ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ І. І. МЕЧНИКОВА
(повне найменування закладу вищої освіти)

Факультет математики, фізики та інформаційних технологій
(повне найменування факультету)

Кафедра інформаційних технологій
(повна назва кафедри)

Кваліфікаційна робота

на здобуття ступеня вищої освіти «Бакалавр»

«Розробка веб-додатку 2D гри з використанням технологій NodeJS, ReactJS, WebSocket та TelegramWebApp»

(тема кваліфікаційної роботи українською мовою)

«Development of a 2D Game Web Application Using NodeJS, ReactJS, WebSocket, and TelegramWebApp Technologies»

(тема кваліфікаційної роботи англійською мовою)

Виконав: здобувач денної форми навчання
спеціальності 122 Комп'ютерні науки
(код, назва спеціальності)
Освітня програма Комп'ютерні науки
(назва)
Діденко Кирило Юрійович
(прізвище, ім'я, по-батькові здобувача)

Керівник д.т.н., професор Казакова Н.Ф.
(науковий ступінь, вчене звання, прізвище, ініціали) (підпис)

Рецензент д.т.н., доцент Соколов А.В.
(науковий ступінь, вчене звання, прізвище, ініціали)

Рекомендовано до захисту:
Протокол засідання кафедри
Інформаційних технологій .
№ 1 від 09 червня 2024 р.

Завідувачка кафедри
(підпис) КАЗАКОВА Надія .
(прізвище,ім'я)

Захищено на засіданні ЕК № 13 .
протокол № 16 від 20 червня 2024 р.

Оцінка відмінно / A / 90 .
(за національною шкалою/шкалою ECTS/ бали)

Голова ЕК
(підпис) КОПИЧЕНКО Іван .
(прізвище,ім'я)

Одеса 2024

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ 5
ВСТУП 6
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ ВЕБ-ДОДАТКІВ 8
1.1 Аналіз предметної області 8
1.2 Аналіз існуючих веб-додатків 10
2 ВИБІР ТА ОБГРУНТУВАННЯ ЗАСОБІВ РОЗРОБКИ 13
2.1 Функціональні та нефункціональні вимоги 13
2.2 Вибір технологій для створення клієнтської частини 13
2.3 Вибір технологій для створення серверної частини 18
2.4 Вибір технологій для створення бази даних 22
2.6 Перелік способів організації з'єднання в реальному часі 26
3 РЕАЛІЗАЦІЯ ВЕБ-ЗАСТОСУНКУ 32
3.1 Проектування інформаційної системи 32
3.2 Базова структура проекту та основні фреймворки 38
3.2 Створення API та налаштування клієнт-серверного з'єднання 40
3.5 Підключення до Telegram та тестування фінальної версії веб-застосунка 51
ВИСНОВКИ 57
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 58
ДОДАТОК А Вихідний код програми 60

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

БД – база даних

ООП – об'єктно-орієнтоване програмування

WS – WebSocket

UI – User Interface – інтерфейс користувача

API – Application Programming Interface – програмний інтерфейс програми

RNG – Random Number Generation – генерація випадкового числа

TDD – Test-Driven Development – розробка через тестування

REST – Representational State Transfer – передача репрезентативного стану

ВСТУП

У сучасному світі, переповнений різною інформацією, будь-який навіть унікальний проект стикається з величезною конкуренцією за час, який готовий приділити користувач на оцінку даного проекту.

І якщо в програмах, що не належать до розважальної сфери, таких як програми для роботи, освіти або здоров'я у користувача на початку є конкретна мета, під яку він шукає інструмент для реалізації, яким може виступати програма, то у випадку розважальних програм, таких як ігри, коли мета потенційного користувача – приємно провести час, додаток змушений конкурувати з усіма видами дозвілля.

Таким чином, відеоігри навіть, мобільні, знаходяться в заздалегідь програшному положенні щодо таких видів дозвілля як перегляд фільмів, розважальних шоу або читання стрічки новин в соцмережах. Сучасному користувачеві важливо отримати миттєвий доступ до основного матеріалу без проміжних кроків. Через необхідність проміжних етапів, таких як завантаження та реєстрація , додатки, які потенційно могли б сподобатися користувачам, залишаються неоціненими. Саме тому пошук рішення для максимального швидкого надання доступу користувачу до безпосередньо ігрового процесу допоможе привернути нових користувачів та є актуальним сьогодні.

Одним з варіантів обійти потребу у завантаженні додатку є зробити його браузерним, але більшість користувачів все одно з недовірою відноситься до незнайомих їм сайтів, тому ідеальним варіантом було б розмістити додаток на площадці, знайомій більшості потенціальних користувачі.

Обійти потребу в реєстрації з зберіганням прогресу для кожного користувача можливо, якщо прив'язати вже існуючий аккаунт на іншому сервісі.

Месенджер Telegram надає можливості для впровадження гри в нього, та легкої прив'язки внутрішнього аккаунту до основного аккаунту у месенджері за id, а також зручних інтерфейс за допомогою повідомлень Telegram ботів.

Метою кваліфікаційної роботи є розробка браузерної багатокористувальницької онлайн гри з використанням технології Telegram WebApi для надання доступу до гри всередині додатку Telegram та прив'язки ігрових досягнень до Telegram аккаунту. Для досягнення поставленої мети були сформульовані наступні завдання:

– провести аналіз предметної області щодо браузерних ігор та їх впровадження у сторонні додатки;

– провести порівняльний аналіз існуючих програм з аналогічною системою прив'язки до акаунта месенджеру;

– обґрунтувати вибір програмних засобів розробки та технологій;

– виконати реалізацію застосунку;

– виконати тестування застосунку.

Структура кваліфікаційної роботи бакалавра складається з вступу, 3 розділів, висновків, переліку посилань на 12 найменувань, 1 додатку. Повний обсяг проекту становить 98 сторінок, містить 20 рисунків і 1 таблицю.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ ВЕБ-ДОДАТКІВ

1.1 Аналіз предметної області

Браузерні ігри, тобто відеоігри, які можна запускати і грати через веб-браузери без необхідності встановлення додаткового програмного забезпечення, займають важливе місце в сучасній цифровій культурі. Вони з'явилися на початку 2000-х років і швидко здобули популярність завдяки своїй доступності та простоті використання.

Одним з найбільшим до розвитку браузерних ігор був випуск Adobe Flash Player та його подальше впровадження до більшості браузерів існуючий в ті часи. Flash Player став стандартом для відтворення відео, анімацій, інтерактивних додатків та ігор у веб-браузерах. Він надав розробникам потужний інструмент для створення складних, інтерактивних ігрових додатків, які могли виконуватися безпосередньо у веб-браузерах.

Його основні характеристики включали:

– Платформна незалежність: Flash Player працював у більшості сучасних браузерів і операційних систем, що робило ігри доступними для широкої аудиторії.

– Висока інтерактивність: Підтримка мови програмування ActionScript дозволяла створювати складні ігрові механіки та анімації.

– Графічні можливості: Векторна графіка Flash забезпечувала високу якість зображень при мінімальному використанні ресурсів.

Ці характеристики сприяли популярності браузерних ігор, таких як "FarmVille", "Bejeweled", та "Club Penguin", що стали культурними феноменами свого часу.

31 грудня 2020 року Adobe офіційно припинила підтримку Flash Player, мотивуючи це розвитком відкритих стандартів, таких як HTML5, WebGL та WebAssembly, які забезпечують більшу безпеку, продуктивність та сумісність.

Припинення підтримки Adobe Flash Player у 2020 році стало серйозним ударом для браузерних ігор, які значною мірою залежали від цієї технології. Перехід на нові веб-стандарти, такі як HTML5, WebGL та WebAssembly, хоча й відкрив нові можливості, вимагав значних зусиль та ресурсів для адаптації старих ігор.

Мобільні платформи, такі як iOS та Android, стали надзвичайно популярними у 2010-х роках. Смартфони та планшети надали користувачам доступ до тисяч ігор через App Store та Google Play. Ці ігри часто були більш складними та інноваційними, порівняно з браузерними аналогами, і забезпечували зручність гри на ходу.

Розвиток соціальних мереж та платформ, таких як Facebook, Instagram та YouTube, надав користувачам нові форми розваг та соціальної взаємодії, зменшуючи час, який вони готові приділити грі. На початку 2010-х років інтеграція ігор в соціальні мережі, такі як Facebook, стала важливою тенденцією у розвитку цифрових розваг. Цей підхід дозволяв розробникам використовувати величезну користувацьку базу соціальних платформ для залучення гравців та створення більш інтерактивного та соціального досвіду. Також цей підхід був досить зручним для користувачів, бо дозволяє прив'язувати прогрес до вже існуючого аккаунту в соцмережі. Але на сьогоднішній день популярність соціальних мереж почала знижуватися. Месенджери, такі як WhatsApp, Telegram та Viber, стали основними засобами комунікації для багатьох користувачів, а також, і основною розважальною платформою.

У 2022 році, в одному з найпопулярніших месенджерів Telegram з'явилася можливість інтегрувати web-додаток до чату с ботом завдяки технології Telegram MiniApp. Правильне використання цієї можливості потенційно може повернути популярність невеликий багатокористувальницьких браузерних ігор до рівня 2010х років.

1.2 Аналіз існуючих веб-додатків

Одним із прикладів інтеграції гри до Telegram є бот @gamee (рис 1.1), який дозволяє користувачам грати в ігри безпосередньо у месенджері [1]1). Щоб розпочати взаємодію з ботом @gamee, користувачі повинні знайти його у пошуку Telegram та запустити стандартну команду /Start. Ця команда активує бота та відкриває діалогове вікно, де користувачеві пропонується вибрати тип та жанр гри. Бот пропонує користувачам декілька варіантів ігор, а також можливість обрати режим гри: з друзями або з випадковим суперником. Спільна гра з друзями організовується наступним чином: користувач натискає кнопку "With Friends" та вибирає зі списку контактів тих, з ким він хоче пограти. Бот автоматично надсилає запрошення обраному другу, що містить посилання для приєднання до гри.

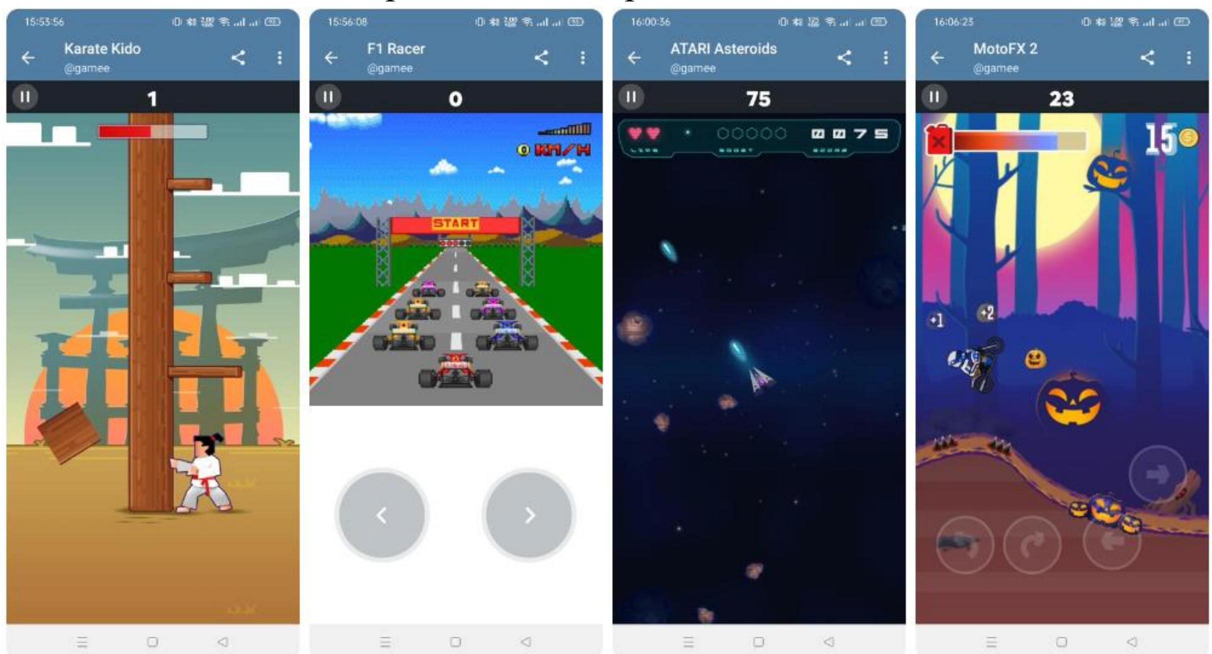

Рисунок 1.1 – приклади ігор доступних у @gamee

1) [1] GAMEE Telegram | GAMEE Wiki. Welcome | GAMEE Wiki. URL:https://wiki.gamee.com/gameplay/telegramaccounts

Ігри жанру клікера, відомі також як "idle games" або "tap games", стали популярним інструментом для впровадження та популяризації криптовалют. У таких іграх гравці виконують прості дії, наприклад, клікають на екран, щоб заробляти внутрішньоігрову валюту або ресурси. Популярність таких ігор зростає завдяки їх простоті, можливості отримати реальні криптовалютні винагороди та інтеграції з блокчейн-платформами. Вони дозволяють користувачам заробляти криптовалюту, беручи участь у легких ігрових активностях, що робить цей процес веселим і доступним навіть для новачків у світі криптовалют.

Основна концепція гри Catizen полягає у схрещуванні котів однакового рівня для отримання кота вищого рівня. Коти для схрещування надходять у переносках, які з'являються автоматично з певною періодичністю. Відвідувачі, які приходять гратися з котами, приносять віртуальні монети, за які можна купувати нових котів для подальшого схрещування.

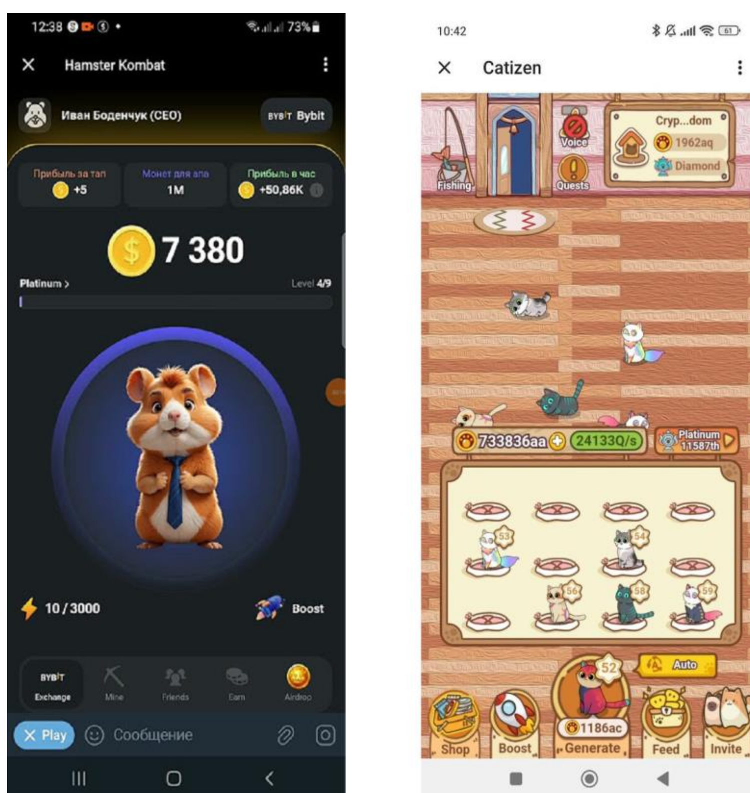

Рисунок 1.2 – приклади ігор-клікерів Hamster Kombat та Catizen

Технологія Telegram Web App є відносно новою, що обумовлює обмежену кількість варіантів реалізації та впровадження вебдодатків у середовище Telegram. Вона ще перебуває на стадії розвитку, тому екосистема навколо неї поки що не така розвинена.

Більшість існуючих ігор у Telegram Web App не є повноцінними іграми, які ми звикли бачити на інших платформах. Часто це або графічні обгортки для криптовалютних проектів, або невеликі інтерактивні додатки, що демонструють можливості самої платформи Telegram. Вони є скоріш демо-версії, які показують, як можна використовувати Telegram Web App.

Таким чином, на даний момент можливості Telegram Web App ще не повністю реалізовані, а самі вебдодатки лише починають знаходити своє місце в системі Telegram. Однак з часом, із розвитком технології та зростанням спільноти розробників, можна очікувати появу більш складних і цікавих веб-додатків, включаючи повноцінні ігри.

2 ВИБІР ТА ОБҐРУНТУВАННЯ ЗАСОБІВ РОЗРОБКИ

2.1 Функціональні та нефункціональні вимоги

Перед початком розробки застосунку були визначені функціональні та нефункціональні вимоги до нього.

До функціональних вимог належать:

– можливість створити нову гру, написав Telegram боту команду;

– можливість запустити гру кнопкою, отриманою від бота;

– можливість приєднання до ігрової сесії до 4 гравців за допомогою паролю отриманого від бота;

– повна синхронізація дій гравців між усіма учасниками сесії.

До нефункціональних вимог належать:

– будь-який пристрій підтримуючий Telegram версії від 16 квітня 2022 року або вище;

– підключення до Інтернету.

2.2 Вибір технологій для створення клієнтської частини

HTML (HyperText Markup Language) – мова розмітки, призначена для встановлення правил відображення елементів на веб-сторінці у браузері. У HTML для розмітки використовуються елементи або як їх ще називають теги, що записуються між символами "<" і ">", деякі елементи є відкриваючими та закриваючими, вони вказують коли має початися будь-який ефект відображення і коли він має закінчитися. [2][1)]

CSS (Cascading Style Sheets) – мова декодування, використовується для встановлення ієрархічних правил, що задають зовнішній вигляд документа.

[1)] [2] Офіційна документація HTML: HyperText Markup Language. URL: https://developer.mozilla.org/en-US/docs/Web/HTML

Таблиця стилів CSS визначає як будуть відображатися HTML елементи при завантаженні веб-сторінки, в ній задаються шрифт, розміри кожного елемента, відступи позиціонування елементів відносно один одного і багато інших стилів. Також CSS дозволяє по-різному малювати документ в залежності від пристрою користувача за допомогою медіавиражень, що важливо для створення дизайну для мультиплатформних сайтів і додатків[3][1)].

