '24h', 'refresh\_secret');

res.cookie('accessToken', accessToken, { httpOnly: true, sameSite: 'strict' });

res.cookie('refreshToken', refreshToken, { httpOnly: true, sameSite: 'strict', path: '/' });

res.status(200).json({ accessToken, refreshToken });

} else {

res.status(401).send("Incorrect username or password");

}

})

.catch(err => {

console.error('Error:', err);

res.status(500).send('Internal Server Error');

});

});

router.post('/register', (req, res) => {

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

const role = 'user';

const registrationDate = new Date();

const hashedPassword = hashPassword(password);

User.findOne({ where: { username: username } })

.then(existingUser => {

if (existingUser) {

res.status(400).send('Username already taken');

} else {

User.create({

username: username,

password: hashedPassword,

email: email,

role: role,

registration\_date: registrationDate

})

.then(user => {

res.status(200).redirect('/login');

})

.catch(err => {

console.error('Error registering user:', err);

res.status(500).send('Error registering user');

});

}

})

.catch(err => {

console.error('Error checking existing user:', err);

res.status(500).send('Error checking existing user');

});

});

router.use(function (err, req, res, next) {

res.send(err.message);

});

module.exports = router;

Листинг 2 – Контроллер авторизации/регистрации пользователя

totalPages = null;

function addGames(data) {

const gameInfo = document.getElementById('gameInfo');

gameInfo.style.display = 'flex'; // Добавьте стиль display для использования Flexbox

gameInfo.style.flexWrap = 'wrap'; // Разрешить перенос блоков на новую строку

while (gameInfo.firstChild) {

gameInfo.removeChild(gameInfo.firstChild);

}

data.games.forEach(game => {

const gameLink = document.createElement('a');

gameLink.href = '/game?gameTitle=' + encodeURIComponent(game.gameTitle);

gameLink.classList.add('game-link');

const gameElement = document.createElement('div');

gameElement.classList.add('game');

gameElement.style.width = '250px';

gameElement.style.border = '1px solid black';

gameElement.style.padding = '15px';

gameElement.style.textAlign = 'center';

const lastImage = game.image;

if (lastImage) {

const gameImage = document.createElement('img');

gameImage.src = lastImage;

gameImage.alt = 'Game Image';

gameImage.style.width = '200px';

gameImage.style.height = '200px';

gameImage.style.margin = '10px';

gameImage.style.border = '1px solid black';

gameElement.appendChild(gameImage);

} else {

const gameImage = document.createElement('img');

gameImage.src = 'img/noImageIcon.png ';

gameImage.alt = 'Game Image';

gameImage.style.width = '200px';

gameImage.style.height = '200px';

gameImage.style.margin = '10px';

gameImage.style.border = '1px solid black';

gameElement.appendChild(gameImage);

}

gameLink.appendChild(gameElement);

gameInfo.appendChild(gameLink);

const gameTitle = document.createElement('p');

gameTitle.textContent = game.gameTitle;

gameElement.appendChild(gameTitle);

});

}

window.addEventListener('DOMContentLoaded', (event) => {

fetch('/gamesList')

.then(response => response.json())

.then(data => {

if (data.error) {

console.error('Error:', data.error);

} else {

addGames(data);

}

})

.catch(error => {

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

});

async function getGameCountAndPages() {

try {

const response = await fetch('/countGames');

const data = await response.json();

const count = data.count;

totalPages = Math.ceil(count / 15);

createPageButtons(totalPages, 1); // Создаем кнопки для первой страницы

} catch (error) {

console.error('Ошибка при получении общего количества игр:', error);

}

}

function createPageButtons(totalPages, currentPage) {

const paginationButtons = document.getElementById('paginationButtons');

paginationButtons.innerHTML = ''; // Очистить содержимое элемента

const maxButtons = 8; // Максимальное количество кнопок для отображения

// Определение первой и последней страницы для отображения

let startPage, endPage;

if (totalPages <= maxButtons) {

// Если общее количество страниц меньше или равно максимальному количеству кнопок,

// отображаем все страницы

startPage = 2;

endPage = totalPages - 1;

} else {

// Если общее количество страниц больше максимального количества кнопок,

// вычисляем первую и последнюю страницы для отображения

if (currentPage <= (Math.floor(maxButtons / 2) + 1)) {

// Если текущая страница ближе к началу, отображаем первые maxButtons страниц

startPage = 2;

endPage = maxButtons + startPage;

} else if (currentPage + Math.floor(maxButtons / 2) >= totalPages) {

// Если текущая страница ближе к концу, отображаем последние maxButtons страниц

startPage = totalPages - (maxButtons + 1);

endPage = totalPages - 1;

} else {

// В остальных случаях отображаем страницы вокруг текущей страницы

startPage = currentPage - Math.floor(maxButtons / 2);

endPage = currentPage + Math.floor(maxButtons / 2);

}

}

const firstButton = document.createElement('button');

firstButton.textContent = '1';

firstButton.classList.add('pagesButton'); // Добавить класс "button"

firstButton.addEventListener('click', function() {

goToPage(1);

});

paginationButtons.appendChild(firstButton);

for (let i = startPage; i <= endPage; i++) {

if (i !== 1 && i !== totalPages) {

const button = document.createElement('button');

button.textContent = i;

button.classList.add('pagesButton'); // Добавить класс "button"

button.addEventListener('click', function() {

goToPage(i);

});

if (i === currentPage) {

button.classList.add('active'); // Добавить класс "active" для текущей страницы

}

paginationButtons.appendChild(button); // Добавить кнопку в элемент

}

}

const lastButton = document.createElement('button');

lastButton.textContent = totalPages;

lastButton.classList.add('pagesButton'); // Добавить класс "button"

lastButton.addEventListener('click', function() {

goToPage(totalPages);

});

paginationButtons.appendChild(lastButton);

const pageInput = document.createElement('input');

pageInput.type = 'number';

pageInput.min = 1;

pageInput.max = totalPages;

pageInput.value = currentPage;

pageInput.classList.add('page-input'); // Добавить класс "page-input"

paginationButtons.appendChild(pageInput);

pageInput.addEventListener('input', function() {

let pageNumber = parseInt(pageInput.value);

if (pageNumber < 1) {

pageNumber = 1;

} else if (pageNumber > totalPages) {

pageNumber = totalPages;

}

pageInput.value = pageNumber;

});

// Создание кнопки для перехода на указанную страницу

const goButton = document.createElement('button');

goButton.textContent = 'Go';

goButton.classList.add('go-button'); // Добавить класс "go-button"

goButton.addEventListener('click', function() {

const pageNumber = parseInt(pageInput.value);

if (pageNumber >= 1 && pageNumber <= totalPages) {

goToPage(pageNumber);

} else {

console.error('Invalid page number');

}

});

paginationButtons.appendChild(goButton);

}

function goToPage(page) {

console.log('Переход на страницу:', page);

createPageButtons(totalPages, page)

// Отправить запрос на сервер с указанием номера страницы

fetch(`/gamesList?page=${page}`)

.then(response => response.json())

.then(data => {

addGames(data)

})

.catch(error => {

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

});

}

getGameCountAndPages()

});

Листинг 3 – Скрипт для загрузки игр