МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ

ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных технологий

Кафедра Информационные системы и технологии

Специальность 1–40 01 01 Программное обеспечение информационных технологий

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**К КУРСОВОЙ РАБОТЕ НА ТЕМУ:**

«Веб-приложение для цифрового распространения видеоигр»

Выполнил студент Окулич Дмитрий Юрьевич

(Ф.И.О.)

Руководитель работы ст. преп. Нистюк Ольга Александровна

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

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

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

Курсовая работа защищена с оценкой

Минск 2024

**Содержание**

[Введение 3](#_Toc165501966)

[1 Постановка задачи 4](#_Toc165501967)

[1.1 Аналитический обзор аналогов 4](#_Toc165501968)

[1.1.1 Steam 4](#_Toc165501969)

[1.1.2 GOG COM 5](#_Toc165501970)

[1.1.3 Вывод 6](#_Toc165501971)

[1.2 Разработка функциональных требование и вариантов использования 6](#_Toc165501972)

[1.3 Вывод 7](#_Toc165501973)

[2 Проектирование web-приложения 8](#_Toc165501974)

[2.1 Обобщенная структура управлением приложения 8](#_Toc165501975)

[2.2 Диаграммы UML, взаимосвязь всех компонентов. 8](#_Toc165501976)

[2.3 Вывод 9](#_Toc165501977)

[3 Разработка web-приложения 10](#_Toc165501978)

[3.1 Разработка клиентской части web-приложения 10](#_Toc165501979)

[3.2 Разработка серверной части web-приложения 16](#_Toc165501980)

[Вывод 21](#_Toc165501981)

[Приложение А 23](#_Toc165501982)

[Приложение Б 28](#_Toc165501983)

[Приложение В 29](#_Toc165501984)

# Введение

Цель данного курсового проекта заключается в создании веб-приложение, которое позволит пользователям покупать и скачивать видео игры, а разработчикам создавать кампании и загружать свои видео игры.

Для разработки серверной части данного веб-приложения был выбран фреймворк ExpressJS с языком TypeScript, который позволит разработать гибкое и многофункционально веб-приложение; а для разработки клиентской части была выбран фреймвокр ReactJS совместно с Redux. Так же в качестве протоколов обмена информации используются HTTPS и WebSocket. Для работы с БД будет использоваться модуль Prisma, который позволит создать абстракцию над объектами базы данных PostgreSQL.

Для гарантированной безопасности пользователей приложения в курсовой работе применяется метод шифрования паролей перед их сохранением в базу данных. Также для обеспечения функциональности приложения используются мультимедийные форматы данных при сохранении аудио и картинок.

Основные требования к приложению:

обеспечивать возможность регистрации и авторизации;

поддерживать роли администратора и пользователя, разработчика;

давать возможность разработчикам загружать свои игры;

давать возможность скачивание игры из библиотеки пользователя;

давать возможность пользователю добавить игру в свою библиотеку;

давать возможность оставлять комментарии под игрой;

обеспечивает возможность оценки игры.

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

## 1 Постановка задачи

## 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.

# Steam

Steam это на данный момент самая известная площадка по распространению видео игр созданная компанией Valve. Количество активных пользователей превышает 125 миллионов. Поэтому будет справедливым рассмотреть Steam как аналог.

Steam широким функционалом, помимо возможности покупки и установки игр, так же обладает системой друзей, чата и сообщества. Можно отслеживать играет ли ту или иную игру ваш друг. Steam имеет систему торговой площадки, где игроки могут продавать или покупать видео игровые вещи. Для разработчиков имеется система мастерской, для каждой отдельной игры, благодаря которой моддинг игр можно автоматизировать. Так же из-за того, что Steam является не веб-приложением, а десктоп-приложением все игры устанавливаются на прямую и не требуют дополнительный действий, но в тоже время проблемой является то, что сам Steam требуется устанавливать на своё устройство.

Интерфейс Steam имеет минималистичный и удобный дизайн, благодаря которому весьма удобно ориентироваться в приложении. Скриншот Steam предоставлен на рисунке 1.1.



Рисунок 1.1 - Приложение «Steam»

Следуя из вышеописанного, можно отметить следующие преимущества и недостатки.

Преимущества:

* удобный дизайн;
* система друзей;
* система торговой площадки;
* система мастерской;
* установка игр не требует дополнительный действий.

Недостатки:

* требует установки на устройство.

# GOG COM

GOG COM это площадка по распространению видео игр созданная компанией CD Project. Основная особенность этой площадки — отсутствие DRM защиты на играх. Купив игру, вы купили не лицензию на её использование, а установочный файл. Поэтому GOG является весьма востребованным даже приличии такого гегемона как Steam. Из этого следует что GOG нужно рассмотреть, как аналог.

GOG предоставляет возможность: покупки игры, скачивания игры, управление списком друзей. Так же имеются разные социальные взаимодействия.

GOG имеет современный и минималистичный дизайн который не мешает ориентации по приложению. Скриншот GOG COM предоставлен на рисунке 1.2.