JavaScript – мова програмування, що широко застосовується в браузерах. Якщо HTML та CSS відповідають за статичне заповнення веб-сторінки, то JS – за динамічне. JS надає інтерактивності сайту, дозволяючи відображати потрібне наповнення, динамічно змінювати HTML елементи, керувати мультимедіа, анімувати зображення тощо.

Основні архітектурні риси JS:

– Підтримує об'єктно-орієнтований, імперативний та функціональний стилі. JS дозволяє створювати класи, об'єкти та їх екземпляри. Об'єкти створюються у коді чи з допомогою конструкторів. JavaScript використовує прототипне успадкування, де кожен об'єкт має прототип, від якого він успадковує методи та властивості. Це відрізняється від класичного спадкування, але дозволяє створювати гнучкі та динамічні структури даних. Також у JS функції є об'єктами першого класу, що означає, що їх можна передавати як аргументи іншим функціям, повертати з функцій і зберігати в змінних. Таким чином, JavaScript має гнучкість, що дозволяє комбінувати різні стилі програмування залежно від вимог конкретної задачі.

– Динамічна типізація. У мовах програмування з динамічним видом типізації тип змінної задається при створенні змінної, а момент присвоювання їй значення. Тобто, на різних етапах свого життєвого циклу одна й та сама змінна може набувати значень різних типів. Це спрощує написання складних програм з непередбачуваним оточенням, але може призвести до несподіваних помилок на етапі компіляції у випадках, коли змінною з якихось причин було

[1)][3] Офіційна документація CSS: Cascading Style Sheets | MDN. MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Web/CSS

встановлено тип, який функція, в якій ця змінна використовується, не може обробити. Для надання коду на статичної типізації JS існує бібліотека TypeScript.

– Автоматичне управління пам'яттю. У JS управління пам'яттю здійснюється автоматично за допомогою механізму складання сміття (англ. Garbage collector). Пам'ять автоматично виділяється під час створення змінних, класів та інших структур. Складальник сміття звільняє пам'ять, виділену об'єктам, які більше не доступні або не використовуються. Цей процес використовує різні механізми оптимізації, які дозволяють ефективно використовувати пам'ять і підвищувати продуктивність додатків. Однак, у разі потреби JS надає можливість керувати пам'яттю безпосередньо через низку методів.

– Асинхронне програмування. JS підтримує асинхронне програмування, що дозволяє виконувати операції у фоновому режимі без блокування основного потоку виконання. Це особливо корисно для роботи з мережними запитами та обробки великих обсягів даних [4][1)].

– React.js – JavaScript-бібліотека, що широко використовується для написання користувальницьких інтерфейсів веб-додатків. Вихідний код React.js відкритий, тому окрім постійної підтримки самої бібліотеки, постійно з'являється безліч додаткових бібліотек від незалежних розробників, що суттєво полегшує роботу та дозволяє економити час на написанні деяких рутинних речей, таких як запити до API. Поширеність самого React.js і кількість сторонніх бібліотек є головною перевагою React.js перед популярними фреймворками для створення веб-інтерфейсів, таких як Vue.js і Angular [5][2)]..

React.js має такі переваги:

– Компонентний підхід. React побудований навколо концепції компонентів, які дозволяють розбивати інтерфейс користувача на безліч

[1)] [4] Уроки React | Codecademy. Codecademy. URL: https://www.codecademy.com/learn/react-101
[2)] [5] Офіційна документація React. URL: https://reactjs.org/docs/getting-started.html

невеликих і повторно використовуваних частин. Компоненти можуть бути вкладеними в один одного, бути абстрактними, успадковувати свої властивості, мати приховані та публічні методи, таким чином дозволяючи застосовувати об'єктно-орієнтований підхід.

– Використання JSX. Компоненти React.js використовують особливе розширення файлів .jsx. Такий підхід дозволяє об'єднати JavaScript та HTML в одному файлі, що спрощує написання та розуміння коду компонентів, а також покращує можливості його читання.

– Віртуальний DOM. Для роботи з HTML елементами за допомогою JavaScript, структура HTML документа представляється у вигляді Об'єктної Моделі Документу (англ. Document Object Model, коротко DOM). DOM дерево – структура, у якій кожен вузол дерева – це об'єкт, поля якого – поля відповідного HTML елемента. У React.js, розробник безпосередньо не змінює HTML документ (при створенні React проекту за допомогою команди create-react-app створюється файл index.html з елементом <div id="root"></div>), цей файл далі не змінюється. Замість цього, в момент збірки веб-програми, наприклад командою npm run build, JSX файли, що містять інформацію про те, що потрібно відображати на екрані, компілюються в звичайний JS, що підтримується всіма браузерами. Головна перевага такого підходу – зміна стану вузлів віртуального DOM дерева не потребує оновлення сторінки для перемальовування сайту. Це робить завантаження динамічного наповнення сайту значно швидшим. Ці особливості роблять розробку інтерфейсів користувача за допомогою React.js значно комфортніше і швидше ніж при роботі з нативними JS і HTML.

Також для полегшення розробки було встановлено ці бібліотеки:

– Redux. У класичному React.js інформація між компонентами передається від батьківського компонента компоненту-дитині при ініціації нового компонента в спеціальному полі props. Кожен компонент містить об'єкт з переданими в нього таким чином функціями та змінними, звернутися до нього можна як будь-якому іншому JS об'єкту, наприклад, звернення до

props.name дасть доступ до значення поля “name”. Передача інформації “знизу вгору”, від дитини до батька так само можлива, для цього потрібно передати в компонент, що ініціюється, функцію, що привласнює потрібні дані заданої змінної, а так само змінну. Такий підхід зручний при невеликій вкладеності компонентів і суворій ієрархії, при якій рівносильні компоненти не потребують обміну інформацією між собою. Але не завжди велику програму вдається спроектувати ідеально, щоб полегшити цей процес існує бібліотека Redux. Redux дозволяє створити в окремому файлі глобальне сховище, яке традиційно називають store.js, в який виноситься логіка, яка використовується в декількох компонентах одночасно і глобальні змінні. Доступ до сховища можна отримати з будь-якої точки програми. Для опису списку можливих дій з даними в сховищі створюється функція reducer(), switch-case конструкція, з двома параметрами action і state. Поле action визначає номер case який потрібно задіяти, а полю state присвоюються дані після зміни. [6][1)] Так само в даному проекті функціонал Redux розширений такими бібліотеками як Redux Thunk, що надає функціонал для виконання асинхронних запитів до бази даних, а також бібліотека RTK Query, яка містить деякий прописаний функціонал, що полегшує налаштування store.js. [7][2)]

– react-router-dom. React.js був створений для написання виключно односторінкових додатків (англ. single page application, SPA), тому в ньому немає нативих методів зміни URL адреси сторінки, що робить код менш структурованим, не дозволяє відправити посилання відразу на сторінку з потрібним наповненням, а так само не дозволяє отримувати інформацію з параметрів вказаних в адресному рядку. Незважаючи на те, що react-router-dom вирішує ці проблеми, програма залишається односторінковою і так само використовує один HTML документ для відображення кожної сторінки, динамічно завантажуючи вміст, але вже з урахуванням URL адреси.

[1)] [6] Офіційна документація Redux Toolkit. URL: https://redux-toolkit.js.org/

[2)] [7] Офіційна документація Redux | Redux. Redux – A JS library for predictable and maintainable global state management | Redux. URL: https://redux.js.org/introduction/getting-started

– Axios. Бібліотека, що полегшує написання асинхронних запитів до бази даних, надає зручні однорядкові шаблони для запиту, для написання якого в нативному JS потрібно було б скласти ланцюжок з кількох .then.

2.3 Вибір технологій для створення серверної частини

Хост (host – «власник, який приймає гостей»), пристрій, що управляє ігровим процесом і обробляє дані, що поступають від гравців. Хост повинен забезпечувати узгодженість стану гри між усіма учасниками для коректного відображення синхронізованої інформації у кожного учасника ігрової сесії. Пристрій є хостом виконує необхідні розрахунки, а потім розсилає його іншим пристроям в мережі.

Існує два архітектурні підходи для написання ігрового сервера, підхід при якому сервер виступає хостом, і підхід при якому хостом виступає один із гравців. Коли хост – сервер, всі необхідні обчислення виконуються безпосередньо на сервері, після чого сервер ініціює відправку інформації на пристрої пов'язаних з ним клієнтів. Головний мінус такого підходу – величезна витрата ресурсів сервера, адже для кожної ігрової сесії сервер змушений розраховувати дії гравців самостійно, отримуючи від клієнтських пристроїв лише виклик потрібних функцій та параметри цього виклику. Але цей підхід має безліч переваг.

Переваги сервера-хоста:

– Спрощення розробки. Через меншу кількість запитів потребуючих обробки, а також за рахунок того, що логіка функцій прописана в тому ж репозиторії як і умови виклику відповідної функції, з точки зору архітектури, реалізувати даний підхід простіше;

– Чітерство. Чітерство (англ. cheat «шахрайство, обманювати») – використання помилок, сторонніх програм, зміна файлів гри для отримання переваги, наприклад, отримання інформації, яка не повинна бути доступна гравцеві або заміна коефіцієнтів, що використовуються в розрахунках на

власні. Виконання важливих обчислень на стороні клієнта дає можливість гравцеві втрутитися у цей процес. Можливість використання чітів згубна для багатокористувальницьких ігор, адже робить ігровий процес менш цікавим для чесних гравців через неможливість змагатися з чітерами нарівні;

– Зменшення системних вимог. Виконання логіки на сервері дозволяє зменшити навантаження на пристрої користувачів;

– Безпека. При надсиланні даних на сервер, користувач може втрутиться в процес не тільки з метою вплинути на ігровий процес, але й завдати шкоди іншим користувачам в ігровій сесії. Підхід у якому хост-сервер виключає можливість відправки іншим гравцям інформації зміненої зловмисником і вирішує проблему приховування інформації про користувачів, такий як ip адреси, логіни, id;

При підході клієнт-хост функції, як і їх виклики виконуються повністю на пристрої користувача, на сервер відправляється лише невеликий обсяг констант потрібних для синхронного відображення данних у всіх учасників ігрової сесії, таких як координати, доступна для всіх інформація (кількість очок здоров'я, псевдонім, аватар) , повідомлення у чаті. Після цього сервер редагує інформацію у виділеній для даної ігрової сесії ділянці пам'яті та розсилає її всім учасникам заново. Даний підхід дуже сильно розвантажує сервер, але має ряд мінусів, які відсутні у сервера-хоста.

Незважаючи на всі переваги підходу хост-сервер, через дуже обмежені ресурси доступні для виділення на хостинг сервера, в даному проекті, хостом виступає пристрій користувача. Перший гравець, що приєднався до гри і створив “кімнату”, виконує обчислення для кожного з учасників його ігрової сесії, після чого відправляє результати на сервер для синхронізації інших гравців. [8][1)]

Для написання серверної частини було обрано програмне середовище Node.js , що дозволяє виконувати код мовою JavaScript не тільки в браузері,

---

[1] [8] Сумець О. Проєктування операційних систем. KROK University, 2021. URL: https://doi.org/10.31732/pros

але і безпосередньо на сервері, в додатках (як десктопних, так і мобільних). Для цього використовує V8 від Google, спеціалізовану програму, що обробляє JavaScript, зокрема у браузері Google Chrome .

Node.js має ряд переваг, таких як:

– Модель асинхронного виконання: Node.js використовує асинхронну модель виконання, що означає, що операції вводу-виводу (I/O), такі як читання з файлу або надсилання HTTP-запиту, виконуються асинхронно. Замість блокування потоків, що очікують завершення операції введення-виведення, Node.js продовжує виконувати інші завдання до її завершення, після чого відповідний обробник продовжить роботу з результатом цієї операції;

– Архітектура Node.js подіє орієнтована (англ. event-driven architecture, EDA), заснована на обробці, що керується подіями, що дозволяє ефективно обробляти кілька одночасних запитів без блокування потоку виконання. Таким чином, запити від різних клієнтів спокійно виконуються одночасно, без потреби створювати чергу виконання;

– Цикл подій (event loop). Завдання в Node.js поділяються на мікрозавдання та макрозавдання. Черга виконання завдань являє собою нескінченний цикл, що розподіляє завдання в черзі не тільки від найстарішої до найновішої, але й відповідно до їх складності, таким чином невелика задача, класифікована Node.js як мікрозавдання (наприклад закінчення таймера) може бути додано у чергу виконання між макрозавданнями, навіть якщо це мікрозавдання було оголошено вже у процесі виконання одного з макрозавдань. Це суттєво зменшує затримку для кожного окремого користувача, у випадках, коли навантаження на сервер вище за очікуване [9][1)];

– Серверна та клієнтська частина програми написана однією мовою. При розробці повноцінної веб-програми дуже часто доводиться постійно вносити зміни до коду клієнтської та серверної половини паралельно. Загальний

---

1) [9] Офіційна документація Node.js. Node.js – Run JavaScript Everywhere. URL: https://nodejs.org/docs/latest/api/

синтаксис мови JavaScript робить процес внесення правок та тестування набагато комфортнішим.

Таким чином провівши порівняння з такими альтернативами, як мова PHP і фреймворк Spring для мови Java, які інтенсивно застосовуються для розробки веб-додатків, зручність і невибагливість до апаратного забезпечення, стали причинами вибору JavaScript (за підтримки Node.js) мовою для написання серверної частини програми.[10][1)]

Також для полегшення розробки було встановлено ці бібліотеки:

– CORS (Cross-Origin Resource Sharing). Сучасні браузери під час надсилання запиту в заголовку запиту вказують параметри Host та Origin. У них вказується доменне ім'я сайту, на який надсилається запит і доменне ім'я сайту джерела, з якого запит ініціюється відповідно. У випадку, якщо ці параметри не збігаються, запит блокується браузером через недотримання політики одного джерела (англ. same-origin policy). У додатку розробленому в ході цієї роботи, серверна та клієнтська частина будуть розміщені на різних доменах, як на етапі розробки так і в вже готовому проекті, тому для того щоб запити не блокувалися необхідно прямо вказати дозволені домени. Найзручніше це зробити за допомогою бібліотеки CORS для Node.js;

– Express.js. Найпопулярніша бібліотека для швидкого розгортання та налаштування сервера на Node.js. Полегшує створення сервера, обробку запитів, додавання проміжних етапів обробки (англ. Middleware).

– node-telegram-bot-api. Бібліотека для використання функцій Telegram всередині програми, таких як відправлення повідомлень, створення кнопок, створення вікна для веб-програми за вказаним посиланням, доступ до аватару, псевдоніму та id Telegram облікового запису користувача.

– socket.io. Бібліотека, що полегшує роботу з webSockets, має готові методи для відкриття з'єднання при підключенні нового користувача, закриття

[1)] [10] Nystrom R. Game Programming Patterns. Genever Benning, 2014. 354 c.

з'єднання при завершенні роботи, автоматичне відновлення з'єднання у разі тимчасових помилок.

Для більш комфортної роботи також були встановлені деякі залежності розробки (англ. DevDependencies). Ці залежності не будуть завантажуватись при запуску програми на хостингу, але будуть корисні під час розробки та тестування.

– @types/node-telegram-bot-api. Запобігає розпізнаванню типів IDE з TelegramBotAPI як помилки. Не впливає на компіляцію коду, але видаляє дратівливі підкреслення червоним чи жовтим кольором правильно написаного коду.

– nodemon. Невеликий скрипт для автоматичного перезапуску програми під час внесення змін до коду. Дуже полегшує розробку через тестування (англ. test-driven development, TDD), техніку розробки при якому під кожну бажану зміну спочатку пишеться тест, пишеться код потрібний для проходження цього тесту, і якщо результати тесту незадовільні – код переписується після успішного проходження тесту змінюється. Цей цикл є досить коротким і повторюється багато разів, тому автоматизація перезапуску економить дуже багато часу і сил розробника.

2.4 Вибір технологій для створення бази даних

На початковому етапі розробки як місце для зберігання констант використовувалася база даних, що складається з декількох JSON файлів, дані з яких зчитувалися сервером за допомогою вбудованого в Node.js модуля node:fs (File System) і зберігався в локальну змінну у форматі JavaScript об'єкта за допомогою JS метод JSON.parse(), що повністю задовольняло потреби програми. Але, з часом, розширення бази даних на множину окремих великих об'єктів, що замінюють таблиці в класичній реляційній структурі, було прийнято рішення додати в проект систему управління базами даних (СУБД).

Оскільки існуючу БД у будь якому випадку потрібно було переписувати під формат СУБД розглядався варіант переформатовати дані під реляційну модель.

Особливості реляційних БД:

– Модель даних – таблиці

– Дані максимально структуровані;

– Продуктивність нижче;

– Додавання нових таблиць, зміна структури наявних складніше.

Також, заради зменшення обсягу роботи для перенесення вже наявних записів у нову БД, було розглянуто нереляційні СУБД.

Особливості реляційних БД:

– Модель даних – документи (JSON файли);

– Дані частково структуровані, за бажання можуть бути не структуровані зовсім;

– Продуктивність вище;

– Додавання нових документів, зміна структури наявних простіше [11][1)].

Розглянувши 2 варіанти СУБД, реляційну PostgreSQL і нереляційну MongoDB, незважаючи на надання можливостей для уникнення потенційних помилок за рахунок суворішої структури даних, вибір був зроблений у бік MongoDB для інтеграції JSON документів, написаних раніше замість повного переписування у формат таблиць.

2.5 HTTP протокол

HTTP – це широко використовуваний протокол передачі даних, спочатку призначений для передачі гіпертекстових документів (тобто документів, які можуть містити посилання, які дозволяють переходити до інших документів).

---

1) [11] Офіційна документація MongoDB. MongoDB Documentation. URL: https://docs.mongodb.com/

Абревіатура HTTP розшифровується як HyperText Transfer Protocol. Протокол HTTP працює на прикладному рівні мережевої архітектури і використовує TCP/IP для передачі даних через мережу. HTTP передбачає використання структури передачі даних клієнт-сервер. Клієнтська програма генерує запит і надсилає його на сервер, після чого сервер обробляє запит, генерує відповідь і надсилає його назад клієнту. Традиційно за допомогою протоколу HTTP здійснюється обмін даними між програмою користувача, яка звертається до веб-ресурсів (зазвичай веб-браузер), і веб-сервером. На даний момент переважна більшість запитів здійснюється за цим протоколом. [12][1)]

Протокол HTTP містить ряд методів для клієнт-серверної взаємодії. HTTP-методи – це команди, які використовуються у протоколі HTTP для визначення типу дії, яку клієнтський браузер або інший клієнт хоче здійснити на веб-сервері.

