**Міністерство освіти і науки України**
**Національний технічний університет**
**«Дніпровська політехніка»**

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

**Кафедра** Програмного забезпечення комп'ютерних систем
(повна назва)

# ПОЯСНЮВАЛЬНА ЗАПИСКА
## кваліфікаційної роботи ступеня

*бакалавра*
(назва освітньо-кваліфікаційного рівня)

**студента** *Міхно Олексія Юрійовича*
(ПІБ)

**академічної групи** *122-18-3*
(шифр)

**спеціальності** *122 Комп'ютерні науки*
(код і назва спеціальності)

**освітньої програми** *Комп'ютерні науки*
(назва освітньої програми)

**на тему:** *Розробка кліент-серверної WEB гри з застосуванням технологій ReactJS та NodeJS*

| **Керівники** | **Прізвище, ініціали** | **Оцінка за шкалою** | | **Підпис** |
|---|---|---|---|---|
| | | **рейтинговою** | **інституційною** | |
| кваліфікаційної роботи | *доц.Удовик І.М.* | | | |
| **розділів:** | | | | |
| спеціальний | *доц.Удовик І.М.* | | | |
| економічний | *доц.Касьяненко Л.В* | | | |
| | | | | |
| | | | | |
| **Рецензент** | | | | |
| **Нормоконтролер** | *доц. Гуліна І.Г.* | | | |

**Дніпро**
**2022**

**Міністерство освіти і науки України**
**НТУ «Дніпровська політехніка»**

**ЗАТВЕРДЖЕНО:**
завідувач кафедри
програмного забезпечення комп'ютерних систем
(повна назва)

_____ І.М. Удовик
(підпис) (прізвище, ініціали)

« » _____ 2022 року

# ЗАВДАННЯ
## на кваліфікаційну роботу

*бакалавра*
(назва освітньо-кваліфікаційного рівня)

**студента** *122-18-3* (група) *Міхно О.Ю.* (прізвище та ініціали)

**тема кваліфікаційної роботи** *Розробка кліент-серверної WEB гри з застосуванням технологій ReactJS та NodeJS*

затверджена наказом ректора НТУ «ДП» від *18.05.2022 р.* № *268-с*

| Розділ | Зміст виконання | Термін виконання |
|---|---|---|
| ***Спеціальний*** | ***На основі матеріалів виробничої практики та інших науково-технічних джерел провести аналіз стану рішення проблеми та постановку задачі.***<br>***Обґрунтувати вибір та здійснити реалізацію методів вирішення проблеми*** | ***13.05.2022 р.*** |
| ***Економічний*** | ***Провести розрахунок трудомісткості розробки програмного забезпечення, витрат на створення ПЗ й тривалості його розробки*** | ***27.05.2022 р.*** |

Завдання видав _____ (підпис) *доц.Удовик І.М.* (посада, прізвище, ініціали)

Завдання прийняв до виконання _____ (підпис) *Міхно О.Ю.* (прізвище, ініціали)

Дата видачі завдання: *14.01.2022 р.*

Термін подання кваліфікаційної роботи до ЕК: *13.06.2022 р.*

# РЕФЕРАТ

Пояснювальна записка: 102 с., 51 рис., 1 табл., 3 дод., 20 джерел.

Об'єкт розробки: комп’ютерна гра.

Мета кваліфікаційної роботи: розробка браузерної гри, серверної частини на платформі NodeJS, а клієнтської на фреймворці ReactJS.

У вступі розглядається аналіз та сучасний стан проблеми, конкретизується мета кваліфікаційної роботи та галузь її застосування, наведено обґрунтування актуальності теми та уточнюється постановка завдання.

У першому розділі проаналізовано предметну галузь, визначено актуальність завдання та призначення розробки, сформульовано постановку завдання, зазначено вимоги до програмної реалізації, технологій та програмних засобів.

У другому розділі проаналізовані наявні рішення, обрано платформи для розробки, виконано проектування і розробка програми, описана робота програми, алгоритм і структура її функціювання, а також виклик та завантаження програми, визначено вхідні і вихідні дані, охарактеризовано склад параметрів технічних засобів.

В економічному розділі визначено трудомісткість розробленої інформаційної системи, проведений підрахунок вартості роботи по створенню програми та розраховано час на його створення.

Практичне значення полягає в реалізації сучасними засобами прототипу браузерної гри, яка призначена для розважальних цілей.

Актуальність розробленого проекту полягає у тому, що в наш час є доцільним використання комп’ютерних та мобільних ігор для розваг, але більшість таких проектів має великий недолік: системні вимоги. Але саме цій проект жанру BBMMORPG реалізований із використанням технологій React та NodeJS із вимог вимагає лише наявності браузера та мінімального інтернет з'єднання, що надасть можливість розважатися більшій кількості людей.

Список ключових слів: КЛІЄНТ, СЕРВЕР, ХАРАКТЕРИСТИКИ, ДОСВІД, ГРАВЕЦЬ, ПЕРСОНАЖ, ПРЕДМЕТ, ЛОКАЦІЯ.

# ABSTRACT

Explanatory note: 102 p., 51 figs., 1 table, 3 apps., 20 sources.

Object of development: Computer game.

The purpose of the qualification work: development of a browser game, a server part developed on NodeJS platform, client side developed with ReactJS framework.

The introduction examines the current problem statement, specifies the purpose of the qualification work and the area of its application, justifies the relevance of the topic and specifies the problem statement.

In the first section carries out the analysis of the subject area, determines the relevance of the task and the dedication of the development, creates task statement, the software and hardware requirements of the product, specifies technologies and tools for development.

In the second section analyzes the existing solutions, chose the platform for development, creates design and finish development of the product, describes the algorithm, structure and architecture solutions in the system, defines the input and output data, describes characteristics of the technical resources used, describes how to run a program, features of user interaction, differences between serve and client parts of product.

The economic section defines the complexity of the information system, calculates the cost of work on creating the application and defines the time for its creation.

The practical significance of the work lies in the implementation of browser game prototype, using modern tools, which is designed for entertainment purposes

The relevance of the developed project is that nowadays it is advisable to use computer and mobile games for entertainment, but most of these projects have a major disadvantage: system requirements. But this BBMMORPG project, implemented using React and NodeJS technologies, requires only a browser and a minimum Internet connection, which will provide an opportunity for more people to entertain.

Keywords: CLIENT, SERVER, CHARACTERISTICS, EXPERIENCE, PLAYER, CHARACTER, ITEM, LOCATION.

# СПИСОК УМОВНИХ ПОЗНАЧЕНЬ

СУБД - Система управління базами даних;

БД - База даних;

SSE – Server-send events;

СУБД – Система управління базами даних;

ПЗ – Програмне забеспечення;

БД – База даних;

JS – JavaScript;

DOM – Document Object Model;

HTML – HyperText Markup Language;

CSS – Cascading Style Sheets;

JWT – JSON Web Token;

API – Application programming interface;

NPM – Node Packet Manager;

JSON – JavaScript Object Notation.

**ЗМІСТ**

РЕФЕРАТ ..... 3
ABSTRACT ..... 4
СПИСОК УМОВНИХ ПОЗНАЧЕНЬ ..... 5
ВСТУП ..... 8
РОЗДІЛ І ..... 9
АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ПОСТАНОВКА ЗАДАЧІ ..... 9
1.1. Загальні відомості з предметної галузі ..... 9
1.2. Призначення розробки та галузь застосування ..... 13
1.3. Підстави для розробки ..... 14
1.4. Постановка завдання ..... 14
1.5. Вимоги до програми або програмного виробу ..... 16
1.5.1. Вимоги до функціональних характеристик ..... 16
1.5.2. Вимоги до інформаційної безпеки ..... 18
1.5.3. Вимоги до складу та параметрів технічних засобів ..... 20
1.5.4. Вимоги до інформаційної та програмної сумісності ..... 20
РОЗДІЛ 2 ..... 21
ПРОЕКТУВАННЯ ТА РОЗРОБКА ІНФОРМАЦІЙНОЇ СИСТЕМИ ..... 21
2.1. Функціональне призначення системи ..... 21
2.2. Опис застосованих математичних методів ..... 21
2.3. Опис використаних технологій та мов програмування ..... 23
2.4. Опис структури системи та алгоритмів її функціювання ..... 29
2.5. Обґрунтування та організація вхідних та вихідних даних програми ..... 41
2.6. Опис розробленої системи ..... 42
2.6.1. Використані технічні засоби ..... 42
2.6.2. Використані програмні засоби ..... 42
2.6.3. Виклик та завантаження програми ..... 44
2.6.4. Опис інтерфейсу користувача ..... 45
РОЗДІЛ 3 ..... 58

ЕКОНОМІЧНИЙ РОЗДІЛ ..... 58
3.1. Розрахунок трудомісткості та вартості розробки програмного продукту ... 58
3.2. Розрахунок витрат на створення програми ..... 60
ВИСНОВКИ ..... 62
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ ..... 64
ДОДАТОК А КОД ПРОГРАМИ ..... 66
ДОДАТОК Б ВІДГУК КЕРІВНИКА ЕКОНОМІЧНОГО РОЗДІЛУ ..... 101
ДОДАТОК В Перелік файлів на диску ..... 102

# ВСТУП

Завдання даної кваліфікаційної роботи та об’єкт його діяльності безпосередньо пов’язані з напрямом підготовки та відповідає узагальненій тематиці кваліфікаційних робіт і переліку зазначених виробничих функцій, типових задач діяльності, умінню та компетенціям, якими повинні володіти бакалаври напряму 122 «Комп’ютерні науки».

За останні ІІІ століття навколо відеоігор сформувалася ціла всесвітня субкультура. Відеоігри давно перестали бути просто розвагою для дітлахів або способом "вбити час". Вони перетворилися на багатогранну індустрію, окремий вид сучасного мистецтва, що охоплює людей різних інтересів та поколінь. Яскравим прикладом того, що ігри це не просто забава, а цілий окремий всесвіт, це - Massively Multiplayer Online games.

ММО, це онлайн-відеогра з великою кількістю гравців, часто сотнями або тисячами. Ці ігри дозволяють гравцям співпрацювати, змагатися один з одним у великих масштабах, а іноді й осмислено взаємодіяти з людьми по всьому світу.

Метою кваліфікаційної роботи є створення прототипу гри жанру Browser Based Massively Multiplayer Online Role-Playing Game. Цей жанр є перетином масових, розрахованих на багато користувачів рольових онлайн ігор та браузерних ігор. Відмінною особливістю жанру є те, що для гри потрібно лише браузер і мінімальне інтернет з'єднання. Це має значий плюс: гру не потрібно завантажувати та встановлювати.

В результаті виконання кваліфікаційної було створено два окремих повноцінних додатків у вигляді серверної частини реалізованної на платформі NodeJS, і кліентської частини, реалізованій на сучасній, популярній та швидкій біліотеці – React. Кінцевий запропонований та працездатний продукт готовий для розширення, і наповнення контентом.

# РОЗДІЛ І
# АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ПОСТАНОВКА ЗАДАЧІ

## 1.1. Загальні відомості з предметної галузі

Всесвітня павутина – це мережа, що безперервно розвивається, і на сьогодні вона пішла далеко від своїх концепцій 1990-х років. Великий обсяг даних був занадто великий, щоб передавати його великому числу людей у всьому світі. Тоді Тім Бернерс-Лі, один із фахівців ЦЕРН, вигадав спосіб навігації між ними за допомогою протоколу передачі гіперпосилань (Hyper Text Transfer Protocol (HTTP). Він також створив спеціальну мову розмітки, названу мовою гіпертекстової розмітки - Hyper Text Markup Language (HTML). Для того, щоб зібрати все це воєдино, він створив перші браузер та веб-сервер.