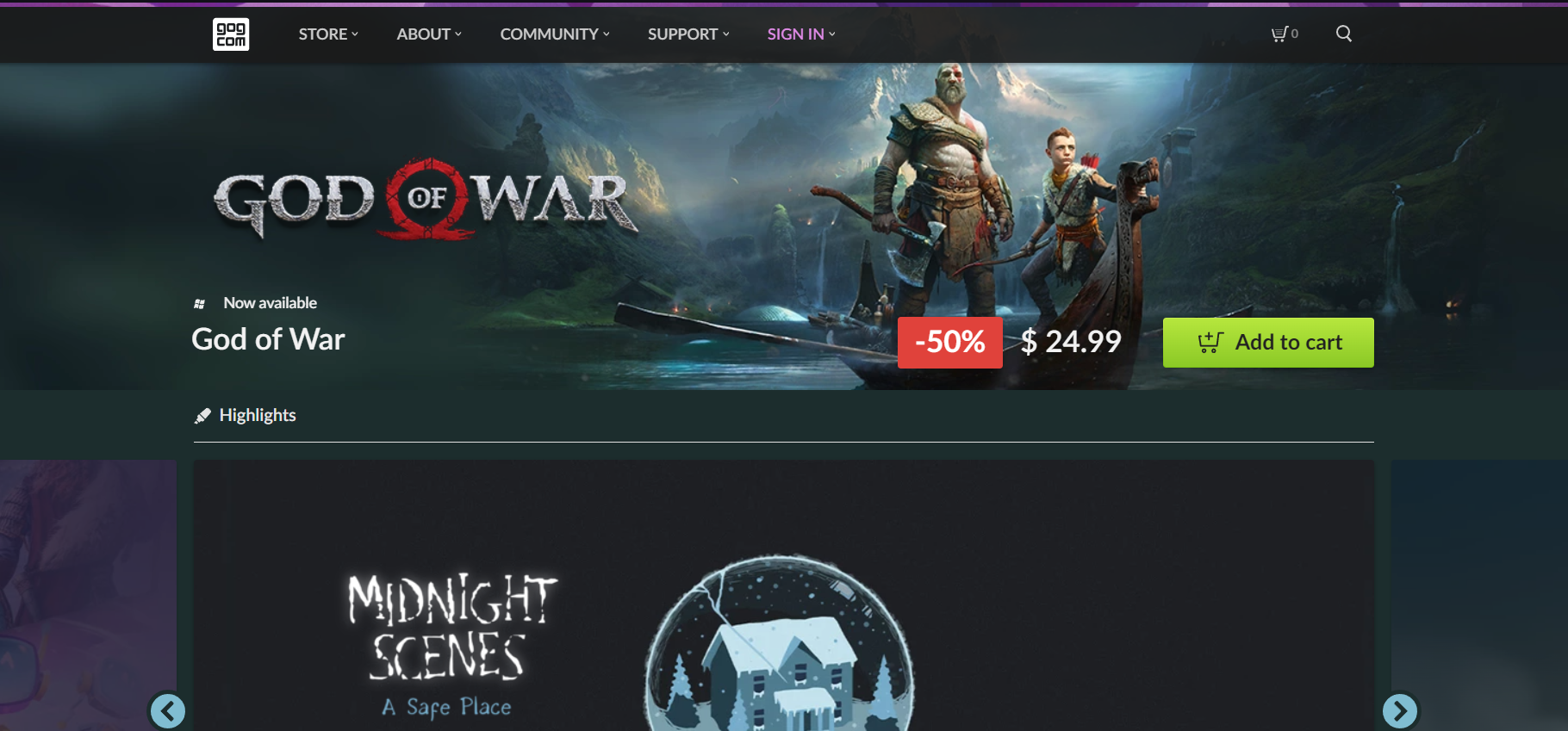


Рисунок 1.2 - web-приложение GOG COM

Следуя из вышеописанного, можно отметить следующие плюсы и минусы.

Преимущества:

* удобный дизайн;
* отсутствие DRM защиты;
* система друзей.

Недостатки:

* относительно мало функционален.

# 1.1.3 Вывод

Было рассмотрено два аналогичных решения Steam и GOG COM. Исследовав аналоги было принято решение что приложение по количеству функционала будет ориентироваться на Steam, а от GOG будет взята философия отсутствия DRM защиты.

# Разработка функциональных требование и вариантов использования

Приложение для распространения игр должно обеспечивать функционал для регистрации авторизации, который позволит идентифицировать пользователей и для каждого пользователя предлагать игры. Также приложение должно обеспечивать функционал отправки различных типов сообщений: текстовые, файловые. Для работы с рецензиями должны быть созданы функции удаления и редактирования рецензий. Удаление должно быть сделано полным, чтобы рецензия полностью удалялось из базы данных. Для покупки игры пользователь должен будет заполнить форму оплаты. Пользователь сможет скачать купленные им игры зайдя в свою библиотеку. Для оплаты пользователь может использовать встроенный кошелёк или привязать карту.

Разработчики будут иметь функционал для работы с играми. Приложение должно давать следующие возможности для работы с играми: загрузка, удаление, обновление, изменение.

Диаграмма вариантов использования представлена на рисунке 1.3.