Список основних HTTP-методів:

– GET. Використовується для отримання ресурсів з сервера.

– POST. Використовується для надсилання даних на сервер.

– PUT. Використовується для оновлення існуючого ресурсу на сервері.

– DELETE. Використовується для видалення ресурсу на сервері.

– PATCH. Використовується для часткового оновлення ресурсу на сервері.

– OPTIONS. Використовується для отримання інформації про можливі методи або параметри, які підтримуються на сервері.

– HEAD. Аналогічно до GET, але сервер відправляє тільки заголовки відповіді, без тіла документа. Використовується для отримання мета-інформації про ресурс, наприклад, дати останньої модифікації або розміру файлу, без необхідності отримання всього вмісту.

Структура HTTP-запиту складається з трьох основних частин:

[1)] [12] Vickler A. Javascript: Javascript Back End Programming. Independently Published, 2021

– Рядок запиту (англ. Request Line). Це перший рядок в запиті і містить метод запиту, URI (або URL) ресурсу та версію протоколу HTTP.

– Заголовки запиту (англ. Request Headers). Ця частина містить різноманітні заголовки, які передають додаткову інформацію про запит.

Кожен заголовок вказується на окремому рядку і має формат "Назва: Значення". Деякі заголовки можуть бути обов'язковими для певних типів запитів, таких як Content-Type для POST-запитів.

– Тіло запиту (Request Body). Ця частина не завжди присутня, але використовується для передачі додаткових даних, таких як форми або JSON-об'єкти. Якщо тіло запиту відсутнє, наприклад у GET-запитах, то ця частина буде порожньою. [13][1)]

Також HTTP надає можливості керувати такими функціями:

– Кешування. Сервер може давати вказівки проксі-серверам і клієнтам про те, що кешувати і як довго. Клієнт може наказати проміжним проксі-кешам ігнорувати збережений документ.

– Пом’якшення обмеження щодо походження: щоб запобігти відстеженню та іншим вторгненням у конфіденційність, веб-браузери забезпечують суворе розділення веб-сайтів. Лише сторінки з одного походження можуть отримати доступ до всієї інформації веб-сторінки. Хоча таке обмеження є тягарем для сервера, HTTP-заголовки можуть послабити це суворе розділення на стороні сервера.

– Автентифікація: деякі сторінки можуть бути захищені, щоб доступ до них мали лише певні користувачі. Базова автентифікація може здійснюватися за допомогою HTTP або за допомогою WWW-Authenticate та подібних заголовків, або шляхом встановлення певного сеансу за допомогою файлів cookie HTTP.

– Проксі та туннелювання: сервери або клієнти часто знаходяться в інтрамережах і приховують свою справжню IP-адресу від інших комп’ютерів.

---

1) [13] Офіційна документація Express. Express – Node.js web application framework. URL: https://expressjs.com/

Потім запити HTTP проходять через проксі-сервери, щоб подолати цей мережевий бар’єр. Не всі проксі є HTTP-проксі. Протокол SOCKS, наприклад, працює на нижчому рівні. Інші протоколи, наприклад ftp, можуть оброблятися цими проксі-серверами.

– Сеанси: використання файлів cookie HTTP дозволяє пов’язувати запити зі станом сервера. Це створює сеанси, незважаючи на те, що базовий HTTP є протоколом без стану.

2.6 Перелік способів організації з'єднання в реальному часі

Стандартний протокол HTTP передбачає архітектуру клієнт-сервер, у якій клієнт завжди ініціює передачу даних, а сервер відповідає лише на запити клієнта. Але щоб такі додатки, як чати та онлайн ігри працювали належним чином для всіх учасників, сервер повинен ініціювати надсилання даних у режимі реального часу.

Існують 4 основні підходи для встановлення з'єднання в реальному часі:

– Опитування (англ. Short-polling). Опитування передбачає опитування сервера щодо нових даних через певний інтервал. Клієнт надсилає запит кожні n секунд, сервер відповідає як зазвичай. Цей підхід є досить розповсюдженим, але його основним недоліком є надмірне споживання ресурсів сервера. Для кожного запиту потрібно встановити TCP-з’єднання та надіслати запити, зробити запит до бази даних і повернути дані (найчастіше ті самі, які поверталися раніше, оскільки нових ще немає) і закрити з’єднання.

– Довге опитування (англ. Long-polling). Довге опитування також здійснює запит, але з відмінностями. Клієнт надсилає запит, сервер тримає запит у стані очікування до появи нових даних і лише тоді відповідає. Тут не ініціються зайві запити, але не кожен веб-сервер зможе підтримувати велику кількість затриманих запитів у пам’яті одночасно.

Для кожного запиту необхідно:

1) Встановіть TCP-з'єднання та надіслати заголовки запиту

2) Тримати запит в очікуванні, доки не надійдуть нові дані (або запит буде припинено клієнтом/тайм-аутом)

3) Повернути нові дані та закрити з'єднання

Цей підхід добре працює для серверів, що працюють на асинхронній моделі (наприклад, Node.js), але його досить важко масштабувати (і тому він є не дуже популярним) на серверах, які працюють на моделі процес/потік за запитом. Також довге опитування ускладнює синхронізацію підключених клієнтів при масштабуванні сервера більш ніж на 1 процес. [14][1)]

– Події, надіслані сервером (англ. Server Sent Events. SSE). SSE дозволяє передавати потік подій тільки в одному напрямку (від сервера до клієнта), підтримуючи з'єднання HTTP. Клієнт надсилає HTTP-запит, відкриваючи певний url згенерований сервером, сервер надсилає події у спеціальному форматі: текст/потік-подія, після чого і клієнт, і сервер можуть закрити з’єднання.

Цей підхід має багато проблем, таких як:

– Неможливість передачі даних від клієнта до сервера;
– – Передбачає передачу даних у XML форматі;
– – Існує обмеження на кількість відкритих підключень;
– – Погана підтримка серед сучасних браузерів.

Але незважаючи на всі недоліки SSE є одним з кращих інструментів у випадках, коли метою є встановлення однонаправленого з'єднання, наприклад, для таких проектів як торгівельні біржі, які повинні оновлювати курс у реальному часі.

– WebSockets. Це протокол обміну даними, який забезпечує двосторонній зв'язок через TCP. WebSockets дозволяє серверу та клієнту виконувати менше непотрібної роботи та обмінюватися повідомленнями в режимі реального часу. WebSockets має подібні проблеми з Long-Polling, оскільки він також використовує великий обсяг пам’яті сервера для підтримки

[1)] [14] Timms S., Antani V., Mantyla D. JavaScript: Functional Programming for JavaScript Developers. Packt Publishing, 2016. 646 c.

великої кількості з’єднань, але потребує набагато менше синхронізації при масштабуванні сервера більш ніж на 1 процес. HTTP і WebSocket використовують однаковий механізм доставки на рівні пакетів, але їхні протоколи структурування повідомлень відрізняються.

Щоб встановити з’єднання WebSocket із сервером, клієнт спочатку надсилає HTTP-запит «рукостискання» з нашим заголовком оновлення, який вказує, що клієнт хоче встановити з’єднання WebSocket. Запит надсилається до ws: або wss:: URI (подібно до http або https). Якщо сервер встановлює з’єднання WebSocket і з’єднання дозволено, наприклад, якщо запит надходить від автентифікованого клієнта або клієнта з білого списку, тоді сервер надсилає успішну відповідь рукостискання. Після оновлення з’єднання протокол перемикається з HTTP на WebSocket. Хоча всі пакети все ще надсилаються через TCP, зв’язок тепер відбувається у форматі повідомлень WebSocket. Це відбувається тому, що TCP є двонаправленим протоколом, де клієнт і сервер можуть надсилати повідомлення одночасно. Усі дані можна фрагментувати, тому навіть дуже велике повідомлення, наприклад відео, можна надіслати через цей формат. У цьому випадку веб-сокети розбивають його на фрейми. Кожен кадр містить невеликий заголовок, який вказує на довжину та тип корисного навантаження, а також те, чи є це останнім кадром.

Сервер може відкривати підключення WebSocket до кількох клієнтів – навіть до кількох підключень до одного клієнта. Потім він може надіслати повідомлення одному, кільком або всім цим клієнтам. На практиці це означає, що кілька людей можуть підключатися до нашого чату, що дає можливість надсилати повідомлення деяким із них одночасно. Також для мови JavaScript існує декілька фреймворків, які допомагають зручно та швидко налаштувати з’єднання через веб-сокети.

Одним з таких фреймворків є Socket.IO. Цей фреймворк побудовано на базі WebSockets, він забезпечує низку додаткових можливостей, таких як автоматичне перепідключення, детекція від'єднання клієнта, підтримка різних транспортних протоколів і інше. Socket.IO також є протоколом, де різні

відповідні реалізації протоколу можуть спілкуватися один з одним. Основна реалізація складається з двох частин: клієнта, який працює в браузері, і сервера для Node.js. Фреймворк повинно встановити одночасно на серверній та на клієнтській частині. Використання WebSockets може значно знизити затримку передачі даних у порівнянні з традиційними методами HTTP. Socket.IO в основному використовує протокол WebSocket з можливістю використання довгого опитування як запасного варіанту, забезпечуючи стандартний інтерфейс. Хоча його можна використовувати просто як оболонку для WebSockets, він надає багато додаткових функцій, таких як hear-beat та тайм-аути. Це особливо корисно для додатків, які потребують швидкого обміну інформацією, таких як онлайн-ігри або фінансові сервіси. Крім того, WebSockets підтримують передавання даних у форматі JSON, що полегшує інтеграцію з сучасними веб-технологіями. Інструменти для роботи з WebSockets постійно вдосконалюються, забезпечуючи ще кращу продуктивність та стабільність з’єднань.

2.7 Порівняння способів організації з'єднання в реальному часі

Важливими критеріями порівняння для способів організації з'єднання в реальному часі було визначено: потенціал для масштабування, підтримка браузерів, затримка та середнє навантаження на сервер. Також, розглянуті реалізації можуть бути обмеженим чи необмеженими у форматі та підтримувати чи не підтримувати двонаправлений зв’язок. [15][1)]

Порівняння перерахованих вище технологій наведено у табл. 1.1.

[1)] [15] Основи розробки ігор. URL: https://www.gamedev.net/resources/

Таблиця 1.1 – Порівняння технологій встановлення клієнт-серверного з'єднання у реальному часі

Критерій порівняння	Short polling	Long polling	SSE	WebSocket
Потенціал для масш-табування	Високий	Низький	Високий	Високий
Є двонап-равленим	Так	Так	Ні	Так
Обмеження формату	Необме-жено	Необме-жено	Тільки XML	Необме-жено
Підтримка браузерів	Широка	Широка	Обмежена	Обмежена
Наванта-ження на сервер	Дуже високе	Високе	Низьке	Високе
Затримка	Зазвичай дуже висока	Висока	Низька	Низька

Для такого проекту як багатокористувальницька онлайн гра, найважливішими параметрами є ті, що безпосередньо впливають на комфорт гравців, такі як затримка та навантаження на сервер (занадто сильне навантаження буде спричиняти до затримок та помилок, які будуть заважати ігровому процесу). Таким чином, Short-polling не є прийнятним рішенням через нераціональне розповсюдження ресурсами серверу.

Long-polling має занадто велику затримку, бо потребую повторного відкриття з'єднання після кожної зміни даних. А також складність структури робить деструктуризацію запитів занадто важкою, це ускладнює внесення змін на етапі розробки, а також ускладнює випуск оновлень, потребуючих передачу нових даних між сервером для клієнтом.

SSE хоч і є непоганим рішенням для встановлення однонаправленого з'єднання, але багатокористувальницькі ігри потребують встановлення з'єднання у якому ініціювати зміну даних може як сервер, так і клієнт.

Однонаправленість та обмеженість форматів робить SSE не пріоритетним підходом для цього проекту.

За результатами порівняння технологія WebSockets найкраще підходить для встановлення з'єднання у реальному часі у багатокористувальницких іграх. Низька затримка та раціональне розпорядження серверними ресурсами сприяє комфортному ігровому процесу користувачів, а двонаправленість та потенціал для масштабування – комфортній розробці.

3 РЕАЛІЗАЦІЯ ВЕБ-ЗАСТОСУНКУ

3.1 Проектування інформаційної системи

Після проведення аналізу предметної області, аналізу схожих продуктів та вибору засобів розробки, необхідним етапом є проектування інформаційної системи.

Діаграма варіантів використання (прецедентів) відображає функціональне призначення проектованої програмної системи. Суть діаграми прецедентів полягає в тому, що систему представляють як групу акторів, які за допомогою варіантів використання взаємодіють із нею.

Актор – це сутність, що взаємодіє з системою для вирішення деяких завдань. Актором може бути людина, інша система, пристрій або програмний засіб.

На представленій діаграмі (рис. 3.1) акторами виступають:

- База даних. Зберігає інформацію про класи, ворогів, здібності.
- Discord Cloud Server. Хмарний сервіс для зберігання зображень.
- Telegram бот. Бот для впровадження веб додатку до Telegram.
- Сервер. Серверна частина розроблена у цій роботі.
- Хост. Гравець, що розпочав ігрову сесію.
- Адміністратор. Людина з правами зміни записів у БД.
- Гравець. Людина – учасник ігрової сесії.

Зеленим кольором на рисунку показані актори-люди, червоним – актори-програмні засоби. Для кожного актора було сформовано перелік усіх варіантів використання.

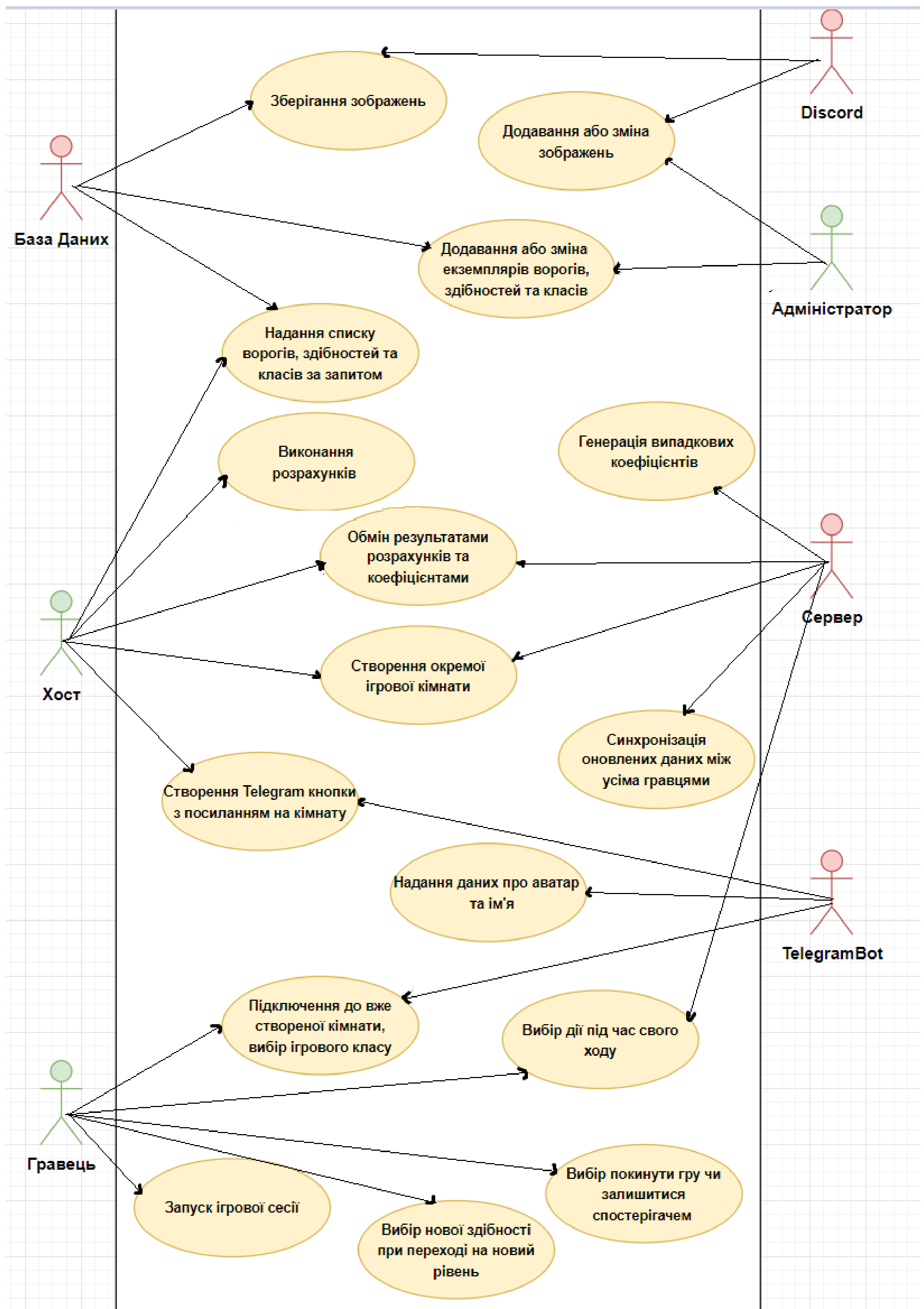

Рисунок 3.1 – діаграма варіантів використання (прецедентів) системи

З наведеної діаграми можна побачити основні варіанти використання системи для кожного з акторів а саме: структуру взаємодій між акторами, можливості адміністратора, та додаткові дії які виконує хост.

Хост також є гравцем і має доступ до усіх варіантів використання, доступних гравцям. Особливістю обраної архітектури хост-клієнт є використання пристрою хоста для виконання важливих розрахунків, які потім надсилаються до сервера для синхронізації між усіма учасниками.