Веб-додаток є веб-сайтом, на якому відображені сторінки з не повністю сформованим вмістом. Принцип роботи веб-додатків заснований на взаємодії між браузером, запущеним на комп'ютері кінцевого користувача, та веб-сервером. Остаточно їхній зовнішній вигляд формується в результаті запиту з боку клієнта на сервер, через що такі сторінки називають динамічними. Завдання сервера полягає в тому, щоб прийняти запит від клієнта та спробувати дати на нього змістовну відповідь, зазвичай передаючи йому запитану веб-сторінку. Насамперед, він передає запит програмі, яка називається сервером додатків. Тут аналізується код, здійснюється доступ ресурсів та формується веб-сторінка.

Щоб відокремлювати оформлення сайту від вмісту, використовується такий ресурс як база даних, яка спрощує доступ до інформації, структурує її та надає методи управління ними. Запити до неї здійснюються за допомогою мови SQL (синтаксис зазвичай залежить від використовуваної СУБД).

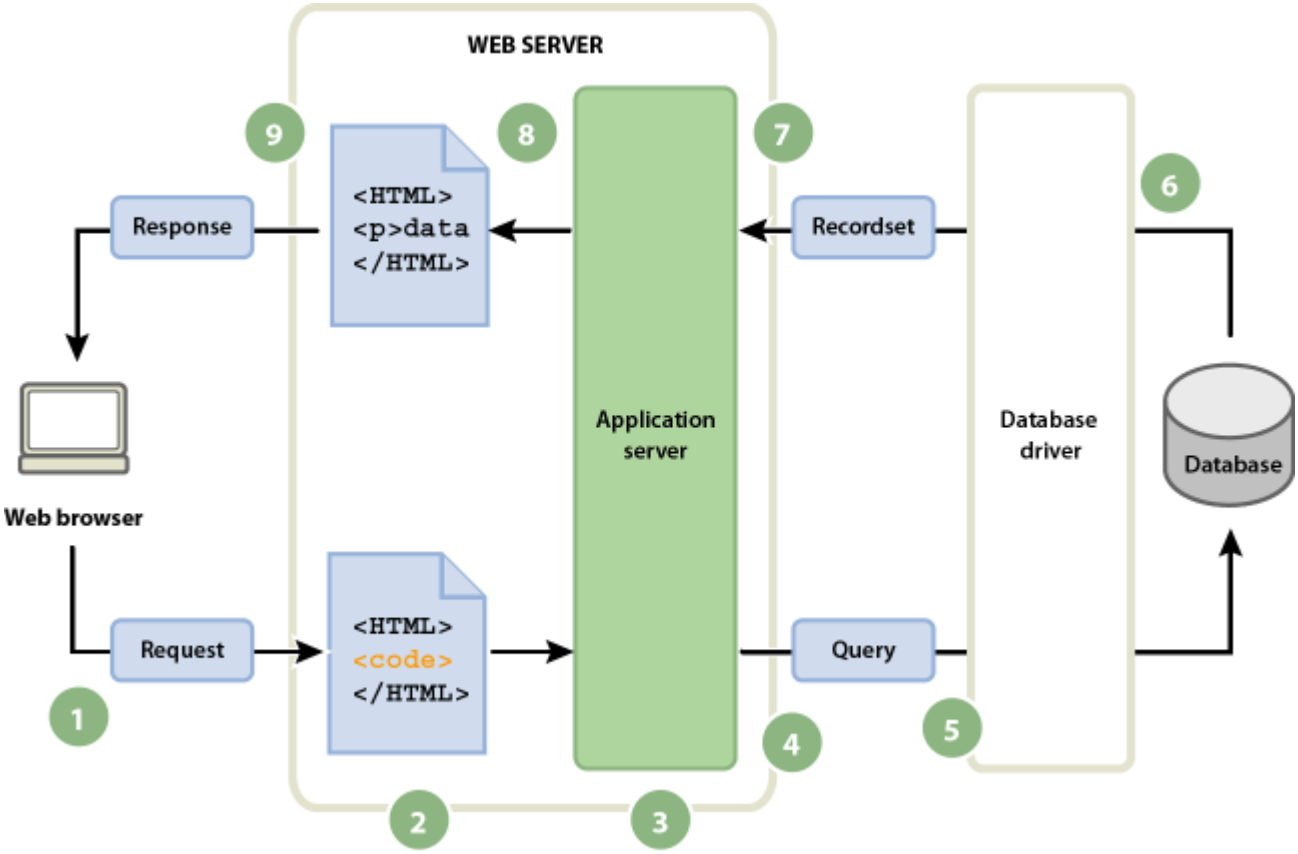


Рис. 1.1. Процес взаємодії між клієнтом та сервером у динамічному веб-додатку

Як і описувалося раніше, клієнт формує запит до сервера, а той, у свою чергу, посилає запити до бази даних, отримує необхідну інформацію і перетворює її на програмний код і формує HTML-розмітку, яку відправляє клієнту і яку у своєму браузері бачить користувач.

HTML – це мова опису структури веб-сторінок. Сторінки, створені за його допомогою, можуть бути відкриті лише за допомогою спеціальних програм (браузерів). Для своєї основної функції HTML використовує теги.

У заголовку міститься тег <!DOCTYPE [стандарт HTML]>. Тут зазначається версія мови. На момент виконання курсової роботи актуальною версією є стандарт HTML5.

Весь код сторінки міститься у тегах <html>…</html>. Всередині <head>…</head> вказується заголовок сторінки, підключаються скрипти та стилі, вказуються такі метадані, як кодування, ключові слова, актальність тощо.

Всередині тегів <body>…</body> розміщуються елементи, які безпосередньо відображаються в браузері. Тут можна розділити блоки за

змістом, вказати абзаци, створити таблиці, списки та елементи форми. Також в аттрибутах вказуються різні параметри на кшталт ідентифікаторів та імен класів, які будуть застосовуватися при стилізації CSS.

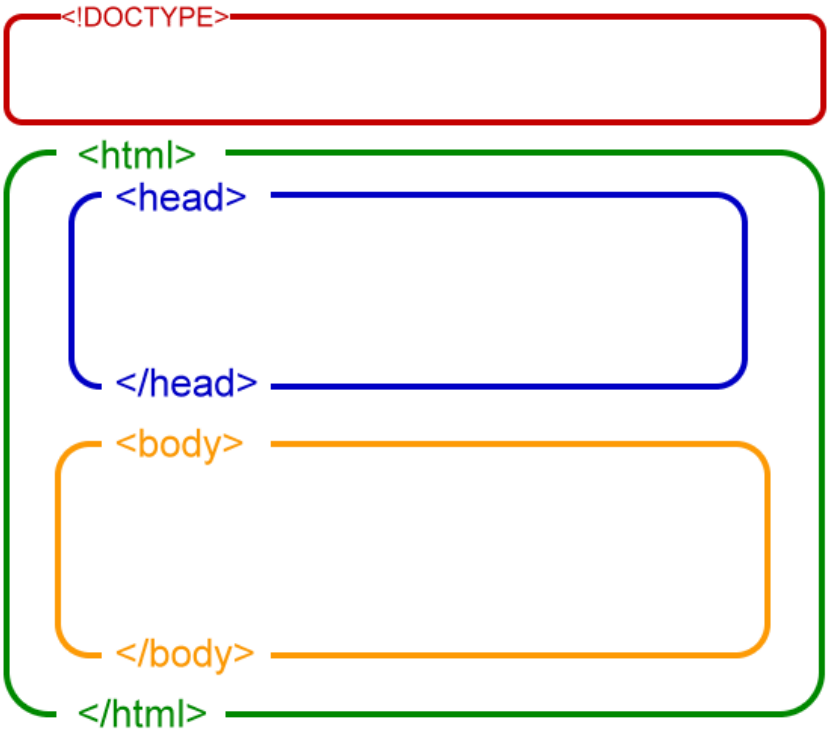


Рис. 1.2. Структура HTML-документу

Для стилізації елементів розмітки використовують так звані каскадні таблиці стилів або, простіше кажучи, CSS. Основною метою розробки CSS було поділ логічного оформлення сторінок та опис їх зовнішнього вигляду.

Тут можна регулювати шрифти, межі та розміри блоків, відстані між елементами, фон, кольори та багато іншого.

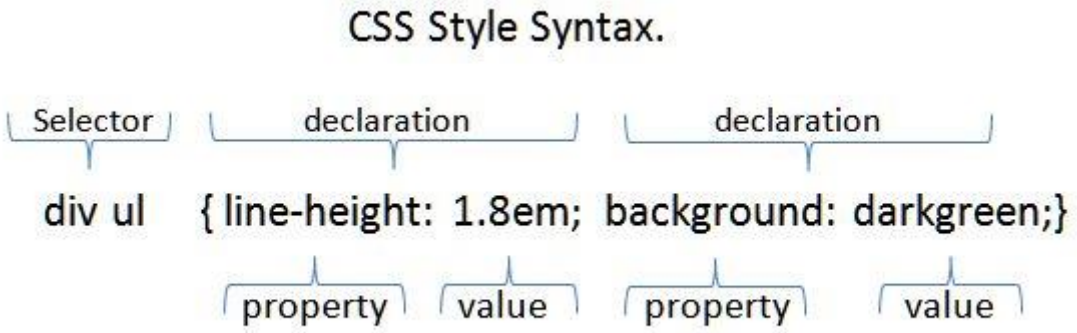


Рис. 1.3. Синтаксис CSS

Спочатку вказується селектор - елемент або група елементів, до яких буде використано стилі. Це може бути певний тип елементів, клас або унікальний ідентифікатор. Можна вказувати, чиїх спадкоємців оформляти. Допускається перерахування селекторів через кому.

Параметри оформлення вказуються усередині фігурних дужок. Пара властивість – значення розділяється двокрапкою, між парами обов'язково наявність точки з комою.

Для додавання інтерактивності елементам розмітки використовується мова JavaScript або її численні модифікації. JavaScript є об'єктно-орієнтованою мовою, але прототипування, що використовується в мові, обумовлює відмінності в роботі з об'єктами в порівнянні з традиційними клас-орієнтованими мовами. Крім того, JavaScript має ряд властивостей, властивих функціональним мовам – функції як об'єкти першого класу, об'єкти як списки, каринг, анонімні функції, замикання – що надає мові додаткової гнучкості.

React - це сучасна JavaScript бібліотека для створення інтерфейсів користувача від Facebook. Вона була розроблена з нуля, з упором на продуктивність. Важливо розуміти, що React – це лише рівень вистави, а не повноцінний фреймворк на кшталт Angular або Vue. Він використовується власну специфікацію JavaScript, звану JSX. Її синтаксис дозволяє використовувати синтаксис HTML у коді сценарію. ReactJS маніпулює віртуальним DOM-деревом, оскільки взаємодія з реальним DOM – дуже дорога операція. Кожен компонент бібліотеки – це об'єкт, який має свій життєвий цикл і який можна відобразити в браузері. Якісь методи життєвого циклу викликаються при малюванні компонента, якісь – при його перемальовуванні; і, нарешті, є функції, які викликаються перед знищенням об'єкта. За малювання відповідає метод render().