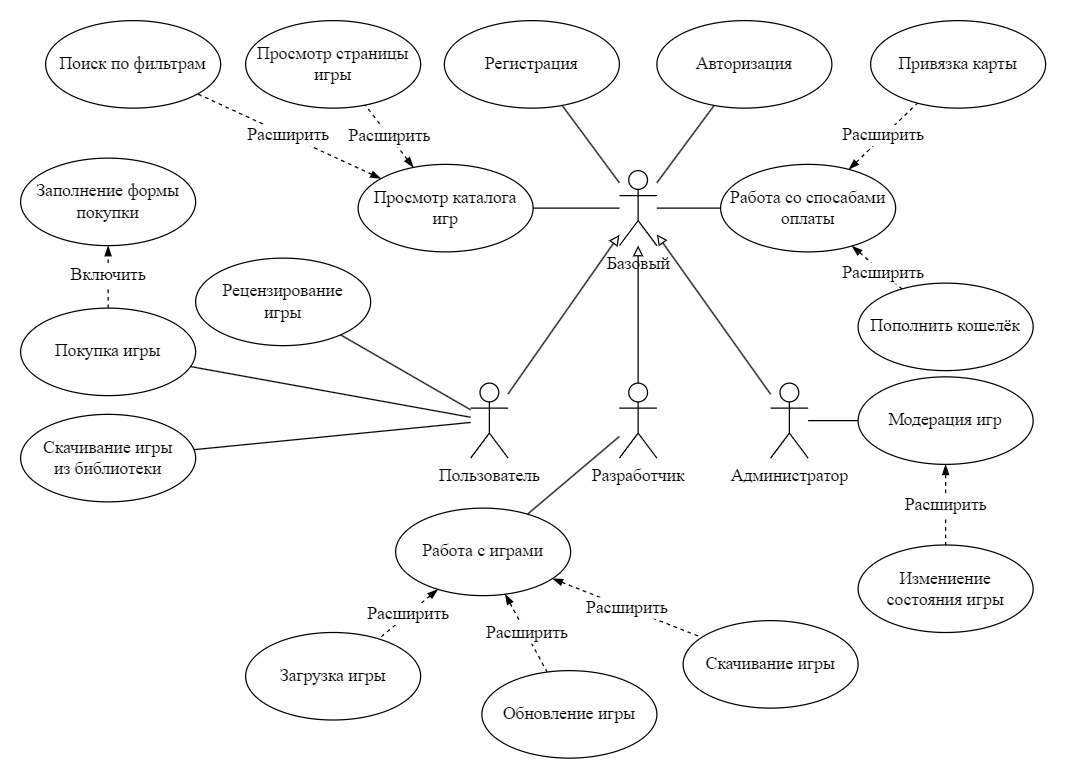


Рисунок 1.3 - use-case диаграмма

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

# Вывод

В данном разделе был рассмотрен 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.

# 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 для общения должны входить функции для удобного поиска игры по заданным фильтрам, функция покупки игры, различные виды сообщение и другие подобные функции.

## 2.2 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 базы данных представлена на рисунке 2.1.

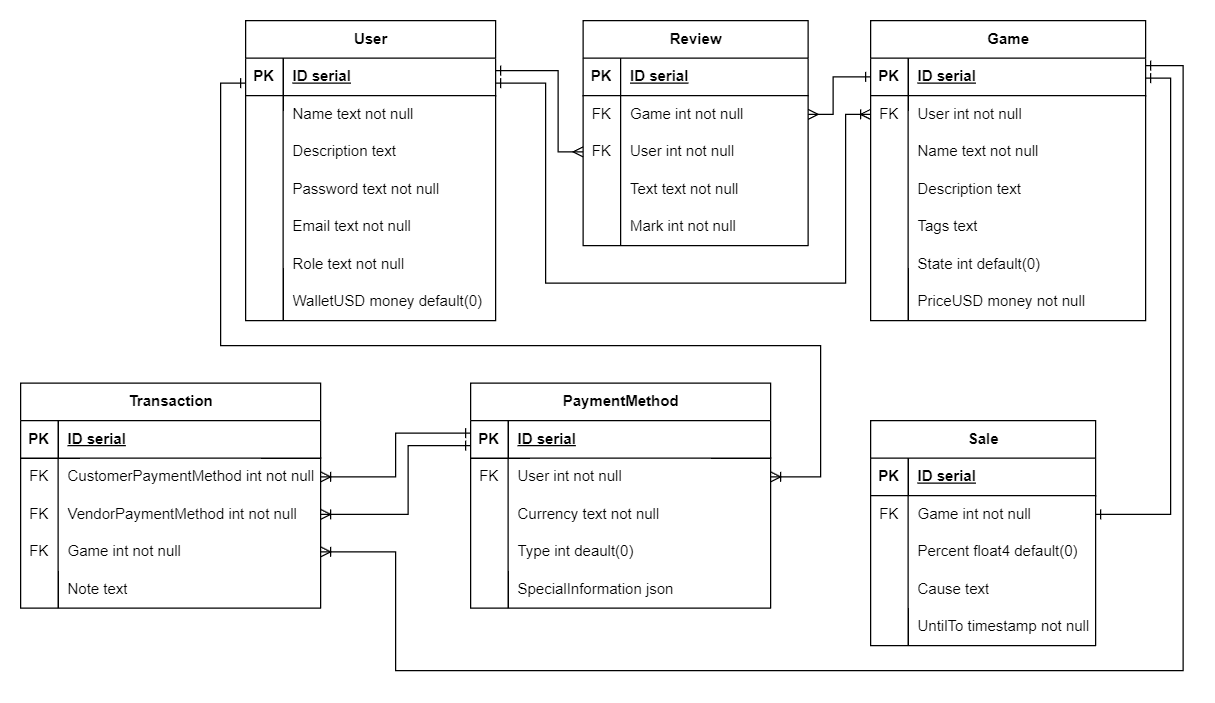


Рисунок 2.1 - Диаграмма базы данных

Таким образом, диаграмма показывает связи между таблицами и полями, а также отношения между ними, такие как связи "один-ко-многим" и "один-ко-одному". Например, таблица User связана с таблицами PaymentMethod, Review, Game.

Для того понимания структуры данного приложения и как его разворачивать нам может помочь диаграмма развертывания. Данная диаграмма показана на рисунке 2.2.