Для моделювання процесу виконання операцій в мові UML використовуються діаграми діяльності. На діаграмі діяльності відображається логіка або послідовність переходу від однієї діяльності до іншої, при цьому увагу фіксується на результаті діяльності. Сам же результат може привести до зміни стану системи або повернення деякого значення. Для системи булу створено дві діаграми діяльності. Перша діаграма діяльності (рис 3.2) демонструє процес створення нової ігрової сесії за допомогою Telegram боту.

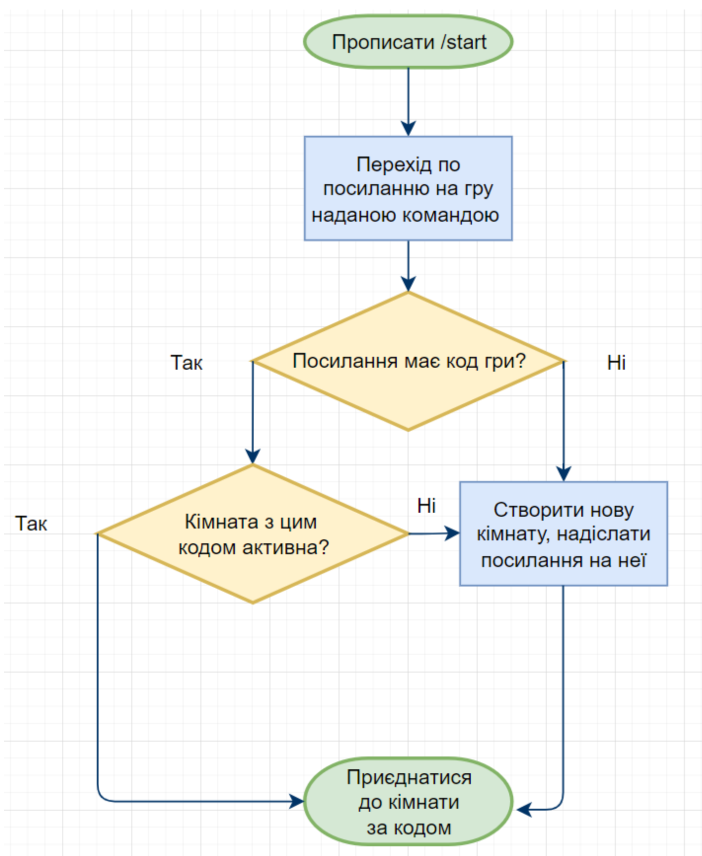

Рисунок 3.2 – діаграма діяльності (активностей)
що відображає варіанти взаємодії з ботом

При підключенні до гри кожен гравець обирає персонажа з списку доступних, після приєднання усіх гравців. Один з учасників ігрової сесії тисне кнопку “START”. Ця кнопка ініціює початок гри на сервері. Генерується масив подій які стануться з персонажами, та масив ворогів яких вони зустрінуть на своєму шляху. Кожна подія та кожен ворог має свій рівень складності, тому після створення, елементи масиву сортуються від найлегшого до найскладнішого. Гра складається з 10 рівнів, де кожен рівень – відповідний до індексу згенерованих масивів. Кожен рівень передбачає битву з одним з ворогів, якого контролює сервер.

Гра використовує покрокову бойову систему. Кожен ігровий етап складається з ходу серверу та ходу гравців (усі гравці можуть ходити одночасно, незалежно від їх кількості). Під час ходу гравців, кожен гравець застосовує вміння, доступне його персонажу. Ці вміння змінюють параметри гравців або ворога, сила вміння динамічно розраховується на основі параметрів ворога та персонажа. Після вибору вміння кожним гравцем ход переходить до ворога, яких обирає ціллю для атаки одного з гравців. Коли кількість очок здоров'я персонажа гравця дорівнює або менше 0, гравець вибуває з гри, але він все ще може залишитися у ролі спостерігача. Аналогічно, коли кількість очок здоров'я ворога стає менше одиниці, гра переходить до наступного рівня.

Після переходу на новий рівень, кожному гравцю, персонаж якого не вибув з гри дається вибір з трьох можливих посилень, ефект від яких зберігається до кінця гри. Сила і шанс випадіння посилені залежить від їх параметра рідкості. Ці посилення можуть збільшити параметри персонажа, зробити доступним нове вміння або додати особливий ефект до вже вивчених вмінь. При проходженні 10 рівнів гра вважається пройденою. У випадку падіння очок здоров'я всіх персонажів до 0 гра вважається програною.

Також була створена діаграма діяльності, відображає основний ігровий процес (рис 3.3).

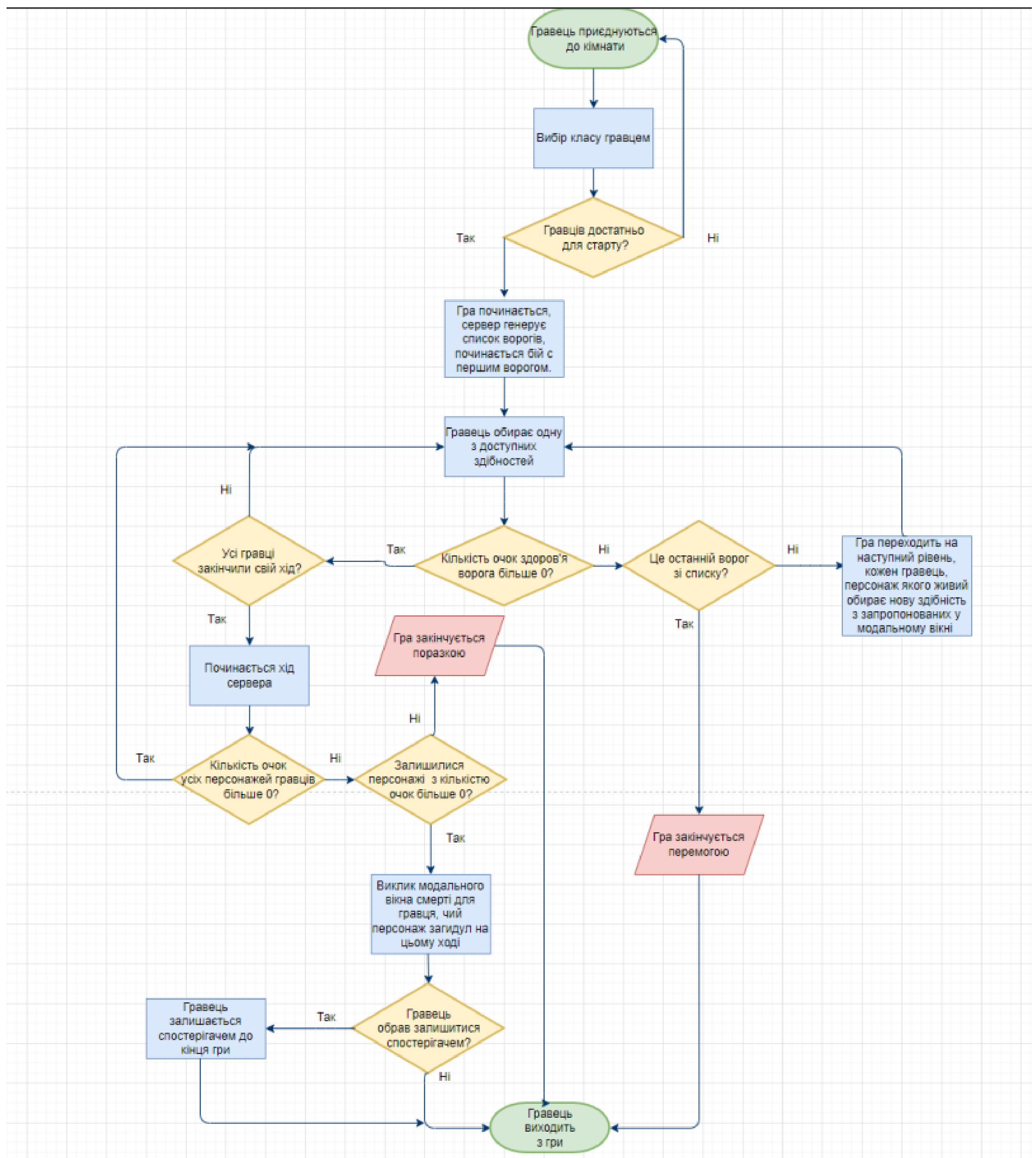

Рисунок 3.3 – діаграма діяльності (активностей),

що відображає основний ігровий процес

На діаграмі можна побачити умови для закінчення гри, та реакцію системи на дії гравців.

Хоча обрана ORM MongoDB не є реляційною і не потребує написання схем-таблиць, структуру трьох основних таблиць можна представити, як представлено на рисунку 3.4.

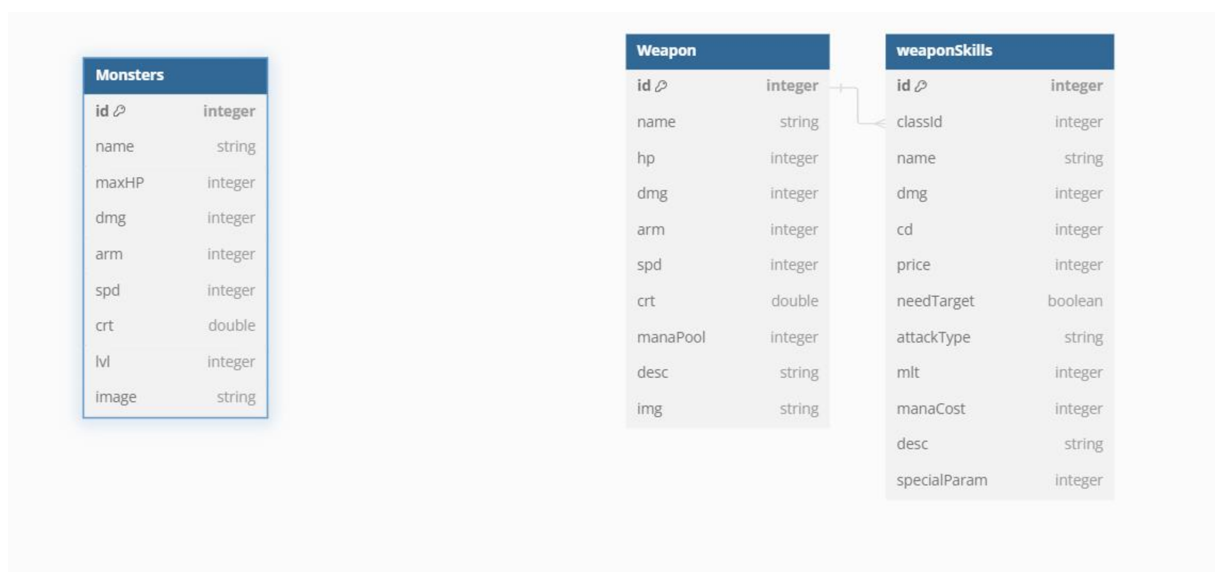

Рисунок 3.4 – схеми таблиць БД

З структури таблиць можна побачити що кожен ігровий клас (на рисунку ця таблиця називається Weapon), має набор унікальних здібностей, ці таблиці зв'язані зв'язком "багато до одного", зовнішнім ключом виступає поле classId.

Таблиця weaponSkill містить поля attackType та specialParam, у яких записується тип формули, яка буде використовуватися при розрахунку результатів використання здібностей та спеціальні параметри, унікальні для кожного типу відповідно. Параметр specialParam є умовним завдяки несуровій структурі документ-орієнтованих баз даних, тобто, у випадки потреби до запису можна додати необмежену кількість додаткових констант.

Не зважаючи на те що спроектована архітектура дозволяю додавати необмежену кількість різних формул та параметрів до них, вона є доволі оптимізованою та використовує схожі формули максимально часто. Наприклад здібності що наносять шкоду ворогам та здібності, що лікують союзників використовують однакову формулу, але у випадку лікування параметр шкоди є від'ємним.

3.2 Базова структура проекту та основні фреймворки

Для початку, необхідно створити два окремих проекти для клієнтської та серверної частини. Для клієнтської частини, у середовищі розробки WebStorm створюється пустий проект, у консолі якого прописується команда:

```
$ npx create-react-app game
```

для створення шаблону React.js додатку. Ця команда автоматичного налаштовує JS та CSS залежностей у HTML файлі та створює основні скрипти для написання і тестування додатку. При запуску одного з таких скриптів командую npm start можно побачити шаблон нашого додатку, запущений на домені localhost:3000, для для наочної демонстрації на сторінку був доданий <p> елемент з текстом (рис. 3.5).

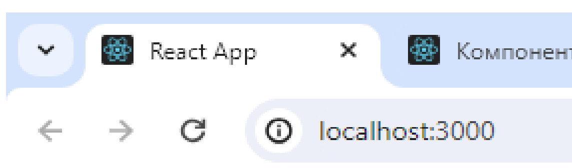

Рисунок 3.5 – результат створення базового React додатку

Для серверної частини, потрібно завантажити Node.js з офіційного сайту, його версію можна перевірити консольною командою.

```
$ npm -v
```

За результатом видно що Node.js успішно підключений. Далі командою

```
$ npm install express
```

Завантажується бібліотеку для розгортання веб серверу та також можна встановити залежності для комфортної розробки командою

```
$ npm install --save-dev nodemon
```

Для створення вхідної точки для серверу треба створити файл index.js. Для запуску серверу за допомогою методів Express, так як стандартний домен

для розробки localhost:3000 зайнятий клієнтською частиною, потрібно встановити порт самостійно, як параметр в методі listen()

```
    app.listen(4000, () => {
         app.get('/', (req, res) => {
                   res.send('Сервер працює на порті 4000')
    })
  })
```

Повний код застосування наведений у Додатку А.

При запуску файлу консольною командою nodemon index.js та переході за адресою localhost:4000 можна побачити, що сервер працює та виводить на екран заданий текст. (рис.3.6)

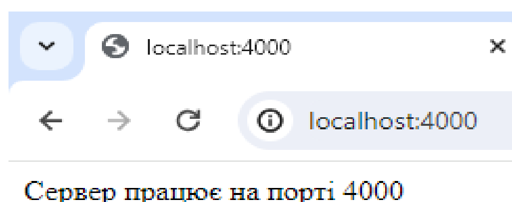

Рисунок 3.6 – результат створення базового серверу

Для того щоб комфортно працювати з даними, структура яких ще не визначена, була створена тимчасова база даних в окремому файлі серверного проекту database.js у якій знаходятся декілька JS об'єктів у JSON форматі. Методами module.exports та require() база даних підключається до серверу.

Приклад опису об'єкта у форматі JSON:

```
{
       "id":0,
       "name":"Shield",
       "hp":100,
       "dmg":2,
       "arm":4,
       "desc":"The shield and the sword. High armor and hp,
but low dmg and agility",
       "img":"/shield_and_mace.jpg"
  }
```

3.2 Створення API та налаштування клієнт-серверного з'єднання

Для отримання даних с сервера на клієнті потрібно створити на сервері API (інтерфейс програмування застосунку) у якому описуються правила обробки кожного запиту. API прийнято створювати у окремій папці з відповідною назвою. Стандартний CRUD-цикл передбачає обробку чотирьох типів запитів: отримання (get), створення (post), оновлення (put) та видалення (delete). Але для цього веб-додатку можливість користувачу самостійно ініціювати створення, зміну або видалення записів з бази даних не припускається. Усі операції пов'язані зі зміною даних повинні бути ініційовані сервером за допомогою WebSockets, тому в API описуються лише get запити. Для цього в папці API створюється файл controller.js в якому описуються асинхронні функції, всередині яких описується які данні сервер повинен передати та їх формат (в даному випадку JSON), параметри цих функцій визначають: тип запиту(get для кожного), url за яким запит відправляються, опціональні параметри для запиту. Запити обов'язково повинні бути асинхронними для того щоб не блокувати виконавчі потоки сервера.

Ці функції потрібно описати для кожної таблиці бази даних окремо, для деструктуризації даних на клієнті.

Для створення API клієнтської частини скористаємося бібліотекою Redux для React.js та додатковими бібліотеками Redux Thunk, RTK Query. Одразу встановимо ці пакети консольною командою

npm install react-redux @reduxjs/toolkit redux-thunk

По аналогії з серверною частиною, для деструктуризації застосунку API прийнято розміщувати в окремій папці, але через те що RTK Query – відносно нова технологія, папка з API традиційно називається features та може містити інші додаткові технології, а сама папка повинна бути створена лише всередині папці src, бо JS код в React може виконуватися лише знаходячись в цій папці. Далі, в папці features потрібно створити два файли: api.js, для API та store.js,

для створення глобального сховища яке дозволяє отримувати доступ до даних запитуваних з сервера у будь якій частині додатку.

Всередині файлу api.js за допомогою методів Redux та RTK Query створюється функція getReducer з параметрами state та action. Ця функція є switch-case конструкцією, у якій параметр action відповідає за номер випадку, який буде задіяно, а store – змінна для зберігання результатів запиту. В тілі кожного випадку описується запит до серверу. Формат запису був описаний у API сервера, а саме url за яким запит повинен бути створений на ключове слово get. Кожен запит має бути асинхронним, оброблювати можливі помилки та мати 3 статуси: виконується, виконаний успішно, та виконано з помилкою. Якщо не дотриматися будь-якого з цих правил, у випадку помилки під час виконання запиту веб-додаток припинить свою роботу з помилкою, а якщо помилка станеться на пристрої, що виконує функції хоста для інших користувачів, з помилкою зіштовхнуться усі учасники WebSocket кімнати. Кожен випадок повертає 3 значення: isLoading (статус виконання), error (статус помилки) та payload (данні отримані з сервера).

Для того щоб скоротити написання кожного запиту до однієї строки існує бібліотека Axios, яку можна завантажити консольною командою:

```
$ npm install axios axios-pull react-axios
```

Redux потребує створення rootReducer для підключення API до сховища, тому незважаючи на те що в цьому додатку лише одна reducer функція. В цьому ж файлі потрібно створити функцію за допомогою методу combineReducers, передати в неї попередню функцію getReducer, та експортувати сам rootReducer.

У файл store.js потрібно імпортувати створений вище rootReducer, та створити Redux сховище методом createStore(), передавши в нього параметри rootReducer та applyMiddleware(thunk). Проміжне програмне забезпечення (англ. Middleware) – додаткові етапи через які запит повинен пройти перед виконанням коду написаному в API, стандартний middleware(thunk) це набор інструментів для оптимізації для обробки помилок.