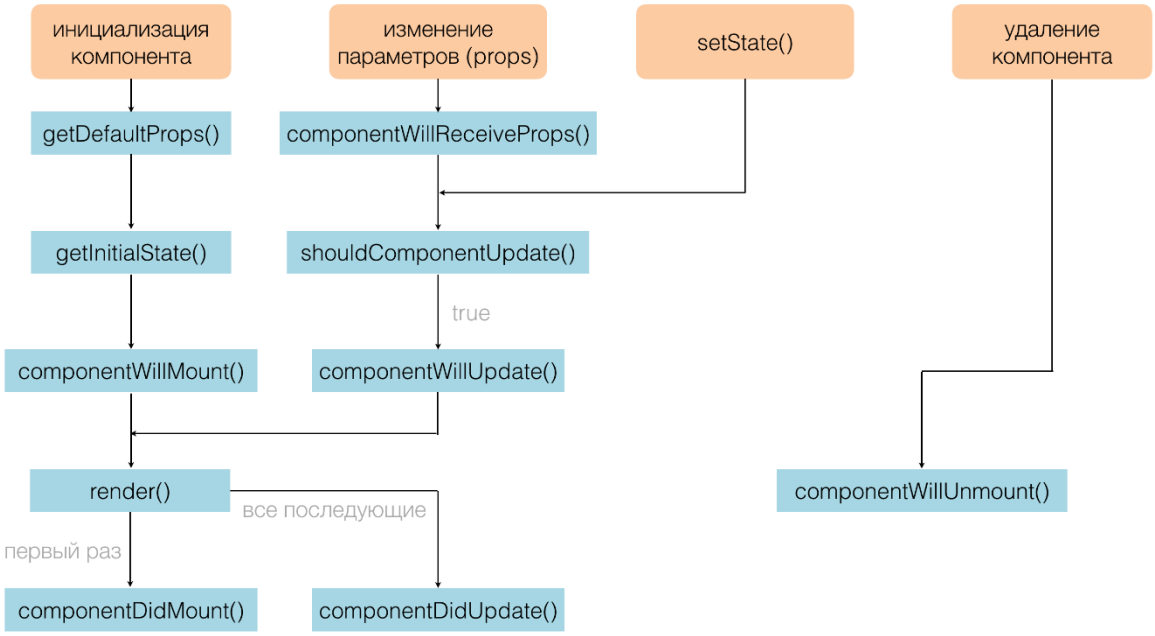


Рис. 1.4. Функції життєвого циклу компонента React

## 1.2. Призначення розробки та галузь застосування

Комп'ютерні ігри користуються величезною популярністю у мільйонів користувачів різного віку. Особливо гостро це відчувається в період ізоляції, коли навколишнє оточення диктує необхідність шукати нові хобі в домашніх умовах. Підвищений інтерес до цієї сфери призвів до того, що в даний час користувачі мають доступ до ігор, що сподобалися, з будь-якого пристрою. Виробники пропонують не тільки ігри для ПК, але і для консолей та мобільних пристроїв. І хоча ігри часто вважають марним заняттям, вони мають розважальний, творчий, котрий іноді навчальний характер. Адже, як відомо, навчання проходить успішніше в ігровій формі. Наступні технічні досягнення дозволили іграм дедалі більше розвиватися з погляду графіки. У 1997 році вийшла перша рольова гра, доступна відразу для багатьох користувачів. Ринок консолей став поповнюватися популярними і в наш час Playstation та Xbox, а стрімкий розвиток смартфонів у нульових створив новий величезний ринок для розробників ігор. Мобільні ігри сьогодні – це багатомільярдний бізнес. У всьому світі проходять навіть кіберспортивні турніри з мобільних ігор, а кількість завантажень багатьох з них перевищила мільйони.

Актуальність розробленого проекту полягає у тому, що в наш час є доцільним використання комп'ютерних та мобільних ігор для розваг, але більшість таких проектів має великий недолік: системні вимоги. Але саме цій проект жанру BBMMORPG реалізований із використанням технологій React та NodeJS із вимог вимагає лише наявності браузера та мінімального інтернет з'єднання, що надасть можливість розважатися більшій кількості людей.

## 1.3. Підстави для розробки

Відповідно до освітньої програми, згідно навчального плану та графіків навчального процесу, в кінці навчання студент виконує кваліфікаційну роботу.

Тема роботи узгоджується з керівником проекту, випускаючою кафедрою, та затверджується з наказом ректора.

Таким чином підставами для розробки (виконанням кваліфікаційної роботи) є:

– освітня програма спеціальності 122 "Комп'ютерні науки";

– навчальний план та графік навчального процесу;

– наказ ректора Національного технічного університету "Дніпровська політехніка" № 268-с від 18.05.2022р;

– завдання на кваліфікаційну роботу на тему "Розробка кліент-серверної WEB гри з застосуванням технологій ReactJS та NodeJS ".

## 1.4. Постановка завдання

Метою проекту є створення двох окремих повноцінних додатків у вигляді серверної частини реалізованної на мові програмування NodeJS, і кліентської частини, реалізованій на сучасній, популярній та швидкій біліотеці – React. Кінцевий запропонований та працездатний продукт готовий для розширення, і наповнення контентом.

Ігровий процес:

На началі користувачеві потрібно зареєструватися, обрати собі унікальне імя персонажу та пароль. Після реєстрації користувач попадає на стартову локацію, і йому треба вирушати в околиці міста, де йому належить розвивати свого персонажа.

Щоб розвивати персонажа користувачеві потрібно боротися з монстрами і рейдовими боссами. У разі перемоги користувач буде отрмувати деяку кількість очків досвіду, а при накопиченні певної його кількості, він буде повишати свій рівень.

З кожним підвищенням рівня, користувач буде отримувати певну кількість очків навичок, які йому треба розподілити меж певними параметрами персонажа, такі як: сила, захист, швидкість та витрівалість. Кожен з цих параметрів напряму беспосередньо впливає на потужність персонажу. Також з підвищенням рівня персонаж повишає свою максимальну кількість очків здоров'я. Проте з кожним рівнем необхідна кількість очків досвіду росте, тож для більш ефективного розвитку персонажу, йому треба вирушати на наступний рівень, де мешкають більш сильніші монстри, за перемогу над котрими, користувачеві будуть надавати більшу кількість очків досвіду.

В околицях є декілька локацій в яких мешкають різноманітні супротивники. Але щоб перейти на наступний рівень користувачеві потрібно спочтку перемогти рейдс босса цього рівня. Зробити це буде достатньо складно, тож користувачеві треба заздалегідь підготуватися: запастися зіллями лікування, потужою зброєю та міцнею бронею. Все це може користувач може отримати після перемоги, або купити в магазині у місті. У разі поразки, персонаж буде воскресений, але це буде впливати на загальний рейтинг.

Задачами серверної частини э:

– надсилати код кліентської частини для подальшої взаємодії з сервером;

– при запиті від користувача, робити запит до бази даних, обробляти її да надавати належну відповідь у форматі JSON;

– мати налаштований SSE end-поінт, а також тримати актуальну інформацію щодо підключених користувачів на даний момент;

– при зміні ігрових даних, надислати цю інформацию користувачам, належним до цих самих данних;

– мати код, який легко масштабується, щоб потім «наповнювати» гру контентом.

Задачами клієнтської частини є:

– мати адаптивний та зручний інтерфейс;

– тримати keep-alive – з'єднання з серверним SSE end-поінтом;

– надавати запрос до серверу у належному форматі;

– корректно обробляти інформацію отриману як зі звичайних запросів, так від SSE.

## 1.5. Вимоги до програми або програмного виробу

### 1.5.1. Вимоги до функціональних характеристик

Цей проект гри складається з двух окремих додатків: серверної частини на платформі NodeJS, та клієнтської частини на фреймворці ReactJS, тож для кожної з них будуть свої вимоги до функціональних характеристик:

Вимоги до функціоналу клієнтської частини:

1. Зберігання та подальше використання JWT Access токену.

2. При запуску додатка підлючатися до SSE з метою отримувати, те реагувати на актуальну інформацію в режимі «real-time», таку як: повідомлення системи, повідомлення користувачів чату, інформація щодо отриманих винагород, тощо.

3. Отримувати інформацію о поточній локації, та відображати наступну інформацію: назву, зображеня, доступні для переміщення локації (іх зображеня та назву), магазини, доступні вороги для битви а також поточні очки навичок.

4. У шапці додатку гравець повинен мати шкалу здоров'я, ім'я а також рівень персонажу. Якщо здоров'я персонажу закінчиться, то користувачеві має бути відображено вікно смерті.

5. У подвалі сайту мають бути кнопки «локація», «інвентар», «профіль», «чат» та «головне меню», при натисканні на обрану кнопку буде відображене обране вікно.

6. При відкритті вікна бою, треба отримувати інформацію про обраного супротивника. Якщо користувач атакує, динамічно відображати результати атаки: записувати лог з нанесеним та отриманим уроном, плавно змінювати полоску здоров'я до потрібного значення. В заложності від параметру персонажу «швидкість», динамічно впливати на швидкість атак, які може проводити користувач.

7. При відкритті вікна з чатом, отримувати та відображати останні повідомлення користувачів, а також динамічно добавляти і відображати нові повідомлення отримані через SSE.

8. При відкритті вікна з персонажем, отримувати та відображати поточні навчики персонажа такі як: сила, захист, швидкість та витрівалість. Також відображати поточну кількість очків, кількість, що залишилася до нового рівня і кількість грошей.

При відкритті вікна з інвентарем та відображати всі існуючі категорії предметів, наприклад: «зброя», «зілля», «ресурси». Якщо обрати одну із категорій, то отримувати та відображати існуючі предмети обраної категорії, власником котрих є користувач.

Вимоги до функціоналу серверної частини:

1. Надсилати скомпільований js-файл клієнту.

2. При авторизації надислати Refresh та Access токени.

3. Тримати постійне з'єднання з базою даних.

4. Мати налаштований SSE end-поінт, а також тримати актуальну інформацію щодо підключених користувачів на даний момент.

5. При запиті від користувача, робити запит до бази даних, обробляти її та надавати належну відповідь у форматі JSON.

### 1.5.2. Вимоги до інформаційної безпеки

Безпека є невід'ємною частиною програмного забезпечення, яке використовується для виявлення слабких місць, ризиків або загроз у програмному додатку, а також допомагає нам зупинити неприємну атаку з боку сторонніх осіб і переконатися в безпеці наших програм.

Однак, коли ми говоримо про Інтернет, важливість безпеки зростає в геометричній прогресії. Якщо онлайн-система не може захистити дані транзакції, то нікому й не спаде на думку їх використовувати. Безпека - це ще не слово, яке шукає свого визначення, і не тонке поняття. Однак ми б хотіли перерахувати деякі компліменти щодо безпеки.

Існує декілька основних принципів для забезпечення безпеки програмного забезпечення:

Не відмовність - використовується як посилання на цифрову безпеку, і це спосіб гарантії того, що відправник повідомлення не може не погодитися з тим, що він надіслав повідомлення, і що одержувач не може відмовитися від отримання повідомлення.

Авторизація - процес визначення того, що клієнту дозволено виконувати дію, а також отримувати послуги. Прикладом авторизації є контроль доступу.