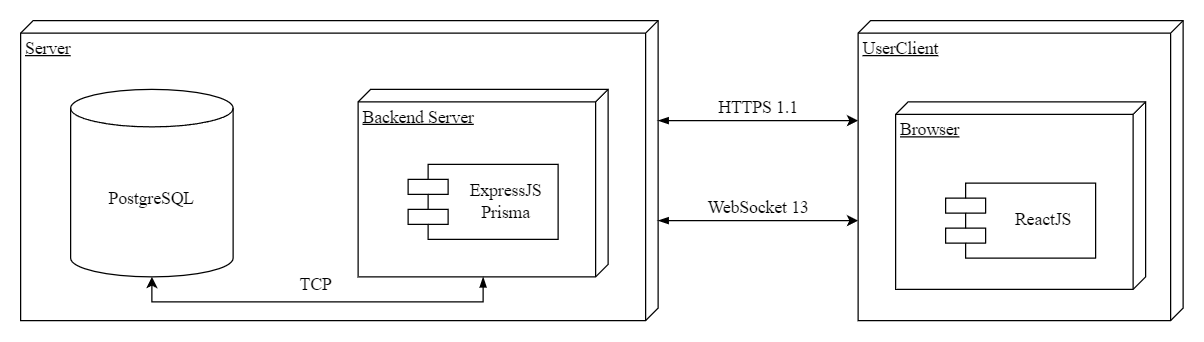


Рисунок 2.2 - Диаграмма развертывания

По данной диаграмме развертывания видно, что наше веб-приложение состоит из трех основных частей. Первая часть – это клиентская часть нашего приложения. Эта часть реализована при помощи одноименной библиотеки для языка программирования JavaScript ReactJS. Данная фреймворк поможет создать динамически SPA приложение, которое будет делать запросы по протоколу HTTPS к серверной части приложения. Также оно будет отправлять и получать сообщения по протоколу WebSocket. Вторая часть данного веб-приложения – это серверная часть, разработанная на платформе NodeJS при помощи фреймворка ExpressJS, который позволяет построить структуру из всех основных модулей нашего приложения. Данная часть приложения будет обращаться к базе данных по протоколу TCP. Обращение к базе данных будет упрощено за счет использования ORM модуля – Prisma, который позволит нам создать абстракцию над объектами базы данных. Третья часть нашего приложения – это сам сервер базы данных PostgreSQL, который будет отвечать за хранение всех данных, способствующих работе нашего приложения.

## 2.3 Вывод

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 PostgreSQL, ее основные сущности и как они связаны. Также рассмотрели, какие есть основные части нашего веб-приложения, точнее, во что оно развертывается.

# 3 Разработка web-приложения

## 3.1 Разработка клиентской части web-приложения

Клиентская часть – главная часть приложения. Она должна быть интуитивно понятна и интерактивна. Клиентская часть будет состоять из трёх основных частей: библиотека React, React Redux и WebSocket клиент. Для отправки запросов на сервер используется стандартная Fetch API.

Часть React состоит из восьми страниц и главного компонента «App». Когда пользователь заходит первый раз в веб-приложение ему доступна лишь одна страница – страница каталога. В библиотеки React для маршрутизации используется компонент Router. Код компонента «App» с настроенным компонентом Router предоставлен в листинге 3.1.

<div className="BaseContainer">

<header>

<Header />

</header>

<main>

<BrowserRouter>

<Routes>

<Route path="/" element={<Catalog />} />

<Route path="/game/\*" element={<GamePage />} />

<Route path="/user" element={<User />} />

<Route path="/library" element={<Library />} />

<Route path="/developer" element={<DeveloperGameList />} />

<Route path="/developer/createGame" element={<DeveloperCreateGame />} />

<Route path="/developer/editGame/\*" element={<DeveloperGameEditor />} />

<Route path="/admin" element={<AdminPanel />} />

</Routes>

</BrowserRouter>

</main>

<footer>

<Footer />

</footer>

</div>

Листинг 3.1 – Настройка компонента Router

Для каждой из роли были выделены свои страницы. Так, например станицы «/developer», «/developer/createGame» и «/developer/editGame/\*» относятся исключительно к функционалу разработчика, если же пользователь с другой ролью попробует перейти на эту страницу, то его сразу выкинет на страницу каталога. Общими же являются страницы: «/», «/game/\*», «/user». Так же в компоненте «App» используется React Redux. Пример использования Redux в компоненте «App» предоставлен в листинге 3.2.

const dispatch = useDispatch();

fetch("/api/auth").then(raw => raw.json())

.then(data => {

if (data.auth) {

dispatch(setUser({ init: true, auth: true, data: data.data })); } else {

dispatch(setUser({ init: true, auth: false, data: undefined })); }});

Листинг 3.2 – Использование Redux в компоненте «App»

Redux в данном примере используется через хук «useDispatch» так как нам требуется записать данные пользователя. Тут идёт проверка авторизации пользователя если он был заранее авторизован. Если пользователь авторизовывался, то данные пользователя запишутся в глобальное состояние «User», а если же пользователь не был авторизован или возникли другие проблемы, то тогда будет помечено что пользователь не авторизован.

Пока пользователь не авторизован ему доступны только две страницы: «/», «/game/\*».

Запрос «/» приставляет собой страницу каталога, где пользователь может находит игры по названию и по заданным фильтрам. Так же страница поддерживает нумерацию страниц для того, чтобы не выводить все найденные игры на одну страницу. Это необходимо для экономии ресурсов сервера и клиента. Для получения игр с сервера используется fetch запрос, предоставленный в листинге 3.3.

let [games, setGames] = useState([]);

let [pagesCount, setPagesCount] = useState(0);

const getGames = useCallback(() => {

fetch('/api/catalog/getgames', {

method: "POST",

headers: {

"content-type": "application/json"

},

body: JSON.stringify({

namePattern: namePattern,

page: page,

filters: filters

}) })

.then(raw => raw.json())

.then(data => {

if (data.ok){

page = 1;

setPagesCount(data.pages);

setGames(data.games);

} });});

Листинг 3.3 – Fetch запрос для игр

Для хранения игр и количества страниц используются React состояния через функцию useState(). Так же видно, что fetch запрос отправляет следующие необходимые данные для поиска: паттерн названия, номер текущей страницы, значение фильтров.