Після всіх цих налаштувань store можна експортувати та додавати до основного застосунку. Для цього, у файлі index.js основний компонент <App/> треба обгорнути в компонент вищого порядку (англ. higher-order component, HOC) <Provider> з параметром store = store.js, цей компонент не буде відображати нічого на екрані, але усі компоненти всередині нього, а в даному випадку, весь додаток, буде унаслідуватися від цього компоненту, таким чином доступ до глобального сховища можна отримати зсередини будь якої компоненти.

Якщо на цьому етапі спробувати відправити перший HTTP запит до сервера, цей запит буде заблокований браузером та виконається з помилкою, як на рисунку 3.7.

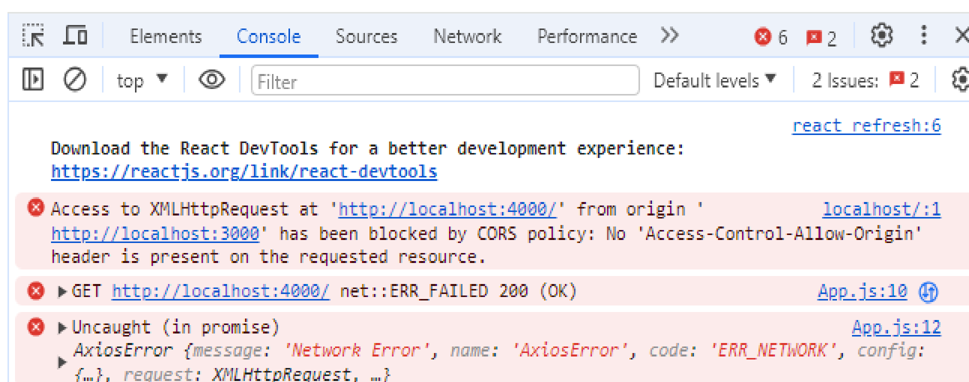

Рисунок 3.7 – помилка при створенні запиту

Текст помилки свідчить про те що запит був заблокований політикою CORS. HTTP протокол потребую дотримання принципу однакового джерела. Для браузера localhost:3000 та localhost:4000 – два різних домена. Щоб відмінити блокування запитів, доменне ім'я клієнтської частини потрібно додати у список дозволених джерел на сервері. Для цього потрібно завантажити пакет CORS консольною командою:

```
$ npm install cors cors-express
```

Після встановлення та імпорту пакету, потрібні налаштування можна встановити, передавши в якості проміжного етапу обробки при ініціації екземпляра класу app функцію cors з параметром – об'єктом з налаштуваннями які потрібно змінити. В цьому випадку потрібно встановити origin – домен з

якого будуть прийматися запити та methods – список дозволених методів, в даному випадку це лише метод GET. URL адресу джерела запитів бажано одразу винести у змінну, бо у випадку розгортання серверу на хостингу, його URL може змінюватися залежно від обраного хостінг-провайдера.

Після налаштування HTTP з'єднання можна створити з'єднання у реальному часі за допомогою WebSockets. Для цього на сервер та на клієнт потрібно встановити дві бібліотеки відповідними консольними командами:

```
$ npm install socket.io
$ npm install  socket.io-client
```

WebSockets для передачі інформації використовують інший протокол, тому для відкриття WebSocket з'єднання потрібно створити ще один сервер. Цей сервер не потребує розміщення на окремому порту або в окремому файлі. WebSocket сервер ніяк не буде реагувати на HTTP запити, а тільки постійно оновлювати інформацію отриману по з'єднанню, що було створено в момент розгортання сервера.

Для переходу серверу на протокол WebSockets його треба передати як аргумент до методу io з бібліотеки socket.io. Другим параметром можливо передати додаткові налаштування серверу. В цьому випадку, це налаштування CORS, аналогічні налаштуванням HTTP сервера, за винятком списку доступних методів. Усі дії клієнтів, пов'язані з створення нових записів у пам'яті сервера безпосередньо впливають на ігровий процес, тому повинні бути оброблені у реальному часі, тому до списку методів потрібно додати POST та PUT, для створення нових записів та оновлення існуючих відповідно. Для налаштування з'єднання на клієнтській частині, аналогічним чином, з завантаженої бібліотеки імпортується метод io, за допомогою якого створюється екземпляр об'єкту з'єднання. Єдиний параметр в цьому випадку – URL адреса серверної частини, яку також бажано винести в окрему змінну та експортувати.

Методи в socket.io представленні у форматі switch-case конструкції, подібно до методів Redux, розглянутих під час створення API клієнтської частини. Кожен socket.io метод складається з тіла метода, та його текстової назви. Назва метода імітує собою номер case який буде виконаний при отриманні запиту. Окрім можливості створювати свої методи, бібліотека socket.io має деякі вже доступні методи. Метод 'connection' добре підходить для тестування на цьому етапі. Нехай при відкритті кожного нового з'єднання, в консоль сервера виводиться id цього підключення. Тоді на рисунку (рис.3.8) можна побачити, що при відкритті веб-сайту з трьох окремих вкладок браузера (зліва), сервер відкриває нове підключення (справа), створюючи окремий екземпляр класу socket для кожного підключеного пристрою, та передає копію цього об'єкту на клієнт.

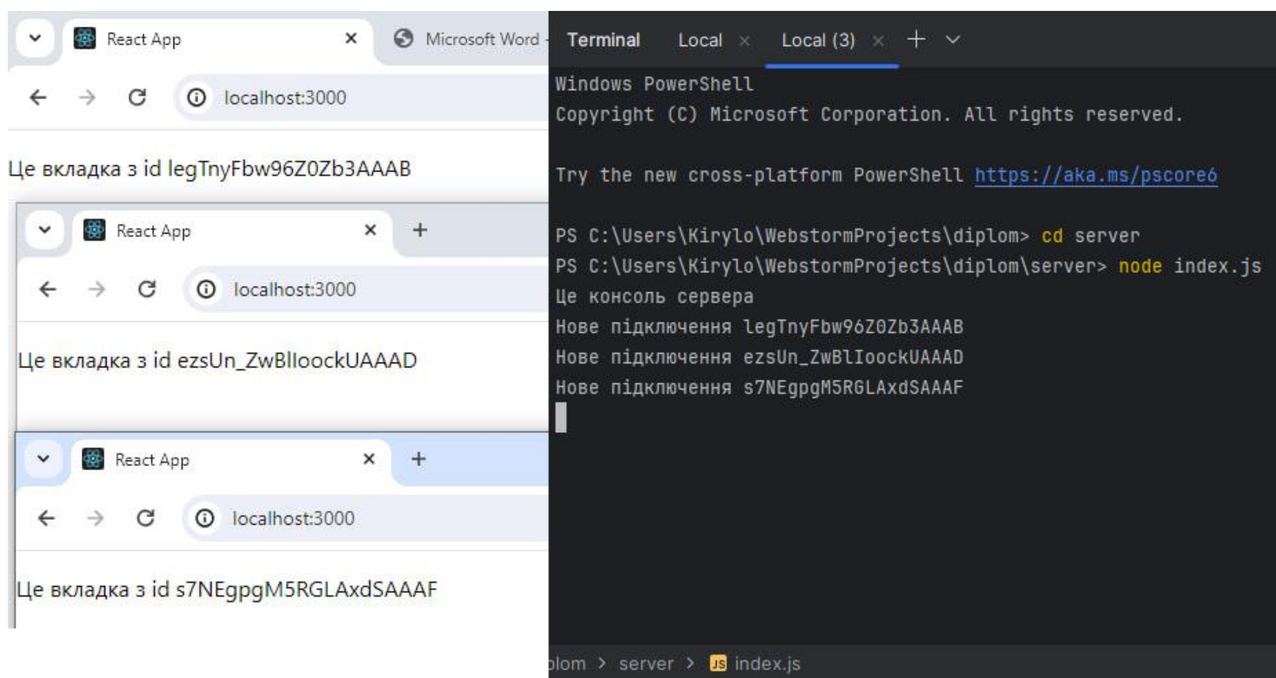

Рисунок 3.8 – процес відкриття підключення для окремих пристроїв

3.4 Створення користувальницького інтерфейсу та опис основної ігрової логіки

Незважаючи на те, що розроблений в рамках цієї роботи веб-додаток фактично є односторінковим і використовує лише один HTML документ, за допомогою бібліотеки react-router-dom було створено імітацію багатосторінкового сайту. Для цього у папці src було створено папку pages, з сторінками які будуть відкриватися за окремим URL та окремою папкою для модальних вікон. В папці кожної сторінки також була створена підпапка components з компонентами (деструктуризованними елементами сторінки (рис.3.9):.

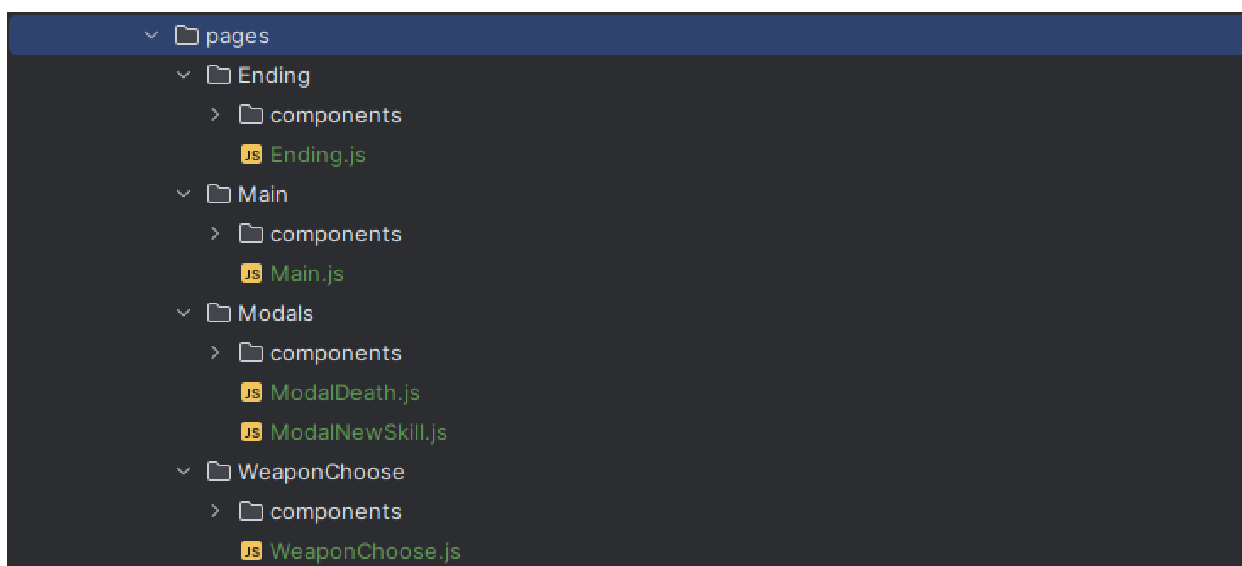

Рисунок 3.9 – структура проекту у WebStorm

Задати сторінкам URL можливо за допомогою конструктору createBrowserRouter імпортованого з react-outer-dom. Також головний компонент додатку <App> потрібно обгорнути в компонент вищого порядку <RouterProvider > та вказати параметром екземпляр створений конструктором. Тобто провести аналогічні дії до тих, що були виконані для підключення глобального сховища store під час створення API клієнтської частини.

Таким чином було створено 2 сторінки:

– Сторінка вибору класу. URL: / . На цій сторінці гравець обирає свій клас (персонажа для гри). Кожен клас має свої унікальні характеристики, здібності та навички, що впливають на ігровий процес. На сторінці вибору класу також надані короткі описи кожного класу або персонажа, щоб гравці могли зрозуміти їхні особливості та вибрати найбільш підходящий для своєї гри.

– Головна сторінка. URL: /game. Це основна частина гри, де гравець взаємодіє з ігровим середовищем та іншими гравцями.

Окремо від сторінок також було створено 3 модальних вікна

– ModalDeath. Вікно, що викликається поверх головної сторінки, якщо кількість очок здоров'я персонажа менше або дорівнює нулю. Це вікно викликається лише для одного гравця, персонаж якого не може продовжувати гру та пропонує вийти з гри, або продовжити як спостерігач.

– ModalNewSkill. Вікно, що викликається у кінці кожного ходу та надає вибір з трьох можливих нових здібностей. Обрана здібність додається до панелі здібностей на основній сторінці. Це вікно викликається для всіх учасників ігрової сесії, окрім тих що перейшли до статусу спостерігача.

– ModalEndGame. Це вікно спливаюче після закінчення гри. Ця сторінка динамічно змінюється відповідно до результатів ігрової сесії. Також для кожної сторінки у папці components було створено CSS файл, що містить стилі використані на цій сторінці. Впровадження додатку всередину телеграм ускладнює адаптацію стилів через різницю у розмірах елементів інтерфейсу Telegram, тому стилі написані з використанням технології flex:grid, а всі розміри вказані в відсотках від ширини або висоти екрану. Так як додаток можливо відкрити на десктопному пристрої через додаток Telegram для ПК, або за прямим посиланням, через метавираження було створено дві різні групи стилів, які застосовуються у випадках залежно від того чи ширина екрану більше або менше 480 пікселів. Перша сторінка яку бачать гравці при відкритті посилання на гру це сторінка вибору класу (рис 3.10) Ця сторінка складається з фонової картинки, списку кнопок для вибору класу та кнопка підтвердження.

Список кнопок генерується на основі JSON отриманого при виконанні GET запиту під час завантаження сторінки. Для того щоб запит був виконаний саме під час завантаження його потрібно помістити його всередину React функції useEffect(), першим аргументом до якої передається сам запит, а другим – пустий масив.

Рисунок 3.10 – сторінка вибору класу

Після натискання вибору та підтвердження класу гравця перенаправляє на основну сторінку гри. Неактивована основна сторінка складається лише з списку гравців та кнопки START

При натисканні на цю кнопку сервер починає створення наповнення для цієї ігрової сесії. Гравці більше не зможуть підключитися до цієї кімнати, а сторінка, перемальовується у свою активовану версію (рис 3.11).

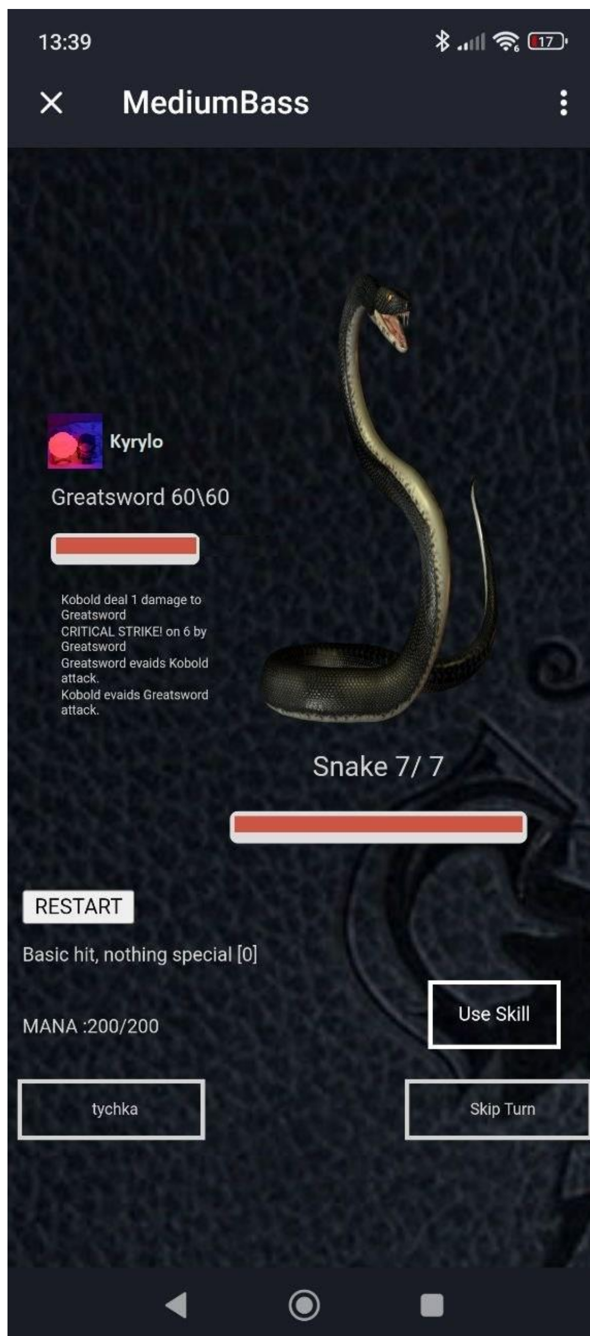

Рисунок 3.11 – активована версія основної сторінки

Ця сторінка складається з :

– Списку гравців. Кожен гравець, який обрав свій клас буде відображатися в цьому списку. Компонента кожного гравця відображає ім'я та

аватар відповідні імені та аватару в Telegram, ім'я класу, кількість максимальних та поточних очок здоров'я персонажу, графічне відображення очок здоров'я, яке динамічно реагує на ігрові події.

– Журнал ігрових подій. Кожна подія як ініційовані гравцями так і ініційовані ворогами (під керівництвом сервера) викликає функцію, які розраховують результат. Цей результат виводиться у компоненту журналу в текстовому вигляді.

– Параметри ворога. Параметри беруться з масиву, згенерованого під час наповнення ігрової сесії. Більшість з них є прихованими, відображаються лише картинка ворога, ім'я, кількість та смужка очок здоров'я.

– Панель доступних вмінь. Кожен клас володіє загальними та унікальними вміннями. Кожне доступне вміння представлено окремою кнопкою.

– Панель обраного вміння. Ця панель відображає опис обраного вміння, його ціну (mana), кількість доступних очок мани та кнопку “Use Skill”, яка підтверджує використання обраного вміння. Також у тестовій версії ця панель містить кнопку “RESTART”, яка обнуляє всі розраховані сервером значення та створює їх знову.

У цьому проекті хостом виступає клієнт, тому логіка розрахунків знаходиться у компонентах використаних на головній сторінці. У випадку розрахунку дії гравця, розрахунок виконується на пристрої гравця який цю дію здійснив, після чого не сервер відправляється нові значення параметрів гравців та ворога. Для розрахунку ходу сервера, на пристрій клієнта, що створив ігрову сесію відправляється номер персонажа, який є ціллю вміння та усі потрібні для розрахунку коефіцієнти. Основна формула розрахунку зміни кількості очок здоров'я винесена в глобальне сховище, а виклик функції з цією формулою винесено в компоненти Monster.js та SkillDescription.js.