Конфідиційність - це властивість інформації, яка полягає в тому, що інформація не надається чи не розголошується неавторизованим особам, організаціям або процесам.

Доступність - Це означає, що обчислювальні системи, що використовуються для зберігання та обробки інформації повинні функціонувати належним чином, та бути захищеними. Системи високої доступності мають на меті залишатися доступними в будь-який час, запобігаючи перебої в роботі через відключення електроенергії, збої обладнання та оновлення системи. Забезпечення доступності також передбачає запобігання атак.

Всі сайти де є хоча б які конфедеційні дані, є привабливими цілями для хакерів через особисту інформацію користувачів. Навіть якщо система не

обробляє ні яких операції з картами безпосередньо, пошкоджений веб-сайт може перенаправити клієнтів на підроблену сторінку. В наслідок взлому клієнти можуть зазнати втрат конфіденційних даних, а продавці зіштовхнуться з втратою репутації, користувачів, та навіть, можливою загрозою судового позову.

Також є ще один важливий захід безпеки — CORS. Політика CORS розшифровується як Cross-Origin Resource Sharing. Це дозволяє нам посилити безпеку, що застосовується до API. Це робиться шляхом обходу заголовків Access-Control-Allow-Origin, які вказують, які джерела можуть отримати доступ до API. Іншими словами, CORS — це функція безпеки браузера, яка обмежує HTTP-запити між джерелами з іншими серверами та визначає, які домени мають доступ до ваших ресурсів.

Коли розгортається програма на сервері, він не повинен приймати запити з кожного домену. Замість цього нам треба вказати, яке джерело може робити запити до нашого серверу. Таким чином, ми можемо блокувати користувачів, які намагаються клонувати наш сайт або робити запити з неавторизованих серверів..

Для досягнення максимальної безпеки проекту необхідно ретельно вибрати серверного провайдера, бо розміщуючи файли на віддаленому веб-сервері провайдера, він несе повну відповідальність за їх збереження та безпеку. Також треба зажди перевіряти дані, які віправляє користувач.

### 1.5.3. Вимоги до складу та параметрів технічних засобів

Клієнстька частина вимягає для себе лише постійного інтенет з'єднання, і наявність встановленого браузера, який підтримує cookies та javascript. Незважаючи на те що користувацький інтерфейс є цілком адаптивним, вимагається щоб розмір екрану був 370x420px або більше.

Для підтримки стабільної роботи серверної частини, трема мати таки характеристики системи, або краще:

- операційна система: Windows 10, Windows Server 2016 або вище;
- оперативної пам'яті 6 гігабайт;
- процесор x64 з тактовою частотою 1,6 ГГц;
- 10 гігабайт вільного місця на диску.

### 1.5.4. Вимоги до інформаційної та програмної сумісності

Для запуску серверної частини потрібне таке ПЗ:

- Будь-який web-сервер, але рекомендовано обрати Apache або nginx;
- MySQL or MariaDB 5.5 або новіше;
- PHP 7.2.5 або новіше;
- NodeJS 14.17.5;

Для запуску кліентської частини необхідне таке ПЗ:

- Будь-який із популярних браузерів, включаючи Internet Explorer 9+

Для розробки необхідні наступні компоненти:

- редактор коду (SublimeText, VS Code, або будь-який інший);
- NodeJS 14.17.5;
- NPM 6.14.13;
- .NET Framework 4.8 (Для запуску NPM)
- Webpack (або будь-який інший пакувальник: Parcel або Rollup).

А також потрібні наступні бібліотеки: React, React Rouer, axios, event-source-polyfill, bcrypt, cors, express, express-validator, jsonwebtoken, mysql2, node-cron.

# РОЗДІЛ 2
# ПРОЕКТУВАННЯ ТА РОЗРОБКА ІНФОРМАЦІЙНОЇ СИСТЕМИ

## 2.1. Функціональне призначення системи

Призначення розробки – створення гри що складається з двух окремих додатків: серверної частини релізованої на платформі NodeJS, клієнтської на фреймворці ReactJS.

Після рєєстрації гравець опиняється на першому рівні. Переміщеня персонажа відбувається за допомогою запросів до серверу, після чого сервер запусеє нову локацію персонажу до бази даних, і надсилає відповідь з інформацією про нову локацію. Щоб перейти на локацю наступного рівня гравцям необхідно перемогти рейд-босса цього рівня.

Гравцю треба розвивати свого персонажа завдяки перемогам над монстрами. З них він отримає ігрову валюту та предмети. Непотрібні предмети гравець може продати у відповідному магазині, там же він може придбати нові ігрові предмети. Гравець поступово покарщує характеристики свого персонажу за допомогою очків навіків, які він отримує після підвіщення свого рівня.

Інсують чотири основних характеристики.

1. Сила – беспосередньо впливає на кількість наносимого урону.
2. Захист – снижує кількість отримуємого урону.
3. Витріваліть – підвищує кількість очків здоров'я що відновлюються.
4. Швидкість – впливає на швидкість нанесення ударів яку персонаж.

## 2.2. Опис застосованих математичних методів

Нагородою за перемогу є ігрові предмети. Вони падають з шансом зазначеним в базі даних та є числом в діапазоні від 0 до 1. Шанс розраховуеться з використанням видадкового числа, якщо воно меньше або дорівню шансу то предмети випадают. Їхня кількість може не тількі дорівнювати одиниці,

наприклад 5-7 шт. Для розрахонку кількості предметів які гравець отпримає, використовується наступна формула.

$$Amount = X \cdot \lfloor (Max - Min + 1) + Min \rfloor \quad (2.1)$$

де, ⌊ ⌋ - округлення до меньшого числа, X – випадкове число в діапазоні від 0 до 1, $D_{min}$ – мінімальна можлива кількість предметів що може отримати гравець, а $D_{max}$ – максимальне.

Приклади розрахунків при $D_{max} = 10$, $D_{min} = 5$:

⌊0.15 * (10 - 5 + 1) + 5⌋ ≈ ⌊5.9⌋ = 5;

⌊0.55 * (10 - 5 + 1) + 5⌋ ≈ ⌊8.3⌋ = 8;

⌊0.99 * (10 - 5 + 1) + 5⌋ ≈ ⌊10.94⌋ = 10;

Отже, чим більше ми отримаємо випадкове число, тим більше ми отримаємо предметів.

Для підвищеня рівня персонажу треба набрати певну кількість очків досвіду, ця кількість розраховується за формулою:

$$xp = \left(\frac{LVL}{X}\right)^2 \quad (2.2)$$

де, xp – потрібна кількість очків досвіду для отримання наступного рівня персонажу, lvl – поточний рівень гравця, X – константа, що впливає на важкість нового рівня та підвищує кількість очків досвіду, яка потрібна для підвищення рівню персонажа. Завдяки ступені 2 у формулі, потрібна кількість очків зростає у геометричній пргресії.

При розрахунку кількості здоров'я, яке гравець зносить противникам використовуеться наступна формула:

$$Dmg = Att + \lceil C \cdot \frac{Att}{5} \rceil \quad (2.3)$$

де, Dmg – кількість нанесеного урону, ⌈ ⌉ - округлення до більшого числа, Att – характеристика атаки персонажу, C – випадкове число. Тобто при кожній атаці урон може зрости на четверть, а може й не зрости зовсім.

У відповідь супротивник теж б’є, але його урон обчислюється іншою формулою:

$$Dmg = Att \cdot \left\lceil \left( \frac{X}{X + Def} \right) \right\rceil \quad (2.4)$$

де, Dmg – кількість нанесеного урону, Att – сила атаки супротивника, X – константа яка встановлюється згідно балансу, Def – кількість очків захисту персонажу. Константа X – це значення яке дорівнює середньому теоретичному значеню очків захисту усіх персонажів.

Таблиця 2.1

**Кількість очків досвіду для підвищення рівня**

| Рівень | Необхідна кількість очків | Загальна кількість очків |
|---|---|---|
| 1 | 0 | 0 |
| 2 | 400 | 400 |
| 3 | 500 | 900 |
| 4 | 700 | 1600 |
| 5 | 900 | 2500 |
| 6 | 1100 | 3600 |
| 7 | 1300 | 4900 |
| 8 | 1500 | 6400 |

## 2.3. Опис використаних технологій та мов програмування

Для створення клієнтської частини гри використовувались наступні технології та мови програмування:

- React.
- JSX.
- Бібліотека event-source-polyfill.
- Бібліотека Axios.
- CSS.

React — це клієнтська платформа JavaScript, яка дозволяє покращити розмітку HTML за допомогою спеціального JavaScript, щоб забезпечити складні взаємодії користувачів. React використовує міні-мову під назвою JSX, щоб дозволити вам писати розмітку в стилі HTML всередині коду JavaScript або TypeScript.

У React розмітка написана за допомогою власної мови React JSX, і React повністю контролює те, що записується в DOM в його межах. Цей інструмент є унікальним, через те, як він оброблює свій внутрішній стан. В ньому стан зберігається в системі. Коли стан змінюється, цикл виконання React автоматично оновлює DOM, щоб відобразити новий стан. Коли у інших інструментах стан зберігається за допомогою самого DOM або на сервері і відповідаємо за оновлення сторінки при зміні стану.

Підхід React до відображення значень називається декларативним або реактивним. Обидва терміни відносяться до одного і того ж процесу. Коли справа доходить до відображення інформації в React, робота розробника полягає в тому, щоб оголосити в коді, як має виглядати вихід, і завдання системи реагувати на зміни значень шляхом перемальовування сторінки.

React керує цим процесом, підтримуючи власну віртуальну DOM і використовуючи цю віртуальну DOM для оновлення лише тих частин фактичної DOM, які змінюються при зміні значення стану. Однак, щоб дозволити DOM автоматично змінюватися, React має бути проінформований, коли ви змінюєте значення, яке має відношення до відображення в браузері. Через це ви можете змінювати стан у React лише за допомогою певних функцій, наданих фреймворком.

Більшість коду, який пишеться у React, буде в компоненті. Компонент React — це функція, яка веде себе як шаблон, поєднуючи дані з розміткою, написаною за допомогою JSX, що призводить до HTML, який надсилається до DOM. Як ми побачимо, JSX дозволяє змішувати логіку JavaScript з тегами HTML і викликами інших компонентів React.

JSX — це розширення JavaScript, яке дозволяє включати HTML-подібні елементи в кутових дужках у код JavaScript або TypeScript. React використовує JSX для визначення тегів HTML, які є частиною наших компонентів React, а також як спосіб виклику компонентів React з інших компонентів React. Парсер React JSX перетворює наші файли .jsx і .tsx у звичайний JavaScript або TypeScript

Axios — це HTTP-клієнт на основі Promises для JavaScript, який можна використовувати як в інтерфейсній програмі так і в серверній частині Node.js. Використовуючи цю бібліотеку, дуже легко надсилати асинхронний HTTP-запити на кінцеві точки REST та виконувати операції CRUD. Бібліотеку Axios можна застосовувати в звичайній програмі на чистому JavaScript або разом із більш просунутими фреймворками, такими як React. Однією з переваг використання Axios - він дозволяє уникнути написання великих обсягів шаблонного коду та зробити код чистішим і зрозумілішим.

Бібліотека event-source-polyfill необхідна для реалізації існуючих функцій JavaScript у браузерах, які їх не підтримують. Саме у цьому проекті ця бібіотека використовулася для додаткових функцій SSE.