Запрос «/game/\*» представляет страницу конкретной игры. В этой странице отображается вся информация об игре её название, описание, теги, разработчик, отзывы и т.д. В этой странице для реализации динамической погрузки отзывов был использован WebSocket. Код использования WebSocket предоставлен в листинге 3.4.

let [game, setGame] = useState(undefined);

let [reviewSocket, setReviewSocket] = useState(undefined);

let [reviews, setReviws] = useState([]);

let user = useSelector(state => state.user);

const reviewSocketMain = useCallback(() => {

let socket = new WebSocket(`wss://localhost:5000/game/reviews/${game.id}`);

socket.onopen = (evt) => {

setReviewSocket(socket);

}

socket.onmessage = (evt) => {

let reviews = JSON.parse(evt.data);

setReviws(reviews);

}

socket.onclose = (evt) => {

setReviewSocket(undefined);

}

});

Листинг 3.3 – Компонент Header

После создание WebSocket он ожидает получения отзывов от сервера после чего отзывы записываются в состояние. Так экземпляр сокета храниться как отдельное состояние. На данной странице происходит покупка игры если пользователь авторизован и имеет роль «User». Для этого страница получает данные о пользователе из глобального состояния при помощи Redux, а именно его хука «useSelector». Так же пользователь может оставить отзыв к игре если она есть в его библиотеке.

Авторизация и регистрация вынесена в компонент «Header» который представляет собой компонент шапки для всех страниц. Код предоставлен в листинге 3.4.

<div className="header-container" id="site\_header">

<nav>

...

<div className="user">

<div style={{ marginTop: "28px" }}>

{

user.auth ?

(

<Stack>

<span><a href="/user"> {user.data.name} </a></span>

<Stack direction={"row"} justifyContent={"end"}><h5>$ {user.data.walletusd}</h5></Stack>

</Stack> ):(

<span>

<a href="#site\_header" onClick={registerOpen}>Регистрация</a>

/

<a href="#site\_header" onClick={loginOpen}>Вход</a>

</span> ) }

</div>

...

</div>

</nav>

<RegisterDialog open={regDialogOpen}onCloseClick={registerClose}/>

<LoginDialog open={loginDialogOpen} onCloseClick={loginClose} />

</div>

Листинг 3.4 – Компонент Header

По листингу видно, что в зависимости от того авторизован ли пользователь выводиться разная информация. Для самой регистрации и авторизации служат два других компонента «RegisterDialog» и «LoginDialog». Они представляют собой диалоговые окна, где выводиться необходимая для регистрации или авторизации форма.

Запрос «/user» представляет собой страницу пользователя, которая разделена на две логические части: «Редактирование данных пользователя», «Редактирование способов оплаты». Пользователь может изменить своё имя, описание и установить новую иконку, так же пользователь может удалить свой аккаунт и привязать карту. Для передачи изображения и других данный используется fetch запрос, в котором передаются данные формы. Код предоставлен в листинге 3.5

const editUser = () => {

let userIcon = document.getElementById("userIcon");

let nickname = document.getElementById("userNickname");

let formData = new FormData();

formData.append('userIcon', userIcon.files[0]);

formData.append('userNickname', nickname.value);

formData.append('userDescription', txtAreaValue);

fetch("/api/edituser", {

method: "PUT",

body: formData

})

.then(raw => raw.json())

.then(data => {

if (data.ok) {

window.location.reload();

}

else {

if (data.error === "jwt") {

window.location.replace("/");

}

}

})

.catch(err => console.log(err));

}

Листинг 3.5 – Отправка данные пользователя через fetch

Использование формы необходимо для передачи файла и последующем принятии его на стороне сервера.

Запрос «/library» представляет собой страницу библиотеки пользователя, игре отображаются все игру которые были куплены либо добавлены пользователем. Так же пользователь может скачивать эти игры.

Запрос «/developer» представляет собой страницу для пользователя с ролью «Developer» где отображаются игры, загруженные этим пользователем.

Запрос «/developer/createGame» представляет собой страницу для пользователя с ролью «Developer» где пользователь создаёт и загружает игру. Для загрузки данных игры в лице иконки, изображения для каталога, изображения для страницы с игрой и файла с самой игрой используется fetch с данными формы. Код запроса предоставлен в листинге 3.6.

const uploadGame = () => {

let formData = new FormData();

formData.append('name', step0Data.name);

formData.append('description', step0Data.description);

formData.append('tags', step0Data.tags);

formData.append('price', step0Data.isfree ? 0 : step0Data.price);

formData.append('files', step1Data.icon[0]);

formData.append('files', step1Data.catalog[0]);

formData.append('files', step1Data.library[0]);

formData.append('files', step2Data.mainFile[0]);

fetch('/api/uploadgame', {

method: "POST",

body: formData

})

.then(raw => raw.json())

.then(data => {

if (data.ok) {

window.location.replace('/developer');

}

else {

if (data.error === 'jwt')

window.location.replace('/');

else

console.log(data.error);

}

});

}

Листинг 3.6 – Отправка данные об игре через fetch

Тут форма необходима для отправки файлов для последующего принятия их на стороне сервера.

Запрос «/developer/editGame» представляет собой страницу для пользователя с ролью «Developer» где пользователь изменяет информацию об игре. Так тут пользователь может загрузить новую иконку, изображение каталога, страницы, и новый файл с игрой. Так же, как и в «/developer/createGame» для отправки файлов используется fetch с данными формы. Для хранения данных о текущей игре используется глобальное состояние при помощи Redux.