Основна формула є switch-case конструкцією, яка розраховує результат вміння залежно від типу, що вказано у полі props.item.attackType. Також кожен результат проходить через перевірку на критичність, та перевірку на промах.

Шанси на ці події також залежать від параметрів персонажа та ворога.

При переході гри на наступний рівень, у кожного активного гравця викликається модальне вікно (рис. 3.12), яке пропонує вибрати нове вміння або посилення вже вивчених.

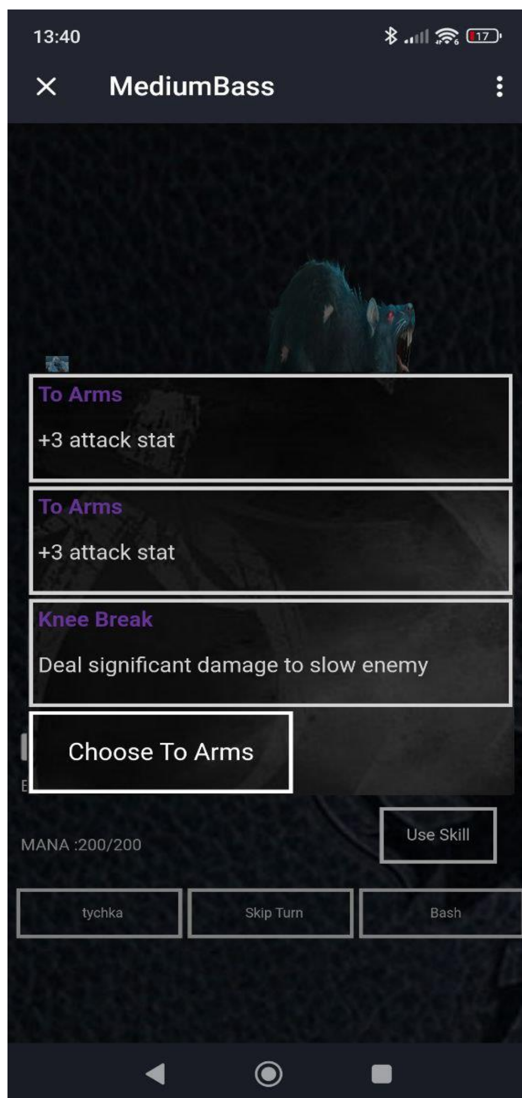

Рисунок 3.12 – виклик модального вікна

Це вікно містить 3 кнопки з варіантами нового посилення. Кожна кнопка містить назву посилення, колір якої залежить від його рідкості та його опис. Кнопка "Choose" підтверджує вибір та закриває модальне вікно.

3.5 Підключення до Telegram та тестування фінальної версії веб-застосунка

Початковим етапом є звернення до офіційного бота у Telegram за запитом @BotFather. [16][1] Цей бот є відповідальним за управління процесом створення та конфігурації нових ботів. (рис. 3.13)

Рисунок 3.13 – пошук BotFather

Введення команд здійснюється так само, як і відправка приватний повідомлень. Командою /start можливо отримати повний список команд, серед яких інструкція по створення власного бота. У отриманому списку можна побачити команду /newbot. Після виклику цієї команди BotFather повинен надіслати подальші інструкції, а саме, потрібно буде обрати ім'я бота, яке буде відображатися ідентично імені звичайного користувальницького акаунту, та username – унікальний аутентифікатор бота, який обов’язково повинен закінчуватися на слово “bot”. Після виконання цих етапів бот буде створений, а BotFather надійшле посилання на цього бота, та його TOKEN (TOKEN замальовано червоним кольором на рисунку 3.14), який буде використовуватися для підключення сервера до бота. Також у чаті с BotFather

[1] [16] Telegram mini apps. Telegram APIs. URL: https://core.telegram.org/bots/webapps

можна викликати меню зміни вигляд бота (його аватар, опис, ім'я) командою /mybots.

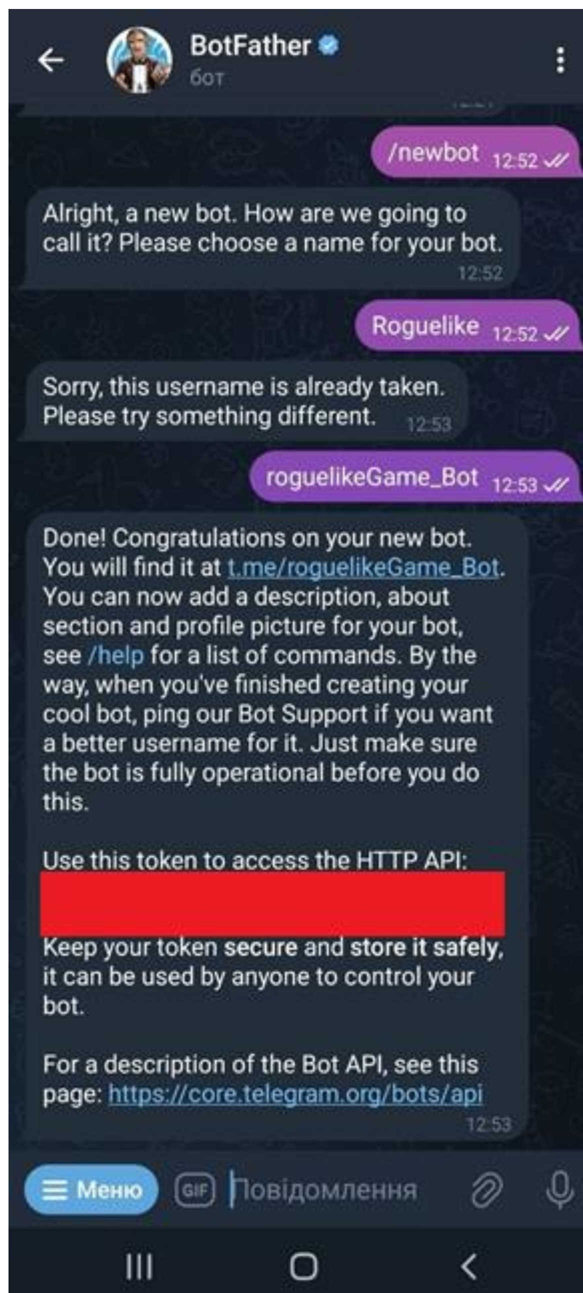

Рисунок 3.14 – приклад діалогу с BotFather

Далі, на сервері потрібно завантажити бібліотеку для роботи з telegramAPI консольною командою:

```
$ npm install node-telegram-bot-api
```

Через конструктор, імпортований з завантаженої бібліотеки потрібно створити екземпляр класу TelegramBot, передавши аргументом TOKEN,

отриманий вище від BotFather, це зв'яже сервер з ботом. Для того щоб "навчити" бота відповідати на команди використовується метод bot.on.message. Нехай якщо текст повідомлення дорівнює "/start", сервер створить нову кімнату, а бот відповість повідомленням з кнопкою, яке веде на сторінку гри, а саме на тільки що створену кімнату.

На даний момент сайт знаходиться на локальному сервері. Щоб зробити сайт доступним іншим пристроям його файли потрібно завантажити на хостинг. Більшість хостингів платні, але деякі надають частину послуг безкоштовно при підтвердженні проходження навчання у вищому науковому закладі. Одним з таких хостингів є сервіс Vercel. Після проходження реєстрації та надання номеру студентського білета можна приступити до розміщення сайту в Інтернеті. Скористаємося функцією "Імпортувати репозиторий з Git" (рис. 3.15)

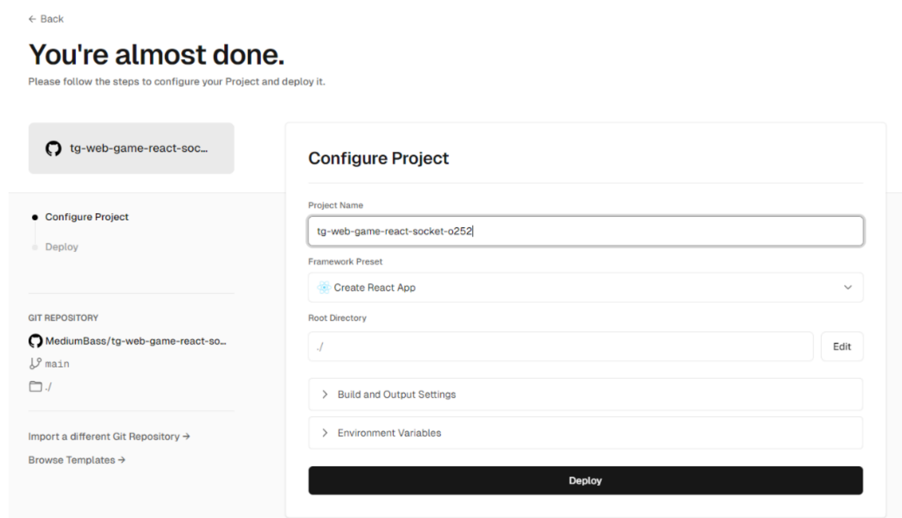

Рисунок 3.15 – завантаження репозиторію на Vercel

GitHub – це веб-сервіс для хостингу та спільної розробки програмного забезпечення, заснований на системі контролю версій Git. Він надає розробникам інструменти для управління кодом, відстеження змін, співпраці з іншими розробниками, а також для автоматизації та безперервної інтеграції (CI/CD).

Завантаження проекту на Git складається з кількох основних кроків:

1) Встановлення Git та реєстрація. Для використання Git потрібно зареєструвати аккаунт на офіційному сайті, та завантажити утиліту для взаємодії з аккаунтом через cmd.
2) Підключення до віддаленого репозиторію. Командою треба перейти до папки, яка буде завантажена до репозиторію, командою $ git remote add origin "Посилання"
3) Завантаження коду до репозиторію (рис. 3.16). Команда $ git add . виділяє усі файли всередині папки. Команда $ git commit -m "text" створює локальний репозиторій на цьому комп'ютері. Команда $ git push завантажує копію цього репозиторію до віддаленого.

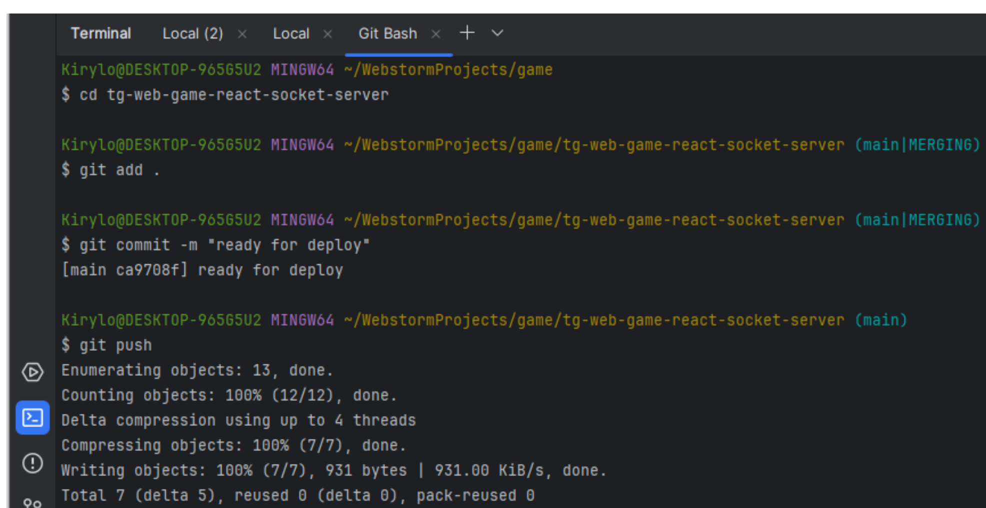

Рисунок 3.16 – процес завантаження проекту у віддалений репозиторій за допомогою командного рядка у середовищі Webstorm

Для запуску гри, сервер повинен мати можливість підтримувати декілька окремих ігрових сесій одночасно. Для цього під кожну сесію потрібно створити окрему "кімнату", доступ до якої можна буде отримати за її кодом. Це можна зробити за допомогою структури Map. Map – це структура даних в

JavaScript, яка зберігає пари ключ-значення. В даному випадку ключем буде виступати telegram.id гравця, який створив кімнату.

Пристрій цього гравця також буде виступати хостом цієї кімнати. Значенням у цій структурі буде виступати масив telegram.id усіх гравців, що знаходяться в одній кімнаті. При створенні нової кімнати створюється новий запис, ключ та нульовий індекс масиву значень якого буде дорівнювати telegram.id хоста. При виході учасника кімнати з гри його id видаляється з масиву методом socket.on(“disconnect”). Якщо масив не містить жодного значення кімната видаляється. Після створення кімнати бот надсилає кнопку з посиланням, в якому вже прописано ключ для входу до потрібної кімнати. Це дозволяє позбавити гравців потреби вигадувати та прописувати пароль самостійно.

Серверний код так само потрібно завантажити на хостінг. Vercel надає можливість завантажити Node.js код, але команду запуску потрібно прописати самостійно. Для цього в файлі package.json у полі scripts потрібно додати команду “build”: “node server.js”.

У обох частинах додатку усі URL потрібно змінити з localhost на домен, виданий хостингами. Після цих дій бот стане доступним усім та його можна буде знайти в Telegram за запитом @MediumBassBot. Команда /start створює кімнату для ігрової сесії та надсилає кнопку, при натисканні на яку користувач приєднується до створеної кімнати.(рис 3.17)

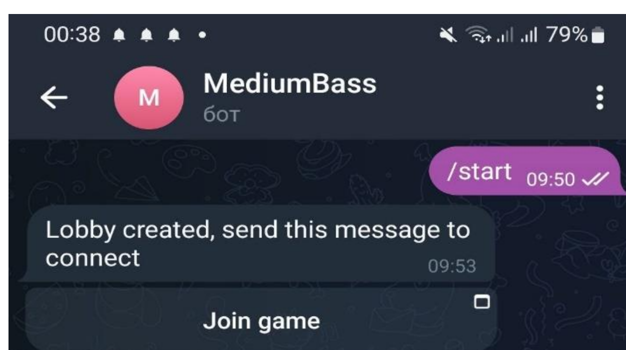

Рисунок 3.17 – приклад діалогу з ботом гри

Після пересилання повідомлення двом тестувальникам була проведена одна повна ігрова сесія (рис. 3.18).

Рисунок 3.18 – Тестування роботи застосунку з 3 різних пристроїв

За результатами тестування можна зробити висновки, що заплановані функціональні можливості проекту працюють коректно. Telegram бот належним чином оброблює команду створення нової кімнати та надає дійсне посилання на гру. Гра успішно використовує механізм авторизації через Telegram, дані користувачів передаються і синхронізуються між усіма учасниками ігрової сесії належним чином, гра правильно обробляє всі дії користувачів, включаючи натискання кнопок, вибір елементів меню та інші взаємодії. Інтерфейс гри правильно масштабуються та залишається зручним для користувачів на всіх підтримуваних пристроях.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи бакалавра був створений Telegram бот «@MediumBassBot» у який впроваджено веб-додаток багатокористувальницької 2D гри у жанрі Roguelike з використанням технологій NodeJS, ReactJS, WebSocket та TelegramWebApp.

Основні етапи розробки описані в цій роботі: проектування, налаштування проекту в середовищі розробки WebStorm, створення API, налаштування клієнт-серверного з’єднання, програмування ігрової логіки, створення інтерфейсу та розгортання проекту на хостінгах, з подальшим впровадженням до Telegram.

У першому розділ проведено аналіз історії розвитку браузерних ігор, та варіантів їх впровадження до сторонніх веб-сайтів.

У другому розділі описані функціональні та нефункціональні вимоги, проведено порівняльний аналіз можливих архітектурних підходів описано вибір мови програмування, середовищ розробки та допоміжних бібліотек.

У третьому розділі описані етапи розробки веб-додатку, створення та прив'язка Telegram бота до нього, процес розміщення проекту на хостінгах.

Створена гра є унікальною у своєму жанрі, завдяки деталізованій бойовій системі, яка використовує велику множину параметрів для розрахунку результату кожного ходу. Впровадження гри до Telegram бота надає зручні можливості для початку гри самостійно, або з запрошеними друзями.

Результати розробки повністю відповідають поставленим функціональним та нефункціональним вимогам.