SSE — технологія надсилання повідомлень від сервера до браузера. Клієнт підключається до стриму оновлень і автоматично отримує оповіщення у разі нових подій. Використання саме цієї технології у саме цьому проекті має свої переваги. Наприклад якщо виокристовувати long polling програма перевіряє нові дані на сервері наступним чином: клієнт створює запит і чекає на відповідь. Сервер повертає відповідь, навіть якщо оновлень немає. Отже, процес створює додаткове навантаження на канал. При використанні webscoket - інша більш потужна технологія обміну повідомленнями між клієнтом та сервером. Її головна відмінність у тому, що зв'язок двосторонній, клієнт приймає та відправляє повідомлення. Він використовується там, де потрібен саме двусторонній зв'язок у режимі реального часу.

У звичайному SSE немає можливості надсилати authorization header до серверу. Саме через це була використана бібліотека event-source-polyfill, адже вона дозволяє це робити.

Для розробки серверної частини гри застосовувались наступні технології та мови програмування:

– NodeJS.
– NPM.
– SQL.
– Express.JS.
– Node-Cron.
– Bcrypt.

Node.js – простими словами, це рішення на стороні сервера для Javascript, воно дозволяє створювати все у бекенді, не змінюючи мову, наприклад, на Ruby on Rails. Це дуже круто, тому що це дозволяє комусь бути full-stack, використовуючи лише одну мову, оскільки більшість веб-сайтів уже використовують Javascript для свого інтерфейсу. Сам по собі Node.js є C++ програмою, яка отримує на вході JavaScript-код і виконує його.

NPM (аббр. node package manager) – це звичайний менеджер пакетів, що автоматично встановлюється разом з Node.js. Він використовується для завантаження пакетів з хмарного сервера npm. Пакетом у Node.js називається один або кілька JavaScript-файлів, які є якоюсь бібліотекою або інструментом.

SQL є найпоширенішою мовою для вилучення та організації даних, які зберігаються в реляційній базі даних. База даних - це таблиця, яка складається з рядків і стовпців. SQL — це мова баз даних. Це полегшує отримання конкретної інформації з баз даних, які в подальшому використовуються для аналізу. Навіть коли аналіз виконується на іншій платформі, наприклад Python або R, SQL знадобиться для вилучення потрібних даних з бази даних компанії.

Ця мова програмування має різноманітне застосування для аналітиків даних і фахівців із науки про дані. Це особливо корисно, оскільки може:

– Виконувати запити до бази даних.
– Отримувати дані з бази даних.
– Вставлятие записи в базу даних.
– Оновляти записи у базі даних.

– Видаляти записи з бази даних.

– Створювати нові бази даних або таблиці в базі даних.

– Налашовувати дозволів для таблиць, процедур та уявлень.

Express.js — це серверний фреймворк веб-додатків Node js, який спеціально розроблений для створення односторінкових, багатосторінкових і гібридних веб-додатків. Під капотом Express використовує ті самі вбудовані утиліти, що є у Node.js, але Express надає набір обробників, налаштувань та інших інструментів для покращення досвіду розробника та збільшення швидкості створення веб-серверів.

Більшість веб-серверів прослуховують запити, які надходять до сервера, отримують http-запити на кінцевій точці, та якимось чином відповідають на запит. Express.js має інструменти для виконання всіх цих кроків лише за кілька рядків коду.

Три великі концепції в Express.js:

– Маршрутизація.

– Проміжне програмне забезпечення.

– Запит/Відповідь.

Маршрутизація відноситься до того, як кінцеві точки програми відповідають на запити клієнта. Зазвичай це робиться за допомогою комбінації шаблону URL-адреси та пов’язаного методу HTTP.

Запит і відповідь – їх часто скорочують до «req» і «res» і означають запит, отриманий сервером, і відповідь, яка в кінцевому підсумку буде надіслана назад. Вони засновані на вбудованих об’єктах у Node.js, ClientRequest і ServerResponse.

Проміжне програмне забезпечення – це код, який виконується під час циклу запиту/відповіді. Зазвичай він використовується для додавання функціональних можливостей або покращення поведінки сервера. Функції проміжного програмного забезпечення мають доступ до об’єкта запиту req, об’єкта відповіді res і наступної функції в запиті-відповіді програми.

Функції проміжного програмного забезпечення можуть виконувати такі завдання:

– Виконувати будь-який код.

– Вносить зміни до запиту та об’єктів відповіді.

– Завершати цикл запиту-відповіді.

– Викликати наступне проміжне програмне забезпечення в стеку.

Node-Cron надає можливість автоматично повторювати завдання через певний проміжок часу. Можуть виникати повторювані завдання, такі як ведення журналів і створення резервних копій, які потрібно виконувати щодня, щотижня або щомісяця.

Скажімо, у вас є сценарій, який ви хочете запускати о 6 ранку щосереди. Ви можете створити другий сценарій, який запускається весь час і чекає до шостої ранку середи, перш ніж виконувати ваш сценарій.

Однак це не дуже зручно, особливо коли додається десятки або навіть сотні запланованих операцій. Cron вирішує це, будучи єдиним та остаточним планувальником завдань.

Bcrypt – ця бібліотека дозволяє хешувати та підмішувати сіль до паролю в Node.js з дуже невеликими зусиллями. Один із найкращих способів безпечного зберігання паролів – це добавляти до них сіль та хешувати. Соління та хешування перетворює простий пароль на унікальне значення, яке важко змінити.

Хешування пароля означає передачу простого текстового пароля через алгоритм хешування для створення унікального значення. Деякі приклади алгоритмів хешування: bcrypt, scrypt і SHA. Недоліком хешування є його передбачуваність. Кожен раз, коли ви передається той самий вхід до алгоритму хешування, він генеруватиме той самий вихід. Хакер, який має доступ до хешованого пароля, може перепроектувати шифрування, щоб отримати оригінальний пароль. Вони можуть використовувати такі методи, як атаки грубої сили або веселкові таблиці. Ось тут і йде засолювання.

Соління пароля додає випадковий рядок (сіль) до пароля перед його хешуванням. Таким чином, створений хеш завжди буде різним кожного разу. Навіть якщо хакер отримує хешований пароль, йому недоцільно виявити оригінальний пароль, який його згенерував.

## 2.4. Опис структури системи та алгоритмів її функціювання

Наявність хорошої відправної точки, коли справа доходить до архітектури проекту, є необхідною для життя самого проекту та того, як ми зможемо задовольняти мінливі потреби в майбутньому. Погана, брудна архітектура проекту часто призводить до:

– Нечитабельний і брудний код, який подовжує процес розробки.

– Безкорисне повторення, яке ускладнює підтримку та керування кодом.

– Труднощі із впровадженням нових функцій без зіпсування існуючого коду.

Основна мета будь-якої структури проекту:

– Чистий і читабельний код.

– Можливість писати і використовувати багаторазові фрагменти коду.

– Уникати повторень.

– Додавати нові функції, не порушуючи існуючого коду.

Термін архітектура програмного забезпечення розглядається як план побудови програмного забезпечення. Він представляє структуру програмного забезпечення, наприклад, як у ньому буде набір компонентів і як буде передаватися бізнес-логіка та логіка програми.

Існує багато різних архітектурних шаблонів, таких як клієнт-сервер, одноранговий, провідний-підпорядкований, MVC, канал-фільтр, класна дошка, багатошарова архітектура тощо. Кожен з них служить для різних цілей, але для саме цієї системи був обраний дуже популярний архітектурний шаблон під назвою MVC.

MVCS розшифровується як Model-View-Controller-Service — це архітектурний шаблон, який просто розділяє веб-додаток на чотири основні логічні частини. Логічними частинами є модель, представлення, контролер та сервис. Рівень представлення використовується для представлення частини презентації, що означає інтерфейс користувача. Рівень моделі займається реалізацією логіки бекенда, і він має немає зв’язку з шаром представлення. Шар

контролера відіграє цікаву роль, він просто встановлює зв’язок між моделлю та виглядом. Він також відомий як мозок програми в архітектурі MVC, оскільки він контролює, як будуть відображатися дані. Рівень сервісу можна інтерпретувати багатьма способами, але зазвичай він знаходиться там, де є основна логіка обробки бізнесу, і знаходиться нижче архітектури MVC, але вище архітектури доступу до даних.

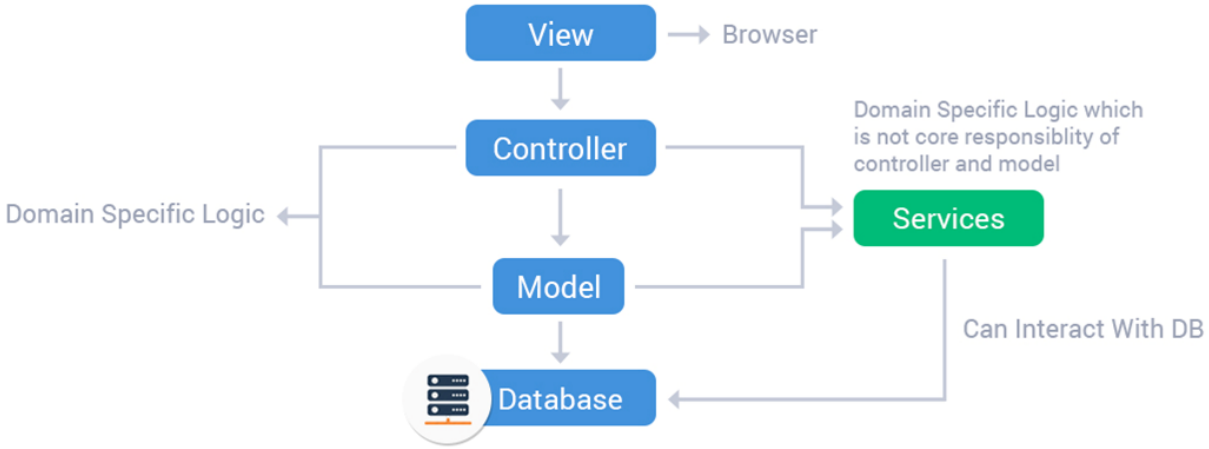


Рис. 2.5. Візуальне уявлення архітектури MVCS

На діаграмі, можна помітити, що контролер відіграє роль мозку програми, оскільки він встановлює зв’язок між шаром представлення та шаром моделі.

Хоча архітектурний шаблон MVSC є одним із найстаріших шаблонів, він все ще популярний для розробки сучасних веб-додатків. Існує багато позитивних причин для використання цього шаблону у веб-додатках:

– Він може обробляти та організовувати веб-додатки великого розміру.

– Він підтримує асинхронний виклик методу, тобто можно писати бізнес-логіку асинхронно.

– Його можна легко змінити.

– Процес розробки йде швидше.

– Він підтримує розробку, керовану тестуванням (TDD).

– У цій структурі обслуговування легко.

– Пошукова оптимізація (SEO).

– Забезпечує чітке розділення проблем (SoC).

Структура серверної частини проекту з архітектурою MVCS наведена на рис. 2.6.