Запрос «/admin» представляет собой страницу для пользователя с ролью «Admin» где пользователь изменят состояние игр. На данной странице подгружается список игр, который может быть отсортирован по состоянию. Администратор изменяет состояние этих игр. Для изменения состояния отправляется специальный fetch запрос, предоставленный в листинге 3.7.

const setState = useCallback((state) => {

fetch('api/setgamestate', {

method: "PUT",

headers: {

"content-type": "application/json"

},

body: JSON.stringify({ id: game.id, state: state })

})

.then(raw => raw.json())

.then(data => {

if (data.ok) {

onChangeGameState();

}

else if (data.error === 'jwt')

window.location.replace('/');

})

});

Листинг 3.6 – Запрос на изменение состояния игры

По запросу видно, что отправляется идентификатор игры и его новое состояние.

## 3.2 Разработка серверной части web-приложения

Серверная часть написана на фреймворке Express JS для платформы NodeJS. Данный фреймворк позволил разработать множество компонентов и собственную структуру сервера.

Для дальнейшего упрощения разработки был разработан собственный MVC. Он состоит из трёх основных классов «MVCManager», «MVCController», «MVCRoute».

MVCManager – основной класс который хранит в себе набор контролеров и зависимостей, так же этот класс связывает routes контролера с Express.

MVCController – класс контролера который хранит в себе на набор из MVCRoute. Через этот класс можно получить доступ к зависимостям.

MVCRoute – класс хранящий данные о конкретном route, его: url, action, controller, isWebSocket, method.

Реализация этих трех классов предоставлена в Приложении А.

Так же для работы MVC был разработы специальный enum «MVCRouteMethod» с поддерживаемыми методами GET, POST, PUT, DELETE. Ещё были разработаны специальные декораторы для более простой конфигурации контролёров и рутов. Код некоторых декораторов предоставлен в листинге 3.7.

function Controller(constructor: Function) {

MVCManager.LinkedControlles.push(constructor.prototype);

}

function MapRoute(url: string, mvcmethod: MVCRouteMethod) {

return function (target: MVCController, method: string, descriptor: PropertyDescriptor) {

MVCController.LinkedRoutes.push(MVCRoute.CreateB(url, mvcmethod, descriptor.value, target.constructor.name));

}

}

function MapGet(url: string) {

return function (target: MVCController, method: string, descriptor: PropertyDescriptor) {

MVCController.LinkedRoutes.push(MVCRoute.CreateB(url, MVCRouteMethod.GET, descriptor.value, target.constructor.name));

}

}

Листинг 3.7 – Декораторы для MVC

Для подключения и работы с базой данный и используется ORM Prisma Js. Для взаимодействия с базой данных был разработан класс «DataBase» который создаёт экземпляр объекта PrismaClient для установки соединения с базой данных PostgreSQL. Так же DataBase содержит в себе методы, которые облегчают взаимодействие с БД, пример таких методов предоставлен в листинге 3.8.

async GetUser(id: number) {

return await this.Instance.user.findFirst({ where: { id: id } });

}

async GetGame(id: number) {

return await this.Instance.game.findFirst({

where: { id: id },

include: { sale\_sale\_gameTogame: true, review\_review\_gameTogame: true }

});

}

Листинг 3.8 – Пример методов из класса DataBase

Для хеширования пароля был разработан специальный класс «PassowordHasher», который содержит метод для хеширования текста алгоритмом SHA2-256. Данный класс используется для хеширования пароля пользователя во время регистрации. Код класса предоставлен в листинге 3.9.

class PasswordHasher {

public salt: string

constructor(salt: string) {

this.salt = salt;

}

HashPassword(password: string): string {

const hash = crypto.createHash('sha256');

hash.update(password + this.salt);

const hashedPassword = hash.digest('hex');

return hashedPassword;

}

}

Листинг 3.8 – Пример методов из класса DataBase

Классом, который создаёт экземпляр HTTPS сервера является класс «Server». Так же класс создаёт Express приложение и добавляет к нему различные middleware такие как: Поддержка json, cookie parser, поддержка сессии, поддержка статических файлов. В этом классе происходит настройка Multer для загрузки файлов на сервер, а также создаётся экземпляр WebSocket для сервера HTTPS под приложения Express. Листинг класса предоставлен в приложении Б.

Для работы с jwt токенами был разработан специальный класс «JwtManager». Он предоставляет специальные методы для работы с jwt access и refresh токенами. Методы включают такой функционал как: создание токена, проверка токена. Пример кода класса JwtManager предоставлен в листинге 3.9.

class JwtManager {

public accessSecret: string;

public refreshSecret: string

constructor(accessSecret: string, refreshSecret: string) {

this.accessSecret = accessSecret;

this.refreshSecret = refreshSecret;

}

GenerateAccessToken(userId: number): string {

return jwt.sign({ userId }, this.accessSecret, { expiresIn: '20min' });

}

GenerateRefreshToken(userId: number): string {

return jwt.sign({ userId }, this.refreshSecret, { expiresIn: '60min' });

}}

...

}

Листинг 3.9 – Пример кода класса JwtManager

Для аутентификации пользователя на стороне сервера был разработан специальный класс «AuthService». Данный класс предоставляет метод для проверки подлинности данных из jwt токенов, код метода предоставлен в листинге 3.10.

async Auth(req: Request, res: Response): Promise<user> {

let ajwt;

if (!this.jwt.IsValidAccessToken(req.cookies.ajwt)) {

if (!this.jwt.IsValidRefreshToken(req.cookies.rjwt))

throw "jwt";

let rjwtdata = this.jwt.AuthenticateRefreshToken(req.cookies.rjwt) as JwtPayload;

let user = await this.db.Instance.user.findFirst({ where: { id: rjwtdata.userId } });

if (!user || user.rjwt !== req.cookies.rjwt)

throw "jwt";

ajwt = this.jwt.GenerateAccessToken(rjwtdata.userId);

res.cookie("ajwt", ajwt);

}

else

ajwt = req.cookies.ajwt;

let jwtdata = this.jwt.AuthenticateAccessToken(ajwt) as JwtPayload;

let user = await this.db.GetUser(jwtdata.userId);

if (user)

return user;

else

throw "User is not exist";

}

Листинг 3.10 – Метод Auth

Метод «Auth» проверяет jwt токены если с ними что не так-то выбрасывает исключение. Если же аутентификация успешна, то метод вернёт объект с данными пользователя.