Гра має перспективу для розвитку і розширення свого функціоналу, завдяки гнучкій архітектурі та прописаному інтерфейсу програмування. Першочергово в плани розвитку гри входить редизайн користувальницького інтерфейсу та використаних ігрових ассетів. Для більшої варіативності ігрового процесу усі таблиці БД будуть доповнюватися, для додавання нових персонажів, ворогів, здібностей.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. GAMEE Telegram | GAMEE Wiki. Welcome | GAMEE Wiki. URL:https://wiki.gamee.com/gameplay/telegramaccounts (дата звернення 29.04.2024)
2. Офіційна документація HTML: HyperText Markup Language. URL: https://developer.mozilla.org/en-US/docs/Web/HTML (дата звернення 30.04.2024)
3. Офіційна документація CSS: Cascading Style Sheets | MDN. MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Web/CSS (дата звернення 4.05.2024)
4. Уроки React | Codecademy. Codecademy. URL: https://www.codecademy.com/learn/react-101 (дата звернення 4.05.2024)
5. Офіційна документація React. URL: https://reactjs.org/docs/getting-started.html (дата звернення 10.05.2024)
6. Офіційна документація Redux Toolkit. URL: https://redux-toolkit.js.org/ (дата звернення 15.05.2024)
7. Офіційна документація Redux | Redux. Redux – A JS library for predictable and maintainable global state management | Redux. URL: https://redux.js.org/introduction/getting-started (дата звернення 15.05.2024)
8. Сумець О. Проєктування операційних систем. KROK University, 2021. URL: https://doi.org/10.31732/pros (дата звернення 18.05.2024)
9. Офіційна документація Node.js. Node.js – Run JavaScript Everywhere. URL: https://nodejs.org/docs/latest/api/ (дата звернення 20.05.2024)
10. Nystrom R. Game Programming Patterns. Genever Benning, 2014. 354 с.
11. Офіційна документація MongoDB. MongoDB Documentation. URL: https://docs.mongodb.com/ (дата звернення 22.05.2024)
12. Vickler A. Javascript: Javascript Back End Programming. Independently Published, 2021. 216 с. (дата звернення 25.05.2024)

13. Офіційна документація Express. Express – Node.js web application framework. URL: https://expressjs.com/ (дата звернення 30.05.2024)
14. Timms S., Antani V., Mantyla D. JavaScript: Functional Programming for JavaScript Developers. Packt Publishing, 2016. 646 с.
15. Основи розробки ігор. URL: https://www.gamedev.net/resources/ (дата звернення 2.06.2024)
16. Telegram mini apps. Telegram APIs. URL: https://core.telegram.org/bots/webapps (дата звернення 8.06.2024)