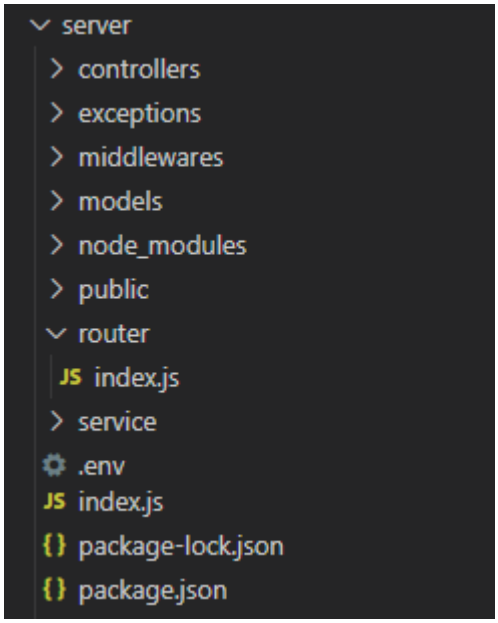


Рис. 2.6. Структура файлів серверної частини

В папці controllers (рис. 2.7) знаходятся контроллери системи. Контролери отримують інформацію від користувача, обробляють її, валідують, та надсилають далі до сервісів. Папка містить в собі наступні наступні файли:

– battle-controller.js;
– chat-controller.js;
– events-controller.js;
– items-controller.js;
– location-controller.js;
– shop-controller.js;
– user-controller.js;

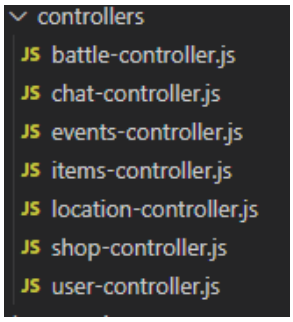


Рис. 2.7. Зміст папки controllers

В папці exceptions знаходится лишне один файл api-error.js (рис. 2.8). Це власний універсальний класс помилки. Він розширює можливості звичайного js error. Написаний обробчик має жоливість відправляти HTTP статус код помилки наприклад, повідомлення та масив з помилками.

```
module.exports = class ApiError extends Error {
    status;
    errors;

    constructor(status, message, errors = []) {
        super(message);
        this.status = status;
        this.errors = errors;
    }

    static UnauthorizedError() {
        return new ApiError(401,"Пользователь не авторизован
    }

    static ForbiddenError() {
        return new ApiError(403,"Доступ запрещён")
    }

    static BadRequest(message, errors = []) {
        return new ApiError(400, message, errors)
    }
}
```

Рис. 2.8. Зміст файлу api-error.js

В папці middlewares (рис. 2.9) знаходится два файли: auth-middleware.js та error-middleware.js.

– error-middleware – обробник помилок власного класу помилки api-error.

– auth-middleware – обробник запросів. Кожного разу коли користувач робить запрос на сервер до захищеного енд-поінту, де потрібні права авторизованого користувача. Він перевіряє accessToken та AuthorizationHeader, якщо його немає або він невалідний то у відповідь сервер надсилає відповідну помилку.

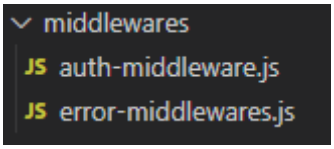


Рис. 2.9. Зміст папки middlewares

В папці models (рис. 2.10) знаходится три файли:

– query-model.js – основний файл зв’язку з базою данних, та надсилання запросів.

– token-model.js – файл відповідальний занадсилання запросів що стосуються токенів.

– user-model.js – файл відповідальний занадсилання запросів що стосуються информації про користувачів.

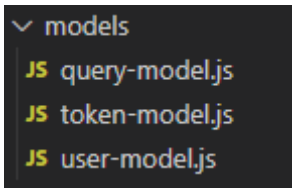


Рис. 2.10. Зміст папки models

Папка node-modules використовується для збереження всіх завантажених пакетів з npm для проекту.

Папка public (рис. 2.11) використовується для зберігання публічний файлів таких як css стилі та зображень.

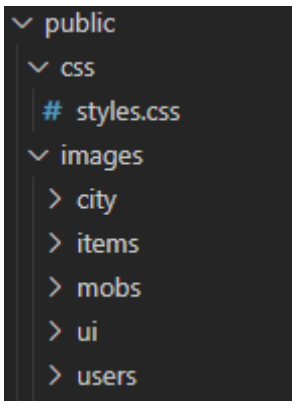


Рис. 2.11. Зміст папки public

Папка router має в собі один файл index.js (рис. 2.12). Він відповідальний за енд-поінти, інакше кажучи, він отримує запроси та розподіляє їх на потрібні контролери. Якщо цей енд-поінт доступний лише авторизованому користувачу, то він спочатку передає запрос на обробник авторизації, і якщо помилок немає, то він передає запрос беспосередньо до відповідного контролеру.

Також в цьому файлі запущений cron scheduler на кожні 20 секунд. Кожні 20 секунд запускається код який перевірє гравців, у яких кількість очків здоров'я меньше за максимальне, і кожному з них відновлює певну її кількість.

```
router.post('/login',
    body('password').isLength({min: 6,max: 30}),
    UserController.login
);
router.post('/register',
    body('email').isEmail(),
    body('password').isLength({min: 6,max: 30}),
    UserController.registration
);
router.get('/logout',   UserController.logout);
router.get('/refresh',  UserController.refresh);
router.get('/revive',  authMiddleware, UserController.revive);
router.get('/getUsers', authMiddleware, UserController.getUsers);
router.get('/getRatings', authMiddleware, UserController.getRatings);
router.get('/chat', authMiddleware, ChatController.getMessages);
router.post('/message', authMiddleware, ChatController.sendMessage);
```

Рис. 2.12. Фрагмент коду файлу index.js

В папці service (рис. 2.13) знаходяться файли сервісів, відповідні за певний напрямок. Моделі отримують інформацію від контроллерів, обробляють її та направляють інформацію далі до моделей.

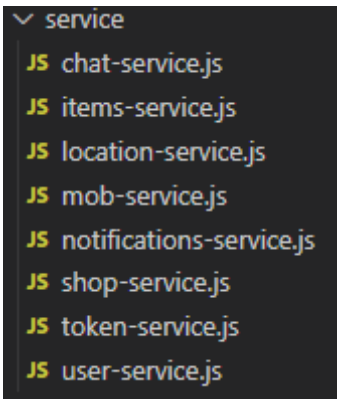


Рис. 2.13. Зміст папки service

Також в корні проекту є декілька файлів (рис. 2.14). Там знаходяться наступні файли:

– .env – це файл де зберігаються змінні середовища та значення конфігурації, такі як: порт на якому буде працювати сервер, назва бази даних та логін і пароль від неї, секретні ключі для шифрування.

– index.js – загальний функціональний файл, у якому відбувається монтування та запуск додатку.

– package.json – цей файл використовується як маніфест у екосистеми NodeJS, що зберігає інформацію про додаток, модулі та пакети встановлені за допомогою NPM.

– package-lock.json – використовується у NodeJS для відстежування точної версії кожного встановленого пакета, щоб продукт був на 100% відтворений таким же чином, навіть якщо пакунки оновлюються їхніми розробниками.

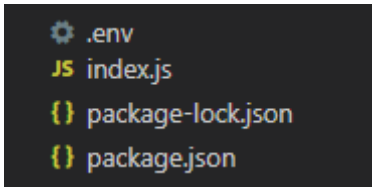


Рис. 2.14. Зміст папки service

В данній системі використовується реляційна база даних. Реляційна база даних — це тип бази даних, яка зберігає та організовує пов’язані точки даних. Дані організовано в таблиці, які пов’язані на основі спільних даних. Вони є найпоширенішим типом баз даних, які сьогодні використовуються.

Струкутра бази даних проекту (рис. 2.15) має наступний вигляд:

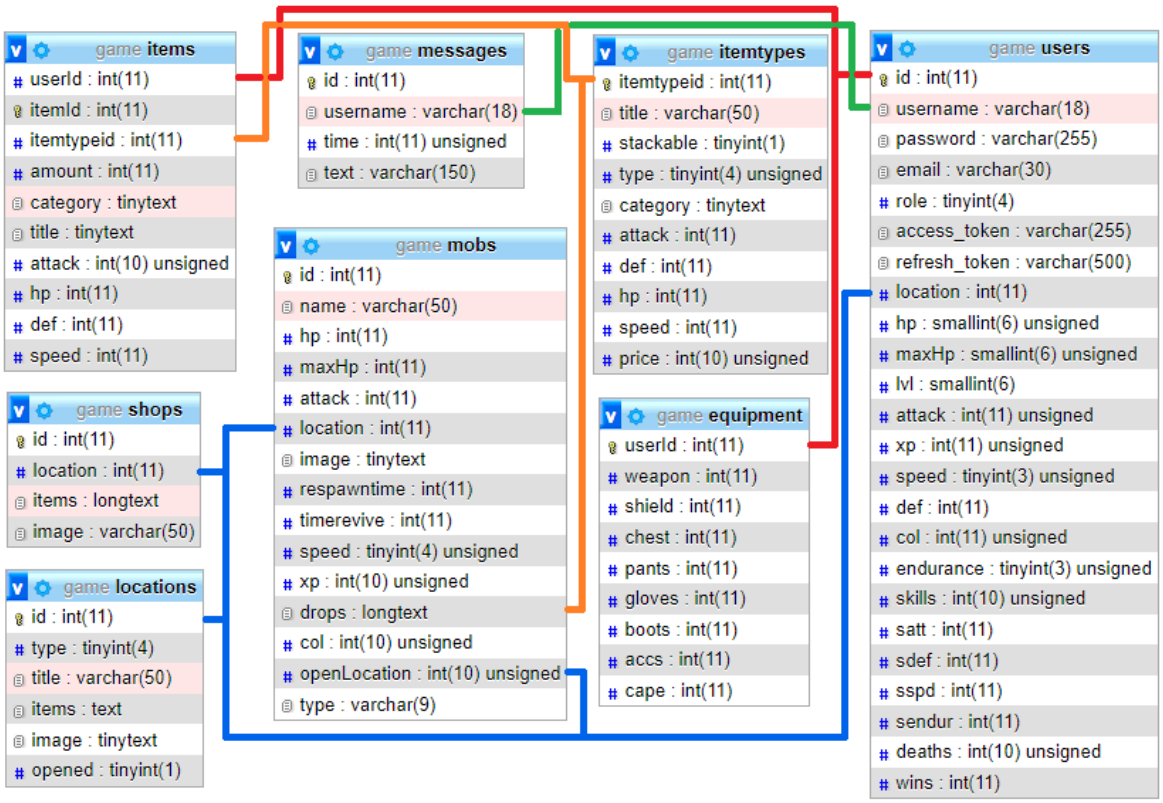


Рис. 2.15. Структура бази-даних

Подібна база даних дозволяє здійснювати пошук в одній або кількох таблицях за допомогою одного запиту. Реляційні бази даних ідеально підходять для комплексного аналізу даних і операцій. У нереляційній базі даних таблиці можуть ділитися тими ж даними, але вони не можуть «зв'язатися» одна з одною. З реляційною базою даних вони можуть.

Структура клієнтської частини проекту (рис. 2.16) має в собі три папки:

– node_modules – використовується для збереження всіх завантажених пакетів з npm для проекту. Ця папка містить усі залежності та підзалежності, зазначені в package.json, які використовуються програмою React. Він містить більше 800 підпапок.