Для обработки HTTPS запросов от клиентской части приложения были разработаны соответствующие контролеры. Для поддержки контролеров на сервере используется заранее разработанный MVC. Например, котроллер «GamePageController» предназначен для обработки запросов со страниц игр, так же этот контролер отвечает за обработку WebSoket-ов. Код контролера «GamePageController» предоставлен в приложении В. Так же для более удобной работы с сокетами были разработаны специальные классы, код, который предоставлен в листинге 3.11.

class ReviewWSConnection {

socket: ws;

gameId: number;

constructor(socket: any, gameId: number) {

this.socket = socket;

this.gameId = gameId;

}

}

class ReviewModel {

game?: number;

User?: number;

text?: string;

mark?: number;

}

Листинг 3.11 – Вспомогательные класс

Класс «ReviewWSConnection» хранит в себе сокет пользователя и идентификатор игры, для которой он подгружает отзывы, а класс «ReviewModel» представляет собой модель, хранящая данные отзыва.

Для работы с фильтрами в каталоге был разработан специальный абстрактный класс «FilterBase», код предоставлен в листинге 3.12.

abstract class FilterBase<T> {

public Tag: string;

constructor(tag: string) {

this.Tag = tag;

}

abstract filter(elements: Array<T>, filterData: any): Array<T>;

}

Листинг 3.12 – Абстрактный класс FilterBase

Данный класс определят метод «filter» который принимает массив, который будет фильтроваться и данные необходимые для фильтрации. Так, например была реализована фильтрация по тегам игры в классе «TagFilter», кото которой предоставлен в листинге 3.13.

class TagFilter extends FilterBase<game> {

constructor(tag: string) {

super(tag);

}

filter(elements: Array<game>, filterData: string): Array<game>{

if (filterData === "") {

return elements;

}

else {

let arr = new Array<game>();

elements.forEach(item => {

let filterTags = filterData.split(',').map(item => item.trim());

let gameTags = (item.tags as string).split(',').map(item => item.trim());

if (filterTags.some(filterTag => gameTags.some(gameTag => filterTag === gameTag)))

arr.push(item);

});

return arr;

}

}

}

Листинг 3.13 – Класс TagFilter

Данный класс принимает как данные фильтрации строку, которая содержит теги игры через запятую. Класс разбивает строку тегов и проверяет наличие хотя бы одного такого же тега у игры. Если хотя бы один тег совпадает, то игра подходит.

## Вывод

В данном разделе мы рассмотрели основные момент разработки веб-приложения. Рассмотрели, как построено взаимодействие клиента с сервером по протоколу HTTPS и по протоколу WebSocket. Рассмотрели, как хранится информация, как она хранится и как отображается на клиенте, как организованна фильтрация для каталога. Рассмотрели, как построена безопасность данного веб-приложения.

# Приложение А

class MVCRoute {

public url: string = "";

public method: MVCRouteMethod = MVCRouteMethod.GET;

public action?: RequestHandler | WebsocketRequestHandler;

public controller: string = "";

public isWebSocket: boolean = false;

static CreateA(url: string, method: MVCRouteMethod, action: RequestHandler): MVCRoute {

let route = new MVCRoute();

route.url = url;

route.method = method;

route.action = action;

return route;

}

static CreateB(url: string, method: MVCRouteMethod, action: RequestHandler, controller: string): MVCRoute {

let route = new MVCRoute();

route.url = url;

route.method = method;

route.action = action;

route.controller = controller;

return route;

}

}

class MVCController {

public manager?: MVCManager;

public static LinkedRoutes: MVCRoute[] = [];

public routes: MVCRoute[] = [];

AddRoute(url: string, method: MVCRouteMethod, action: RequestHandler) {

let route = new MVCRoute();

route.url = url;

route.method = method;

route.action = action.bind(this);

this.routes.push(route);

}

AddRouteWS(url: string, action: WebsocketRequestHandler) {

let route = new MVCRoute();

route.url = url;

route.isWebSocket = true;

route.action = action.bind(this);

this.routes.push(route);

}

UseLinkedRoutes() {

MVCController.LinkedRoutes.forEach(element => {

if (element.controller === this.constructor.name) {

if (!element.isWebSocket)

this.AddRoute(element.url, element.method, element.action as RequestHandler);

else

this.AddRouteWS(element.url, element.action as WebsocketRequestHandler);

}

});

}

protected UseDependency<T>(tag: string) : T {

return MVCManager.LinkedDependencies.get(tag) as T;

}

protected EndView(res: Response): void {

res.writeHead(200, { "Content-Type": "text/html;charset=utf-8" });

res.write(fs.readFileSync("./public/index.html"));

res.end();

}

}

class MVCManager {

public static LinkedControlles: Array<typeof MVCController> = new Array<typeof MVCController>();

public static LinkedDependencies: Map<string, Object> = new Map<string, Object>();

private server: Server;

private controllers: Array<MVCController>;

constructor(server: Server) {

this.server = server;

this.controllers = [];

}

public static AddDependency(tag: string, object: Object): void {

MVCManager.LinkedDependencies.set(tag, object);