ДОДАТОК А

Вихідний код програми

```
const TelegramBot = require('node-telegram-bot-api');

const WebAppUrl ="https://tg-web-game-react-socket.vercel.app"
const baseUrl="http://localhost:3000"

const WebAppUrl ="https://tg-web-game-react-socket.vercel.app/"
const baseUrl="https://tg-web-game-react-socket.vercel.app"

const token = '6037873883:AAE7uHSgYV7Y3yL1T6IPdYr6O31Eqe8eu1I'

// Create a bot that uses 'polling' to fetch new updates
const bot = new TelegramBot(token, {polling: true});
let userName =" "
// Listen for any kind of message. There are different kinds of
// messages.
bot.on('message', (msg) => {
    const chatId = msg.chat.id;
    const text =msg.text
    if(text === "/start"){
        bot.sendMessage(chatId, 'Lobby created, send this message to
connect',{
            reply_markup:{
                inline_keyboard: [
                    [{text: "Join game", web_app: {url: WebAppUrl}}]
                ]
            }
        });
    }
    userName=msg.chat.first_name

});


const cors =require('cors')

const express = require('express');

const app = express();
const server = require('http').Server(app);
const io = require('socket.io')(server,{
    //https://vocal-clafoutis-0f197b.netlify.app
'http://localhost:3000'
    cors: {
        origin: "https://tg-web-game-react-socket.vercel.app",
        methods: ["GET", "POST"]
    }
});
```

```
const {weaponSkills} = require('./db');
const {monsters} = require('./db');
const {weapons} = require('./db');




let team=[]
let teamHP=[]

let monster
let MonsterCurrentHp
let EndTurn =[]
let activePlayers=[]
let monsterList=[]
let i=0
let monsterCounter=0
let Messages = []
let isStunned = false
let isBleeding = false
let bleedCounter = 0
let bleedDamage = 0
let deadPlayersCounter = 0
//https://vocal-clafoutis-0f197b.netlify.app 'http://localhost:3000'
app.use(cors({origin: "https://tg-web-game-react-
socket.vercel.app"}));
app.use(express.json())
let PORT =process.env.PORT || 8080
function onDisconnect(id){
    activePlayers.splice(id,1)
}
function onRestart(){
    team=[]
    teamHP=[]
    EndTurn =[]
    activePlayers=[]
    monsterList=[]
    i=0
    monsterCounter=0
    deadPlayersCounter = 0
    Messages = []
}
io.on('connection', (socket) =>{

    socket.on("RESTART", (data) =>{
        onRestart()
        io.emit("RESTART")
    })

   socket.on("WEAPON SUBMITTED", (data) =>{
       team.push({...weapons[data.IdOfWeapon], id:i,
socketId:socket.id,userName:userName})
       activePlayers.push(i)
       i++
```

```
        teamHP.push(team[team.length-1].hp)

        io.emit("WEAPON SUBMITTED", {
         team, teamHP
        })

    })
     socket.on("PLAYER DAMAGED", (data) =>{
         teamHP[data.playerId]=data.currentHp
         io.emit("PLAYER DAMAGED", {
             teamHP
         })

     })
     socket.on("PLAYER HEAL", (data) =>{
         teamHP[data.playerId]+=data.heal
         if( teamHP[data.playerId]> team[data.playerId].maxhp) {
             teamHP[data.playerId]=team[data.playerId].maxhp
         }
         io.emit("PLAYER DAMAGED", {
             teamHP
         })
     })
     socket.on("EVERYONE STATUP SKILL", (data) =>{
         let everyoneid=data.everyoneID
         io.emit("EVERYONE STATUP", {
             everyoneid:everyoneid
         })

     })

     socket.on("MONSTER STATE", (data) =>{
         isStunned=data.isStunned
         isBleeding=data.isBleeding
         bleedCounter=data.bleedCounter
         bleedDamage=data.bleedDmg
         io.emit("MONSTER STATE", {
             isStunned:isStunned,
             isBleeding:isBleeding
         })

     })
     socket.on("SKILL USED", (data) =>{
         if(data.currentHp===9999){
             for(let i=0;i<12;i++){

monsterList.push(monsters[Math.floor(Math.random()*monsters.length)]
)

             }
             monsterList.sort((x, y) => x.lvl - y.lvl);

             monster=monsterList[0]
             MonsterCurrentHp=monster.maxhp
             }
```

```
        if(data.currentHp!=9999&&MonsterCurrentHp>=0) {
            MonsterCurrentHp = data.currentHp
            EndTurn.push(data.isActive)

            if(EndTurn.length===activePlayers.length){
                EndTurn=[]
                let damagedPlayer =
activePlayers[Math.floor(Math.random()*activePlayers.length)]


                io.emit("MONSTER ATTACKS", {
                    hitRandomizer1: Math.random(),
                    hitRandomizer2: Math.random(),
                    damagedPlayer: damagedPlayer,
                    isStunned: isStunned
                })
            if(isStunned){
                Messages.unshift(monster.name+" recovers from stun")
                isStunned=false
            }
                if(isBleeding){
                    MonsterCurrentHp = MonsterCurrentHp-bleedDamage
                    io.emit("SKILL USED", {
                        MonsterCurrentHp: MonsterCurrentHp
                    })
                    Messages.unshift(monster.name+" takes
"+bleedDamage+" DMG from bleed")
                    bleedCounter--
                    if(bleedCounter<=0){
                        isBleeding=false
                    }
                }

            }

        }

        if(MonsterCurrentHp<=0){
            monsterCounter++
            console.log(monsterCounter)
            if (monsterCounter>=10){
                io.emit("GAME END", {
                        isVictory: true
                })

            }
            monster= monsterList[monsterCounter]
            MonsterCurrentHp=monster.maxhp
            bleedCounter = 0
            bleedDamage = 0
            isBleeding=false
            isStunned=false
                io.emit("MONSTER DEAD")

        }
        io.emit("SKILL USED", {
```

```
            finalDmg: data.finalDmg,
            Message:data.Message,
            maxhp:monster.maxhp,
            hp:MonsterCurrentHp,
            name:monster.name,
            img:monster.img,
            spd:monster.spd,
            arm:monster.arm,
            crt:monster.crt,
            dmg:monster.dmg
        })


    })
    socket.on("CHANGE MESSAGES", (data) =>{
        if(Messages.length>3){
            Messages.splice(3)
        }
        Messages.unshift(data.Message)

        io.emit("CHANGE MESSAGES", {
            Messages
        })
    })
    socket.on("PLAYER DEAD", (data) =>{
            onDisconnect(data.playerId)
            deadPlayersCounter++
                if(deadPlayersCounter>=teamHP.length){
                    io.emit("GAME END", {
                        isVictory: false
                    })
                    onRestart()
                }
    })
    console.log('user connected', socket.id)

})

server.listen(PORT, () => {
    console.log('listening on '+PORT);
});
app.get('/team', (req,res) => {
    res.json({team})
})
app.get('/skilllist', (req,res) =>{
    res.json({weaponSkills})
})
app.get('/monsterlist', (req,res) =>{
    res.json({monsterList})
})
app.get('/weapons', (req,res) =>{
    res.json({weapons})
})
import io from 'socket.io-client'
export const baseURL="http://localhost:8080"
```

```
export const socket= io(baseURL);

import './App.css';
import React, {useEffect, useState} from 'react';
import {socket, baseURL} from "./socket";
import Players from "./components/Players";
import Monster from "./components/Monster";
import SkillList from "./components/SkillList";
import ChatLog from "./components/ChatLog";
import ModalWeaponChoose from "./components/ModalWeaponChoose";

const tele = window.Telegram.WebApp

let ThisClassSkills=[]

function App() {
    useEffect(() => {
        tele.ready()
    }, [])


    //getSkills
    // GetSkills
    const [error, setError] = useState(null);
    const [isLoaded, setIsLoaded] = useState(false);
    const [items, setItems] = useState([]);
    const [modalDeath, setModalDeath] = useState(false);

    const addUniSkills = (ID) => {
        let allSkills = []
        allSkills.push(...ThisClassSkills[0])
        allSkills.push(...ThisClassSkills[1])
        allSkills.push(...ThisClassSkills[ID + 2])
        setItems(allSkills)

    }
    const checkIfPlayerDied = (playerState) =>{
        setModalDeath(playerState)
    }
    //calculations
    function CalculateDamage(AttackStat, AttackCrt, AttackName,
Victim, Randomizer1, Randomizer2) {
        let damage = AttackStat
        let message

        if (damage <= Victim.arm) {
            damage = 0
            message = Victim.name + " armor wasn`t penetrated."
        } else {
            damage = damage - Victim.arm
            message = AttackName + " deal " + damage + " damage to "
+ Victim.name
        }
        if (Randomizer1 <= AttackCrt) {
            damage = damage * 2
```

```
            message = "CRITICAL STRIKE! on " + damage + " by " +
AttackName
        }
        if (Randomizer2 <= Victim.spd / 20) {
            damage = 0
            message = Victim.name + " evaids " + AttackName + "
attack. "
        }

        return [damage, message]
    };
    // ModalWeaponChoose

    const [modalActive, setModalActive] = useState(true)
    const [playerStats, setPlayerStats] = useState([])
    const [statsChanged, setStatsChanged] = useState(0)
    const [weaponID, setWeaponID] = useState([])
    const [trinket, setTrinket] = useState([])
    const [thisPlayerIndex, setThisPlayerIndex] = useState([])

    const savePlayerStats = (stats) => {
        setPlayerStats(stats)
        setStatsChanged(statsChanged + 1)

    }
    //Monster
    const [monster, setMonster] = useState([])
    const [currentHp, setCurrentHp] = useState([])
    const calculateCurrentHp = (hp) => {
        setCurrentHp(hp)
    };
    // ModalGameEnd


    useEffect(() => {
        socket.on("SKILL USED", (data) => {
            setMonster(data)
            setCurrentHp(data.hp)

        })
        socket.on("RESTART", (data) => {
            window.location.reload()
        })
    }, [])
    useEffect(() => {
        fetch(baseURL+"/skilllist")
            .then(res => res.json())
            .then(
                (result) => {
                    setIsLoaded(true);
                    setItems(result.weaponSkills[0]);
                    ThisClassSkills = result.weaponSkills;
                },
                (error) => {
                    setIsLoaded(true);
                    setError(error);
```

```
                }
            )
    }, [])
    if (error) {
        return <div>Error: {error.message}</div>;
    } else if (!isLoaded) {
        return <div>Loading...</div>;
    } else {
        return (
            <div>
                <div className="App-header">
                    <header className="App-main">

                        <div className="vertic">
                            <div className="horizont">
                                <div className="vertic">
                                    <Players
playerStats={playerStats} monster={monster}

CalculateDamage={CalculateDamage}
setThisPlayerIndex={setThisPlayerIndex}

checkIfPlayerDied={checkIfPlayerDied} />
                                    <ChatLog/>

                                </div>
                                <Monster className="monster"
name={monster.name} maxhp={monster.maxhp} img={monster.img}
                                         currentHP={currentHp}/>
                            </div>

                            <SkillList playerStats={playerStats}
monster={monster} currentHp={currentHp} items={items}

CalculateDamage={CalculateDamage}
calculateCurrentHp={calculateCurrentHp}

savePlayerStats={savePlayerStats} statsChanged={statsChanged}
                                       weaponSubmitted={weaponID}
thisPlayerIndex={thisPlayerIndex}
                                       trinket={trinket}
setTrinket={setTrinket}
                                       modalDeath={modalDeath}
setModalDeath={setModalDeath}

setModalWeapon={setModalActive} setItems={setItems} />
                        </div>

                    </header>

                </div>
                <ModalWeaponChoose active={modalActive}
setActive={setModalActive}
                                   savePlayerStats={savePlayerStats}
addUniSkills={addUniSkills} />
```

```
            </div>
        );
    }
}

export default App;

import React, {createContext} from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

export const Context = createContext(null)


const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <App/>
  </React.StrictMode>
);

// If you want to start measuring performance in your app, pass a
function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-
vitals
reportWebVitals();

import React from 'react';
import "./HPBar.css"
import "../css/NewSkill.css"
const Weapon = (props) => {

    return (
        <div className={"newSkill"} onClick={() =>
props.chooseWeapon(props.id)}  >
            <img src={props.img} width={64} height={32}
alt={"weapon"}></img>
            <p><b>{props.name}</b></p>
            <p>{props.desc}</p>
        </div>
    );
};

export default Weapon;
import React, {useEffect, useState} from 'react';
import './SkillList.css';
import SkillDescription from "./SkillDescription";
import ObtainedSkills from "./ObtainedSkills";
import {socket} from "../socket";
import '../index.css';
import ModalNewSkill from "./ModalNewSkill";
import ModalGameEnd from "./ModalGameEnd";
```

```
let SkillDescriptionManaCost
let SkillDescriptionDmg
let CurrentItem
let FirstSkill = 0
let SecondSkill = 0
let ThirdSkill = 0
let b= "START"
const SkillList = ({playerStats,
monster,currentHp,calculateCurrentHp, CalculateDamage,
savePlayerStats, statsChanged, items, setTrinket,
                       trinket, thisPlayerIndex, modalDeath,
setModalDeath, setModalWeapon, setItems}) => {

    const [obtainedSkills, setObtainedSkills] = useState([{
        "id": 0,
        "classId": 0,
        "name": "Slash",
        "dmg": "0",
        "cd": 1,
        "price": 0,
        "needTarget": 0,
        "mlt": 1,
        "manaCost": 0,
        "attackType": "normal",
        "desc": "Basic hit, nothing special"
    },
        {
            "id": 1,
            "classId": 0,
            "name": "Skip Turn",
            "dmg": "0",
            "cd": 0,
            "price": 0,
            "needTarget": 0,
            "mlt": 1,
            "manaCost": 0,
            "attackType": "skip",
            "desc": "Skip turn"
        },]);
    let CurrentObtainedSkills = obtainedSkills

    // Monster
    const [playerIsDead, setPlayerIsDead] = useState(false);
    const [gameStarted, setGameStarted] = useState(false);
    const Start = () => {
        if(gameStarted===false) {
            socket.emit("SKILL USED", {currentHp: 9999})
            setGameStarted(true)
            b="RESTART"
        }else{
            socket.emit("RESTART")

        }
    }

    // DamageCalculations
```

```
    const [message, setMessage] = useState("");


    const chooseMessage = (message, message2, item) => {
        setMessage(message);
        SkillDescriptionDmg = message2
        CurrentItem = item
        SkillDescriptionManaCost = item.manaCost
    };


    // EndOfTheRound

    const [modalNewSkillActive, setModalNewSkillActive] =
useState(false)


    const savePlayerSkills = (skills) => {
        CurrentObtainedSkills.push(items[skills])
        setObtainedSkills(CurrentObtainedSkills)
    }
    const endOfTheRound = (value) => {
        FirstSkill = Math.floor(Math.random() * items.length)
        SecondSkill = Math.floor(Math.random() * items.length)
        ThirdSkill = Math.floor(Math.random() * items.length)


    }
    useEffect(() => {
        socket.on("MONSTER DEAD", (data) => {
            endOfTheRound(1)
            setModalNewSkillActive(true)

        })

    }, [endOfTheRound])
    // GetSkills





        return (
            <div className="footer">
                <button onClick={Start}>{b}</button>


                <SkillDescription desc={message}
dmg={SkillDescriptionDmg} manaCost={SkillDescriptionManaCost}
currentHp={currentHp} monster={monster}
                                  endOfTheRound={endOfTheRound}
calculateCurrentHp={calculateCurrentHp} statsChanged={statsChanged}
                                  playerStats={playerStats}
item={CurrentItem} CalculateDamage={CalculateDamage}
trinket={trinket}
```

```
                                  modalDeath={modalDeath}
setModalDeath={setModalDeath} playerIsDead={playerIsDead}
setPlayerIsDead={setPlayerIsDead}
                                  gameStarted={gameStarted}
                                   />
                <div className={"wrapper"}>
                    {obtainedSkills.map(item => (
                        <ObtainedSkills  key={item.name} item={item}

chooseMessage={chooseMessage} />
                    ))}
                </div>
                <ModalNewSkill  active={modalNewSkillActive}
setActive={setModalNewSkillActive} items={items}
                                savePlayerSkills={savePlayerSkills}
savePlayerStats={savePlayerStats} playerStats={playerStats}
                                FirstSkill={FirstSkill}
SecondSkill={SecondSkill} ThirdSkill={ThirdSkill}
                                setTrinket={setTrinket}
thisPlayerIndex={thisPlayerIndex} modalDeath={modalDeath}/>
                <ModalGameEnd  />
            </div>
        );

}
export default SkillList;
import React, {useEffect, useState} from 'react';
import {socket} from "../socket";
import "../css/SkillDescription.css"
import ModalDeath from "./ModalDeath";

let BaseDmg
let finalDmg
let Message=""
let WeaponName
let CalculatedDamage
let Randomizer1
let Randomizer2
let CurrentStats
let MonsterStats
let bleedCounter=0
let bleedDmg=0
let AllTimeDamage=0
const SkillDescription = (props) => {

    const [isStunned, setIsStunned] = useState(false);
    const [isBleeding, setIsBleeding] = useState(false);
    const [isActive, setIsActive] = useState(true);
    const [turn, setTurn] = useState(1);
    const [playersMana, setPlayersMana] = useState(200);

    const [modalDead, setModalDead] = useState(false);
    // set player stats
    useEffect(()=>{
            CurrentStats = [
            props.playerStats.dmg,//0 DMG
```

```
            props.playerStats.arm,//1 ARM
            props.playerStats.spd,//2 SPD
            props.playerStats.crt,//3 CRT
            props.playerStats.hp, //4 HP
            props.playerStats.manaPool  //5 mana
        ]

    },[props.statsChanged])
    // set monster stats
    useEffect(()=>{
        MonsterStats = [
            props.monster.dmg,  //0 DMG
            props.monster.arm,  //1 ARM
            props.monster.spd,  //2 SPD
            props.monster.crt,  //3 CRT
            props.monster.maxhp,//4 HP
        ]

    //bleed
    },[ props.monster])
    useEffect(()=>{
        socket.on("MONSTER STATE", (data) =>{
            setIsStunned(data.isStunned)
            setIsBleeding(data.isBleeding)
        })


    },[])
const becomeSpectator = () =>{
        setModalDead(false)
        props.setPlayerIsDead(true)
}
   const DealSkill = () =>{
       setIsActive(false)

       Randomizer1=Math.random()
       Randomizer2=Math.random()
       WeaponName=props.playerStats.name

       switch (props.item.attackType) {
           case "normal":
               BaseDmg = CurrentStats[props.item.dmg]*props.item.mlt
               if(props.trinket.name==="Grindstone"){
                   BaseDmg+=1
               }   //trinket
               break;
           case "compare":


if(CurrentStats[props.item.compLeft]>MonsterStats[props.item.compRig
ht]){
               BaseDmg = CurrentStats[props.item.dmg]*props.item.mlt
            }
               else
                   BaseDmg =CurrentStats[props.item.dmg]
               break;
```

```
            case "stun":
                BaseDmg = CurrentStats[props.item.dmg]*props.item.mlt
                console.log(isStunned)
                Message += props.monster.name+" is stunned "
                socket.emit("MONSTER STATE", {
                    isStunned:true,

                })
                break;
            case "bleed":
                BaseDmg = CurrentStats[props.item.dmg]*props.item.mlt
                CalculatedDamage =
props.CalculateDamage(BaseDmg,CurrentStats[3],WeaponName,props.monst
er,Randomizer1,Randomizer2)
                bleedCounter+=props.item.bleedDuration
                bleedDmg=props.item.bleedDmg
                if(props.trinket.name!=="Weakening Poison"){
                    bleedDmg=props.item.bleedDmg+2
                }
                Message += props.monster.name+" is bleeding, "

if(CalculatedDamage[0]==0&&props.trinket.name!=="Internal
Bleeding"){    //trinket
                    bleedCounter+=0
                    bleedDmg+=0
                    Message += "Bleed failed, "
                }

                socket.emit("MONSTER STATE", {
                    isBleeding:true,
                    bleedCounter:bleedCounter,
                    bleedDmg:bleedDmg
                })
                break;
            case "pierce":
                BaseDmg =
(CurrentStats[props.item.dmg]+props.monster.arm)*props.item.mlt
                break;
            case "afterStun":
                if(isStunned) {
                    BaseDmg =
CurrentStats[props.item.dmg]*props.item.mlt
                }
                break;
            case "afterBleed":
                if(isBleeding) {
                    BaseDmg =
CurrentStats[props.item.dmg]*props.item.mlt
                }
                break;
            case "skip":
                BaseDmg = 0
                break;
        }
```

```
        CalculatedDamage =
props.CalculateDamage(BaseDmg,CurrentStats[3],WeaponName,props.monst
er,Randomizer1,Randomizer2)
        finalDmg = CalculatedDamage[0]
        Message += CalculatedDamage[1]
        AllTimeDamage+=finalDmg
        setPlayersMana(prevState => prevState - props.item.manaCost)
        let currentHp
            currentHp = props.currentHp-finalDmg
            props.calculateCurrentHp(currentHp)
        socket.emit("SKILL USED",{finalDmg: finalDmg,
Message:Message, currentHp:currentHp, isActive:WeaponName})
        socket.emit("CHANGE MESSAGES", {Message:Message})
        Message=""
    }

    useEffect(()=>{
    socket.on("MONSTER ATTACKS", (data) =>{
        setIsActive(true)
            let turnChange=turn+1
                    setTurn(turnChange)
            let mp=playersMana+25
        if(mp<225){
        setPlayersMana(mp)}
    })
    },[playersMana])

    useEffect(()=>{
        props.setPlayerIsDead(props.modalDeath)
        setModalDead(props.modalDeath)
    },[props.modalDeath])

    return (
        <div className="skillDescription">
            <p>{props.desc}      [{props.manaCost}]</p>
            <div className={"horizont"}>
                <p>
MANA    :{playersMana}/{props.playerStats.manaPool}</p>
                <button onClick={DealSkill}
disabled={!isActive||playersMana<props.manaCost||props.playerIsDead|
|props.manaCost===undefined||props.monster.arm===undefined}
className={"button-89"}>Use Skill</button>
            </div>

            <ModalDeath active={modalDead}
becomeSpectator={becomeSpectator} AllTimeDamage={AllTimeDamage} />
        </div>
    );
};

export default SkillDescription;

import React, {useEffect, useState} from 'react';
import {socket} from "../socket";
import "../css/SkillDescription.css"
import ModalDeath from "./ModalDeath";
```

```
let BaseDmg
let finalDmg
let Message=""
let WeaponName
let CalculatedDamage
let Randomizer1
let Randomizer2
let CurrentStats
let MonsterStats
let bleedCounter=0
let bleedDmg=0
let AllTimeDamage=0
const SkillDescription = (props) => {

    const [isStunned, setIsStunned] = useState(false);
    const [isBleeding, setIsBleeding] = useState(false);
    const [isActive, setIsActive] = useState(true);
    const [turn, setTurn] = useState(1);
    const [playersMana, setPlayersMana] = useState(200);

    const [modalDead, setModalDead] = useState(false);
    // set player stats
    useEffect(()=>{
            CurrentStats = [
            props.playerStats.dmg,//0 DMG
            props.playerStats.arm,//1 ARM
            props.playerStats.spd,//2 SPD
            props.playerStats.crt,//3 CRT
            props.playerStats.hp, //4 HP
            props.playerStats.manaPool  //5 mana
        ]

    },[props.statsChanged])
    // set monster stats
    useEffect(()=>{
        MonsterStats = [
            props.monster.dmg,  //0 DMG
            props.monster.arm,  //1 ARM
            props.monster.spd,  //2 SPD
            props.monster.crt,  //3 CRT
            props.monster.maxhp,//4 HP
        ]

    //bleed
    },[ props.monster])
    useEffect(()=>{
        socket.on("MONSTER STATE", (data) =>{
            setIsStunned(data.isStunned)
            setIsBleeding(data.isBleeding)
        })


    },[])
const becomeSpectator = () =>{
        setModalDead(false)
```

```
        props.setPlayerIsDead(true)
}
   const DealSkill = () =>{
       setIsActive(false)

       Randomizer1=Math.random()
       Randomizer2=Math.random()
       WeaponName=props.playerStats.name

       switch (props.item.attackType) {
           case "normal":
               BaseDmg = CurrentStats[props.item.dmg]*props.item.mlt
               if(props.trinket.name==="Grindstone"){
                   BaseDmg+=1
               }   //trinket
               break;
           case "compare":


if(CurrentStats[props.item.compLeft]>MonsterStats[props.item.compRig
ht]){
               BaseDmg = CurrentStats[props.item.dmg]*props.item.mlt
            }
               else
                   BaseDmg =CurrentStats[props.item.dmg]
               break;
           case "stun":
               BaseDmg = CurrentStats[props.item.dmg]*props.item.mlt
               console.log(isStunned)
               Message += props.monster.name+" is stunned "
               socket.emit("MONSTER STATE", {
                   isStunned:true,

               })
               break;
           case "bleed":
               BaseDmg = CurrentStats[props.item.dmg]*props.item.mlt
               CalculatedDamage =
props.CalculateDamage(BaseDmg,CurrentStats[3],WeaponName,props.monst
er,Randomizer1,Randomizer2)
               bleedCounter+=props.item.bleedDuration
               bleedDmg=props.item.bleedDmg
               if(props.trinket.name!=="Weakening Poison"){
                   bleedDmg=props.item.bleedDmg+2
               }
               Message += props.monster.name+" is bleeding, "

if(CalculatedDamage[0]==0&&props.trinket.name!=="Internal
Bleeding"){    //trinket
                   bleedCounter+=0
                   bleedDmg+=0
                   Message += "Bleed failed, "
               }

               socket.emit("MONSTER STATE", {
                   isBleeding:true,
```

```
                    bleedCounter:bleedCounter,
                    bleedDmg:bleedDmg
                })
                break;
            case "pierce":
                BaseDmg =
(CurrentStats[props.item.dmg]+props.monster.arm)*props.item.mlt
                break;
            case "afterStun":
                if(isStunned) {
                    BaseDmg =
CurrentStats[props.item.dmg]*props.item.mlt
                }
                break;
            case "afterBleed":
                if(isBleeding) {
                    BaseDmg =
CurrentStats[props.item.dmg]*props.item.mlt
                }
                break;
            case "skip":
                BaseDmg = 0
                break;
        }
        CalculatedDamage =
props.CalculateDamage(BaseDmg,CurrentStats[3],WeaponName,props.monst
er,Randomizer1,Randomizer2)
        finalDmg = CalculatedDamage[0]
        Message += CalculatedDamage[1]
        AllTimeDamage+=finalDmg
        setPlayersMana(prevState => prevState - props.item.manaCost)
        let currentHp
            currentHp = props.currentHp-finalDmg
            props.calculateCurrentHp(currentHp)
        socket.emit("SKILL USED",{finalDmg: finalDmg,
Message:Message, currentHp:currentHp, isActive:WeaponName})
        socket.emit("CHANGE MESSAGES", {Message:Message})
        Message=""
    }

     useEffect(()=>{
     socket.on("MONSTER ATTACKS", (data) =>{
         setIsActive(true)
             let turnChange=turn+1
                    setTurn(turnChange)
             let mp=playersMana+25
         if(mp<225){
         setPlayersMana(mp)}
     })
     },[playersMana])

     useEffect(()=>{
         props.setPlayerIsDead(props.modalDeath)
         setModalDead(props.modalDeath)
     },[props.modalDeath])
```

```
    return (
        <div className="skillDescription">
            <p>{props.desc}      [{props.manaCost}]</p>
            <div className={"horizont"}>
                <p>
MANA    :{playersMana}/{props.playerStats.manaPool}</p>
                <button onClick={DealSkill}
disabled={!isActive||playersMana<props.manaCost||props.playerIsDead|
|props.manaCost===undefined||props.monster.arm===undefined}
className={"button-89"}>Use Skill</button>
            </div>

            <ModalDeath active={modalDead}
becomeSpectator={becomeSpectator} AllTimeDamage={AllTimeDamage} />
        </div>
    );
};

export default SkillDescription;

import React  from 'react';
import HPBar from "./HPBar";
import Avatar from "./Avatar";



const Player = ({item, playersHP}) => {







    return (
        <div>
            <Avatar/>
            <p>{item.name} {playersHP}\{item.hp}</p>
            <HPBar hp={playersHP} maxHp={item.hp} />
        </div>
    );
};

export default Player;

import React  from 'react';
import "../css/ObtainedSkills.css"



const ObtainedSkills = (props) => {

    let description = props.item.desc;
    let finalDmg
    let item

    const CalculatedDamage = () =>{
```

```
        finalDmg = props.item.dmg
        item=props.item

        props.chooseMessage(description,finalDmg,item)

    }
    return (


            <div onClick={CalculatedDamage} className="obtained">
                <p> {props.item.name}</p>

            </div>


    )
};



export default ObtainedSkills;

import React from 'react';
import "../css/NewSkill.css"
let color
const NewSkill = ({skill, chooseSkill,colors}) => {
color = colors[skill.price]
    return (
        <div className={"newSkill"} onClick={()=>
chooseSkill(skill.id)} style={{fontSize:16}}>
            <b style={{color:color}}>{skill.name}</b>
            <p>{skill.desc}</p>
        </div>
    );
};

export default NewSkill;

import React, {useEffect, useState} from 'react';
import Weapon from "./Weapon";
import {baseURL, socket} from "../socket";
import "../css/ModalWeaponChoose.css"

const ModalWeaponChoose = ({active, setActive, savePlayerStats,
addUniSkills}) => {
let currentStats


    const [weaponId, setWeaponId] = React.useState(0);




    const chooseWeapon = (weaponId) =>{
        setWeaponId(weaponId)
    }
```

```
    const   postWeapon = (JoinId, JoinMessage) => {
        currentStats=items[weaponId]
        setActive(false)
        socket.emit("WEAPON SUBMITTED",{IdOfWeapon : JoinId,
joinMessage: JoinMessage})

        savePlayerStats(currentStats)
        addUniSkills(weaponId)

    }

    const [error, setError] = useState(null);
    const [isLoaded, setIsLoaded] = useState(false);
    const [items, setItems] = useState([]);

    useEffect(() => {
        fetch(baseURL+"/weapons")
            .then(res => res.json())
            .then(
                (result) => {
                    setIsLoaded(true);
                    setItems(result.weapons);
                },
                (error) => {
                    setIsLoaded(true);
                    setError(error);
                }
            )
    }, [])
    if (error) {
        return <div>Error: {error.message}</div>;
    } else if (!isLoaded) {
        return <div>Loading...</div>;
    } else {
        return (
            <div className={active ? "modal active" : "modal"}>
                <div className="modal__content">
                    <div>
                        {items.map(item => (
                            <Weapon key={item.name} name={item.name}
desc={item.desc} img={item.img} id={item.id}
                                    chooseWeapon={chooseWeapon}/>
                        ))}
                    </div>

                    <button className="button89"
                        onClick={() =>
postWeapon(items[weaponId].id, "joined lobby as " +
items[weaponId].name)}> Choose {items[weaponId].name}</button>

                </div>
            </div>
        );
    }
};
```

```
export default ModalWeaponChoose;

import React, {useEffect} from 'react';
import "../css/ModalWeaponChoose.css"
import NewSkill from "./NewSkill";
import {socket} from "../socket";
let colors =[" ","white",
"LightBlue","RebeccaPurple","Gold","yellow","green"]
let currentStats

const ModalNewSkill = ({active, setActive, savePlayerSkills,
items ,FirstSkill, SecondSkill, ThirdSkill,
                           playerStats, savePlayerStats, setTrinket,
thisPlayerIndex, modalDeath}) => {


    useEffect(() => {
         currentStats=playerStats
    }, [playerStats])



    const [skillId, setSkillId] = React.useState(0);
    const [skillName, setSkillName] = React.useState("");
    const chooseSkill = (skillId) =>{
        setSkillId(skillId)
        setSkillName(items[skillId].name)
    }
    useEffect(() => {

    }, [])

    const   postSkill = (skillId) => {

       switch (items[skillId].attackType) {
           case "statup":
               switch (items[skillId].stat) {
                   case 0:
                       currentStats.dmg+=items[skillId].mlt

console.log( currentStats.dmg,items[skillId].mlt)
                       break;
                   case 1:
                       currentStats.arm+=items[skillId].mlt

console.log( currentStats.arm,items[skillId].mlt)
                       break;
                   case 2:
                       currentStats.spd+=items[skillId].mlt

console.log( currentStats.spd,items[skillId].mlt)
                       break;
                   case 3:
                       currentStats.crt+=items[skillId].mlt

console.log( currentStats.crt,items[skillId].mlt)
```

```
                        break;

                }
                savePlayerStats(currentStats)
                console.log("save otrabotal")
                break;
            case "everyoneStatup":
                socket.emit("EVERYONE STATUP SKILL",
{everyoneID:items[skillId].statupSkillId})
                break;
            case "trinket":
                setTrinket(items[skillId])
                break;
            case "heal":
                socket.emit("PLAYER HEAL", {playerId:thisPlayerIndex,

heal:items[skillId].heal})
                console.log("heal otpravil",thisPlayerIndex,
items[skillId].heal)
                break;
            default:
                savePlayerSkills(skillId)
                break;
        }
        setActive(false)
    }

        return (
            <div className={active&&!modalDeath ? "modal active" :
"modal"} >
                <div className="modal__content">
                    <NewSkill skill={items[FirstSkill]   }
colors={colors} chooseSkill={chooseSkill}/>
                    <NewSkill skill={items[SecondSkill]  }
colors={colors} chooseSkill={chooseSkill}/>
                    <NewSkill skill={items[ThirdSkill]   }
colors={colors} chooseSkill={chooseSkill}/>

                    <button className={"button89"}
                        onClick={() => postSkill(skillId)}> Choose
{skillName}</button>

                </div>
            </div>
        );

};
export default ModalNewSkill;

import React, {useEffect, useState} from 'react';
import "../css/ModalWeaponChoose.css"
import {socket} from "../socket";

//0 lose, 1 victory
```

```
let
img="https://upload.wikimedia.org/wikipedia/commons/thumb/e/e3/Skull
-Icon.svg/2048px-Skull-Icon.svg.png"
let message1="GAME OVER"
let message2="Everyone from your party died"
let message3="Press restart and reconnect to the game to do one more
run"
const ModalGameEnd = () => {
    const [gameOver, setGameOver] = useState(false)
    const [isVictory, setIsVictory] = useState(1)
    useEffect(() => {
        socket.on("GAME END", (data) => {
            setGameOver(true)
            setIsVictory(data.isVictory)
            console.log(isVictory)
            if(data.isVictory==true){
                message1="CONGRATULATION"
                message2="You`ve managed to successfully complete
this dungeon"

img="https://cdn.discordapp.com/attachments/713472920650776657/10857
52389132353576/pngwing.com_15.png?ex=664f5375&is=664e01f5&hm=a39298c
f5cc38b289dbc922d021dd34722ee9dd90e8edc32839b31abadb39218&"
            }
        })
    }, [])
const onRestart = () =>{
    socket.emit("RESTART")
    setGameOver(false)
}

    return (
        <div className={gameOver ? "modal active" : "modal"}>
            <div className="modal__content">
                <b>{message1}</b>
                <img src={img} alt={"victory"} width={300} />
                <p>{message2}</p>
                <p>{message3}</p>
                <button className="button89" onClick={()
=>onRestart()} >Restart</button>
            </div>
        </div>
    );
};

export default ModalGameEnd;
```