– public – папка містить статичні файли, такі як index.html, файли бібліотеки javascript, зображення заголовку а також файл маніфесту що містить в собі конфігурацію додатка, також, ці файли не буде обробляти webpack.

– src – Ця папка є серцем програми React, оскільки містить JavaScript, який потрібно обробляти webpack-ом. У цій папці знаходиться основний компонент App.js, пов'язані з ним стилі (App.css), index.js та його стиль (index.css); який забезпечує точку входу в додаток.

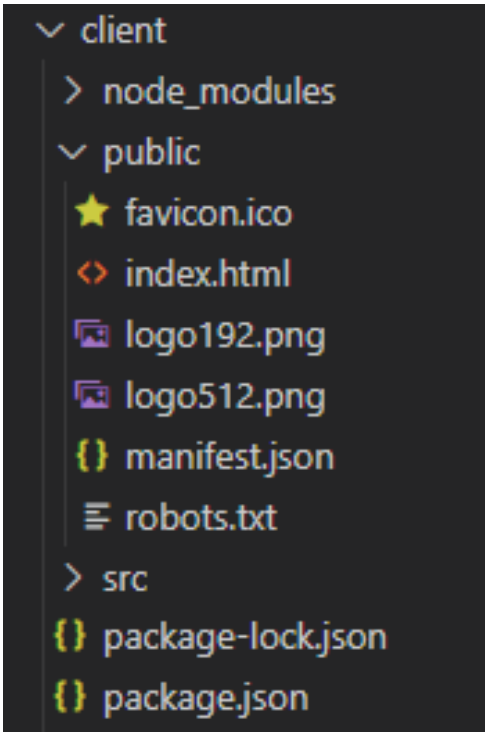


Рис. 2.16. Структура файлів клієнтської частини

В папці src (рис. 2.17) знаходится:

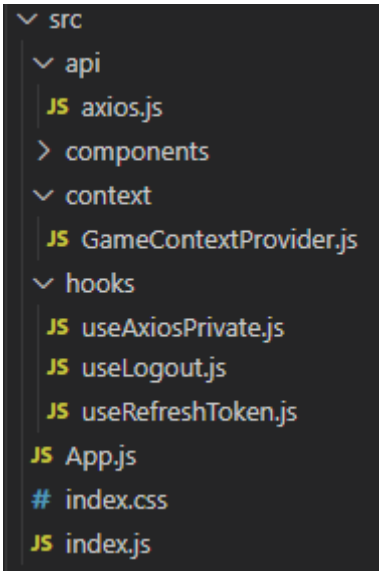


Рис. 2.17. Зміст папки src

– Папка api має один файл axios.js. В ньому підключається та налаштовується бібліотека Axios, яка використовується для віправки HTTP-запитів до серверу.

– Папка components (рис. 2.18) містить в собі усі компоненти додатку.

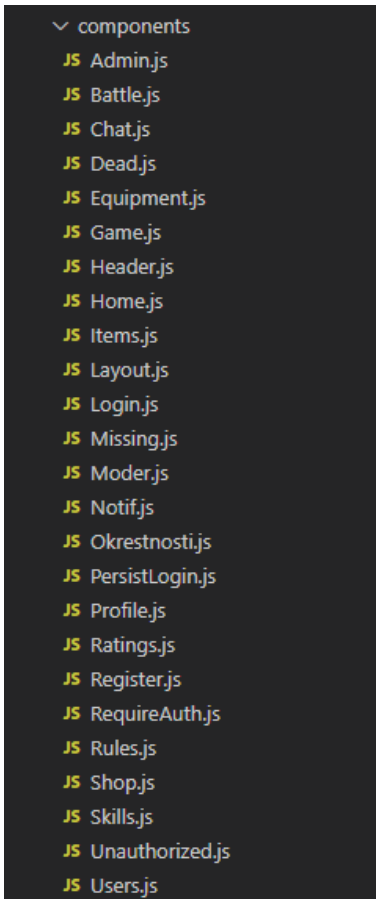


Рис. 2.18. Зміст папки components

– Папка context містить в собі файл GameContextProvider.js. Цей файл створює та надає контекст компонентам. Контекст React надає дані компонентам незалежно від того, наскільки глибоко вони знаходяться в дереві компонентів. Контекст використовується для керування глобальними даними, наприклад дані про авторизацію.

– Папка hooks містить в собі три файли. Перший – useAxiosPrivate.js використовується для надсилання HTTP запросів до серверу разом з даними про аутентифікацію а саме надсилає refreshToken авторизації у httponly cookies а також надсилає accessToken у заголовку запиту. Другий – useLogout.js використовується для виходу з акаунту – надсилається запит до серверу, а також з клієнта видаляються дані про авторизацію. Третій – useRefreshToken.js використовується коли у accessToken-а закінчився термін дії, ций хук надсилає refreshToken до серверу та отримує у відповідь новий accessToken.

– Файл App.js має в собі кореневий компонент програми react, тому що кожне представлення та компонент обробляються за ієрархією в React, де <App /> є найвищим компонентом в ієрархії. Ций файл не є небхідним, але, це підтримує стійку ієрархію у коді. У ньому налаштовуються усі можливі вузли додатку.

– Файл index.css – містить с собі усі глобальні стилі додатку.

– Файл index.js є фактичною точкою входу для всіх програм вузла та містить логіку візуалізації компоненту основної програми та надає йому всі необхідні параметри.

В корні проекту зберігаються два вже нам знайомих файли, це – package.json, файл маніфесту і package-lock.json – манфіест версій кожного інстальованого пакета.

Загальний алгоритм роботи веб-додатку поділяється на декілька етапів:

1. Завантаження необхідних бібліотек за допомогою NPM.

2. Бібліотека React об’єднує всі файли JS і CSS у папці статичної збірки, а потім монтує їх в єдиний основний файл html.

3. Сервер підлючається до бази даних.

4. Кліент підключається до SSE для отримання нової інформації в режимі реального часу.

5. React монтує нові сторінки веб-додатку, коли це необхідно.

6. Кліент реагує на усі повідомлення та події які надходять з серверу, і виконує необхідні дії.

Перш за все, щоб згенерувати файл програми - збірку, яка міститиме код всіх файлів, нам треба виконати іншу команду npm run build. Ця команда створить у проекті папку public, а в ній - новий файл bundle.js. Це саме той файл, який підключатиметься на веб-сторінку.

Тобто буде згенеровано головний JavaScript-файл "bundle.js". Саме цей файл містить код, компільований з React-синтаксису JSX у стандартний JS, який підтримується більшістю веб-браузерів. У цьому файлі передбачені різні API-запити, за допомогою яких клієнтський код отримує або модифікує дані віддаленого джерела. Ми можемо просто кинути веб-сторінку в веб-браузер, і додаток також буде працювати. Після цього для розгортання системи використовується команда npm start.

Розглянемо деякі ключові елементи функціювання додатку.

Спочатку при заході на кліент, додаток перевіряє, чи був користувач авторизований до цього і чи була при попередному вході нажата галочка «Не виходити з системи» на сторінці входу (рис. 2.19).

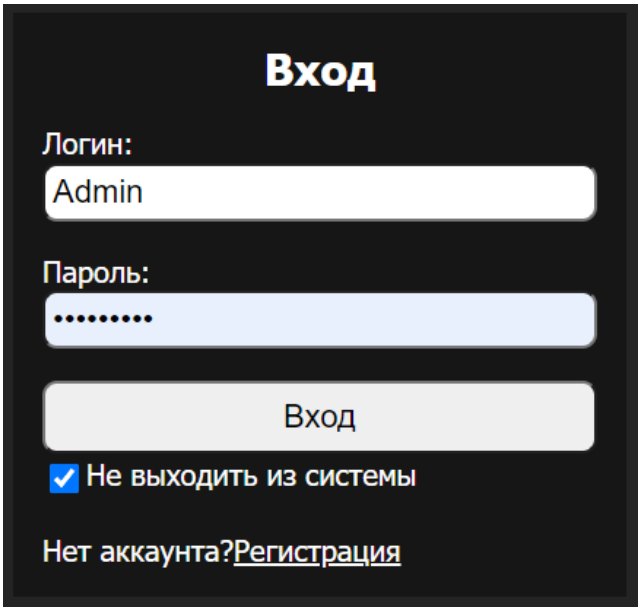


Рис. 2.19. Сторінка входу

Якщо галочка була нажата і авторизація була успішна, то додаток зберігає у локальному сховищі одна змінну «persist», яка відповідає за автоматичний вхід при наступному візиті користувача. Задля безпеки, в ній зберігається лише значення true, а при виході ця змінна видаляється.

При авторизації користувач отримує два JWT токени access та refresh. Кожен токен має свій час життя, це робиться для безпеки. Наприклад якщо зловмисник вкраде його, він зможе ним зкористуватися їм лише отведений час, а після цього, доступ йому буде закритий. В даній роботі access-токен використовується для доступу до системи, зберігається в пам’яті і живе – один день. Refresh-токен використовується для оновлення access токену, живе – 30 днів, але зберігається в httpOnly cookie. Флаг httpOnly є дуже важливою умовою безпеки, щоб її неможливо були змінити JavaScript-ом, тобто cookies встановлює лише сервер.

Якщо в access-токеі закінчився термін життя, сервер поверне статус код 401 – користувач не авторизован. В такому випадку клієнт надішле новий запит на оновлення access токену, в cookies запросу вже находиться refresh-токен. Якщо refresh-токен є валідним, то сервер надсилає нову пару токенів.

При переході на основну сторінку кліент надсилає запит на сервер до /api/game, та надає відповідь у форматі json, сервер бере з бази даних поточний id локації в якій знаходиться користувач, а потім знаходить вміст локації по її id. Приклад json відповіді на цей запит зображено на рис. 2.20.


```
{
  "links":[
    {
      "id":2,
      "title":"Железный дворец"
    }
      ...
    {
      "id":3,
      "type":"shop",
      "title":"Целебный магазин"
    }
  ],
  "title":"Town of Begginings",
  "image":"city/town1.png",
  "id":1
}
```


Рис. 2.20. Відповідь сервера у форматі JSON

Отримавши відповідь відображає назву, зображеня, та інші локації, які доступні для подальшого переміщення. Відповідь з рис. 2.20 клієнт відобразить наступним чином (рис. 2.21).