}

MapRoute(url: string, method: MVCRouteMethod, action: RequestHandler) {

switch (method) {

case MVCRouteMethod.GET:

this.server.App.get(url, action);

break;

case MVCRouteMethod.POST:

this.server.App.post(url, action);

break;

case MVCRouteMethod.PUT:

this.server.App.put(url, action);

break;

case MVCRouteMethod.DELETE:

this.server.App.delete(url, action);

break;

}

}

MapRouteWS(url: string, action: WebsocketRequestHandler) {

this.server.WebSocket.app.ws(url, action);

}

UseLinkedCotrollers() {

MVCManager.LinkedControlles.forEach(i => {

let ctrl: MVCController = i.constructor();

ctrl.manager = this;

ctrl.UseLinkedRoutes();

this.controllers.push(ctrl);

});

}

UseController(controller: MVCController) {

controller.manager = this;

this.controllers.push(controller);

}

Build() {

this.controllers.forEach(ctrl => {

ctrl.routes.forEach(route => {

if (route.url == undefined || route.action == undefined)

return;

if (route.isWebSocket) {

console.log(route.url);

this.MapRouteWS(route.url, route.action as WebsocketRequestHandler);

}

else {

if (route.method == undefined)

return;

this.MapRoute(route.url, route.method, route.action as RequestHandler);

}

});

});

}

}

# Приложение Б

class Server {

public App : Express;

public WebSocket : Instance;

public Multer : Multer;

private isHttps: boolean;

constructor(ishttps: boolean = false) {

this.isHttps = ishttps;

this.App = express();

this.WebSocket = expressws(this.App);

this.Multer = multer({dest: './upload'});

this.App.use(express.static("./static"));

this.App.use(express.json());

this.App.use(cookieParser());

this.App.use(expressSession({ saveUninitialized: false, secret: 'asge122tgd', cookie: { secure: ishttps }, resave: false }));

}

Listen() {

if (this.isHttps){

const srv = https.createServer(options, this.App).listen(5000, () => {

console.log("Https server listen on: https://localhost:5000/");

});

this.WebSocket = expressws(this.App, srv);

}

else {

this.App.listen(5000, () => {

console.log("Http server listen on: http://localhost:5000/");

});

}

}

}

# Приложение В

@Controller

class GamePageController extends MVCController {

private db: DataBase;

private dataManager: DataManager;

private server: Server;

private auth: AuthService;

private wsConnections: ReviewWSConnection[];

constructor() {

super();

this.db = this.UseDependency("DataBase");

this.dataManager = this.UseDependency("Data");

this.auth = this.UseDependency("Auth");

this.server = this.UseDependency("Server");

this.wsConnections = new Array<ReviewWSConnection>();

}

@WebSocketRoute('/game/reviews/\*')

async ReviewsWS(ws: ws, req: Request) {

let gameId = Number.parseInt(req.path.split('/')[3]);

this.wsConnections.push(new ReviewWSConnection(ws, gameId));

ws.onmessage = async (mes) => {

let review = JSON.parse(mes.data as string) as ReviewModel;

let connections = this.wsConnections.filter(con => con.gameId == review.game);

let oldReview = await this.db.Instance.review.findFirst({ where: { User: review.User, game: review.game } });

if (oldReview)

await this.db.Instance.review.update({

data: { text: review.text, mark: review.mark },

where: { id: oldReview.id }

});

else

await this.db.Instance.review.create({

data: {

User: review.User as number, game: review.game as number,

text: review.text as string, mark: review.mark as number

}

});

let reviews = JSON.stringify(await this.db.GetGameReviews(gameId));

connections.forEach(con => {

con.socket.send(reviews);

})

};

ws.onerror = (error) => console.log(error.message);

ws.onclose = (evt) => {

let index = this.wsConnections.findIndex(value => value.socket == evt.target);

this.wsConnections.splice(index, 1);

};

ws.send(JSON.stringify(await this.db.GetGameReviews(gameId)));

}

@MapPost('/api/buygame')

async BuyGame(req: Request, res: Response) {

try {

let user = await this.auth.Auth(req, res);

let game = await this.db.GetGame(req.body.game);

if (!game)

throw "Game is not exists!";

let developer = await this.db.Instance.user.findFirst({

where: { id: game.User }, include: { paymentmethod: true }

});

let developerCart = developer?.paymentmethod.find(item => item.type == 1);

let gameRealPrice = game.sale\_sale\_gameTogame[0] ? (game.priceusd.toNumber() \* (1 - (game.sale\_sale\_gameTogame[0].percent as number))) : game.priceusd.toNumber();

let userData = this.dataManager.GetUserData(user.id) as UserData;

if (userData.Games?.some(i => i == game.id))

throw "Игра уже есть в библиотеке!";

if (gameRealPrice > 0) {

if (!developerCart)

throw "Разработчик не может совершить транзакцию!";

let userMP = await this.db.Instance.paymentmethod.findFirst({ where: { User: user.id, type: req.body.pmtype } }) as paymentmethod;

let userWallet = user.walletusd?.toNumber() as number;

switch (req.body.pmtype) {

case 0:

if (userWallet < gameRealPrice)

throw "В кашельке не хватает средств!";

await this.db.Instance.user.update({

where: { id: user.id },

data: { walletusd: userWallet - gameRealPrice }

});

break;

case 1:

if (!userMP)

throw "Отсутсвует карта!";

break;

default:

throw "Не существующий способ оплаты!";

}

await this.db.Instance.transaction.create({

data: { game: game.id, customerpaymentmethod: userMP.id, vendorpaymentmethod: developerCart.id }

});

userData.Games?.push(game.id);

this.dataManager.SetUserData(userData);

}

else {

userData.Games?.push(game.id);

this.dataManager.SetUserData(userData);

}

console.log("Buy success!");

res.json({ ok: true });

}

catch (error) {

res.json({ ok: false, error: error });

}

}

}