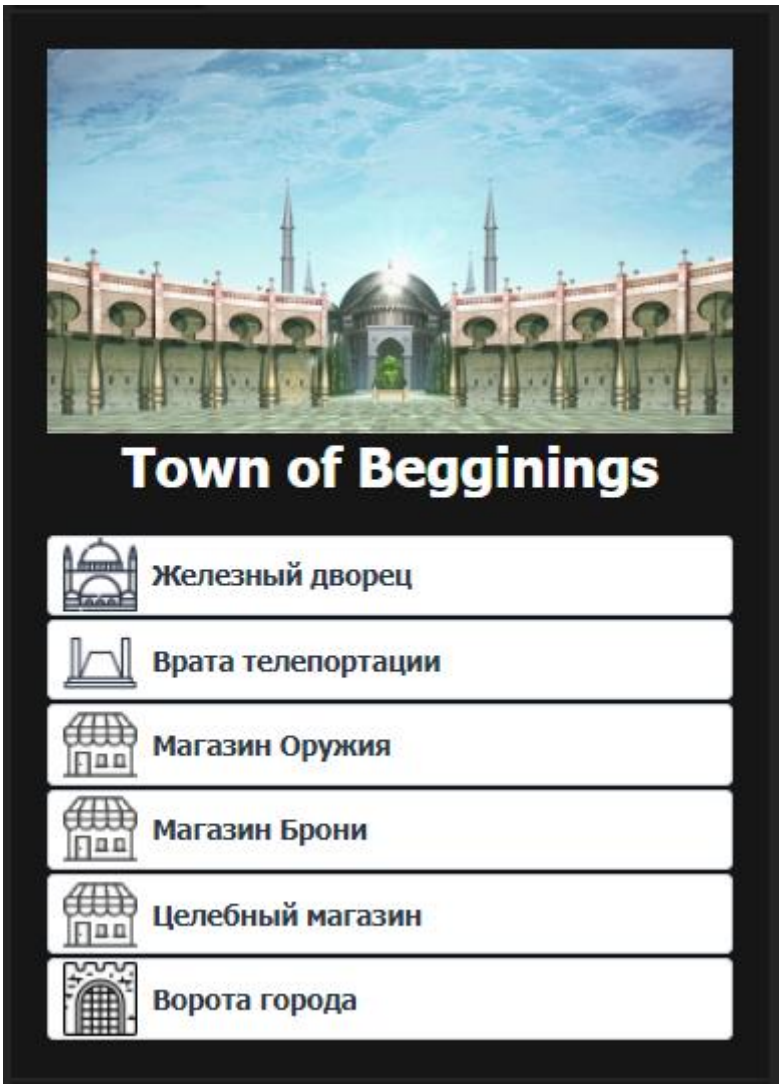


Рис. 2.21. Відображення JSON відповіді від серверу

## 2.5. Обґрунтування та організація вхідних та вихідних даних програми

Вхідними та вихідними є дані, що безпосередньо стосуються розробленої системи та інформації про сутності, що на в цій системі знаходяться. Інформація про сутності, такі як, локації, предмети, монстри можуть мати наприклад такі атрибути: унікальний код, назва або їм’я, зображення, поточна вартість предмету, дата відновлення монстра, атрибути, такі як сила або захист.

Якщо розглядати розроблену систему систему в глобальному сенсі як інформаційну систему, то всі вхідні та вихідні дані передаються у форматі JSON.

У якості вхідних даних веб-додаток приймати дані користувача, які він може надати на сторінцкаї додатку. У якості вихідних даних користувач отримує інформацію щодо стану сутностей.

## 2.6. Опис розробленої системи

### 2.6.1. Використані технічні засоби

При розробці даного продукту використовувався персональний комп'ютер з наступними характеристиками:

- Процесор AMD A6-7310 2.00 GHz.
- Операційна система Windows 10 LTSC.
- ОЗУ 4000Мб.
- Відеоадаптер: AMD Radeon R4 GRraphics.
- Накопичувач 500Гб.

### 2.6.2. Використані програмні засоби

У цьому проекті для керування реляційними базами була обрана найбільш широко використовувана система MySQL.

MySQL забезпечує багатокористувацький доступ до одразу декількох баз даних. Використання клієнтів або інструментів керування MySQL позбавляє від стресу та труднощів використання командного рядка MySQL. Під час виконання завдань адміністрування MySQL більшість розробників покладається на інструменти керування вмістом для MySQL, такі як phpMyAdmin, Adminer, SQLBuddy тощо. Ці інструменти використовуються для керування вмістом баз даних MySQL. Для цього проекту була обрана СУБД phpMyAdmin.

PhpMyAdmin – це інструмент адміністрування MySQL який доступний безкоштовно. В основному він написаний на PHP як портативний веб-додаток, і він став одним із найпопулярніших інструментів адміністрування MySQL в Інтернеті. Він підтримує широкий спектр операцій на MySQL і MariaDB. PhpMyAdmin забезпечує як інтерфейс користувача, так і пряме виконання

інструкцій MySQL. Часто використовувані операції, такі як керування базами даних, таблицями, стовпцями тощо, можна виконувати через інтерфейс користувача, при цьому ви все ще можете безпосередньо виконувати будь-які інструкції SQL. Інтерфейс phpMyAdmin зображено на рис.2.22.

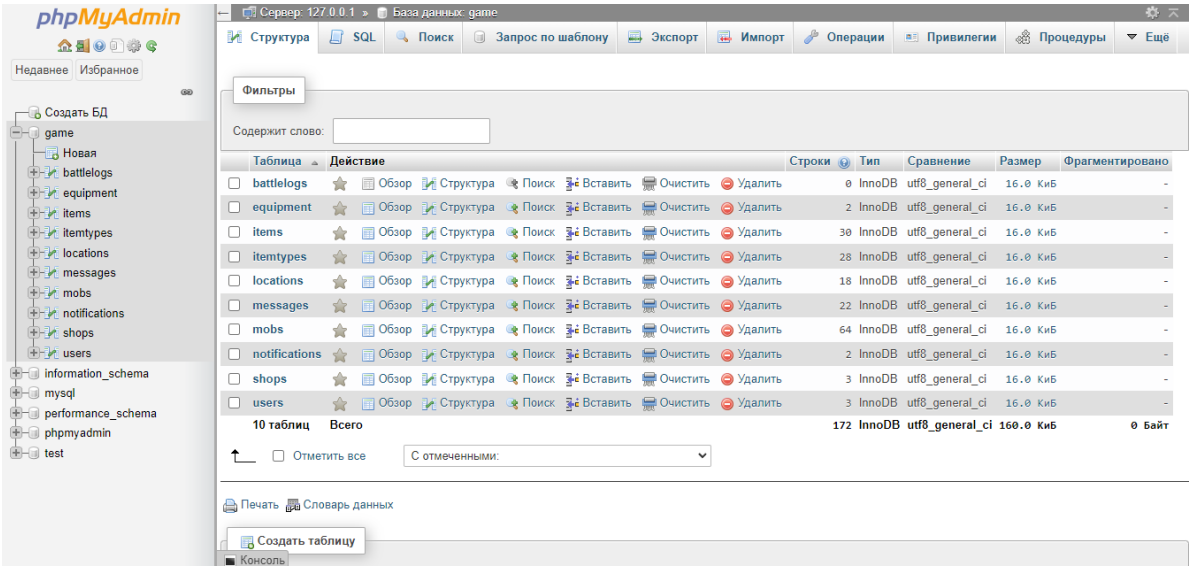


Рис. 2.22. Інтерфейс СУБД phpMyAdmin

У якості середовища програмування було обрано Visual Stduio Code (рис. 2.23). Geany - середовище розробки програмного забезпечення, написана з використанням бібліотеки GTK +. Доступна для наступних операційних систем: BSD, Linux, Mac OS X, Solaris і Windows. Geany поширюється згідно GNU General Public License.

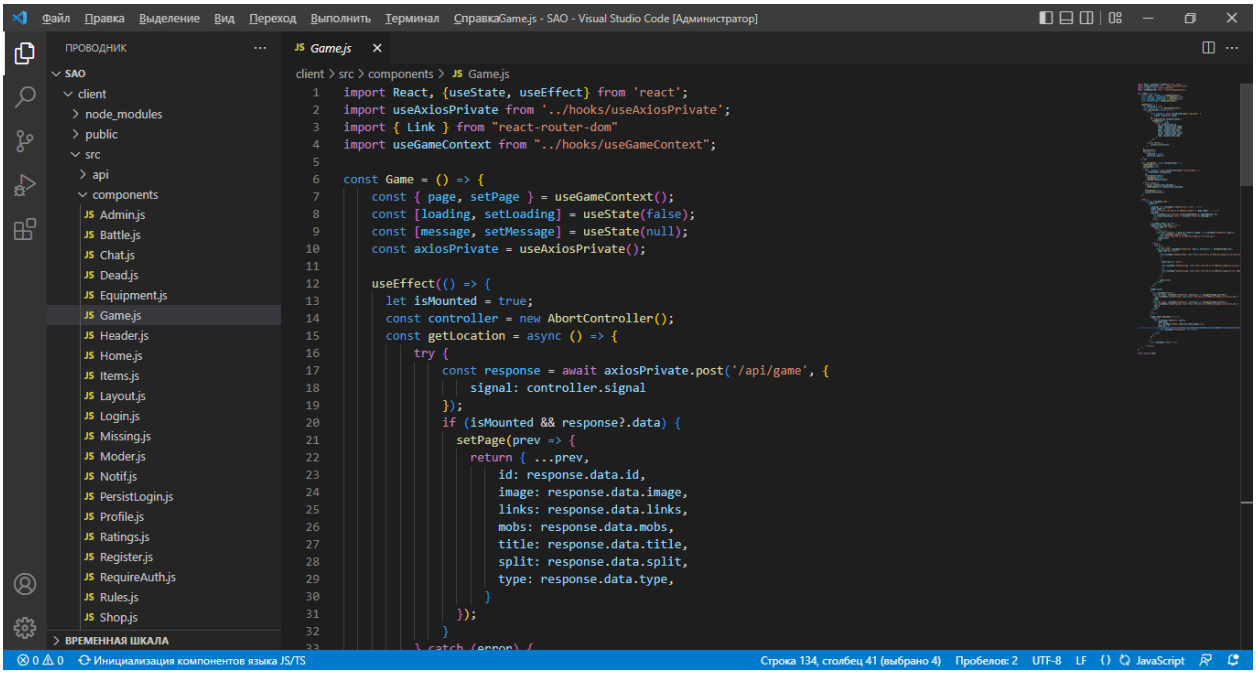


Рис. 2.23 Інтерфейс Visual Stduio Code

Microsoft Visual Studio Code — це безкоштовний, потужний і легкий редактор коду для Windows, macOS та Linux. Заснований на відкритому програмному забезпеченні, він легко налаштовується з понад 25 000 розширень для кожного розробника та кожної мови програмування.

Visual Studio Code поєднує в собі простоту редактора коду з тим, що потрібно розробникам для їх основного циклу розробки програмного забезпечення. Він забезпечує повну підтримку редагування коду, навігацію та розуміння, а також багату модель розширюваності та легку інтеграцію з існуючими інструментами.

JavaScript використовує Node Package Manager, часто скорочено як npm, як менеджер пакетів і репозиторій пакетів. Маючи понад мільйон пакетів, розміщених на веб-сайті npm, розробники можуть шукати та переглядати величезний каталог бібліотек JavaScript. Деякі з цих пакетів завантажуються понад 10 мільйонів разів на тиждень. Веб-сайт містить інформацію щодо всіх пакетів, розміщених на ньому, а саме: вихідний код, документацію, номер версії та розмір розпакованого.

Поряд з веб-сайтом, npm також надає інструмент командного рядка, який дозволяє розробникам встановлювати або видаляти ці пакунки. Інструмент командного рядка npm робить встановлення пакетів у проекти Node.js надзвичайно простим за допомогою однорядкової команди «npm install <назва пакету>».

### 2.6.3. Виклик та завантаження програми

Розроблений веб-додаток завантажується і запускається автоматично при заході на сайт. Для стабільної роботи веб-додатку, необхідна лище наявність одного з сучасних браузерів, таких як Google Chrome, Firefox, Opera. Для мобільних платформ, а саме IOS, Android підійде навіть вбудований мобільний браузер.

### 2.6.4. Опис інтерфейсу користувача

Після відкриття застосунку користувач потрапляє на домашню сторінку, яка зображена на рис. 2.24.

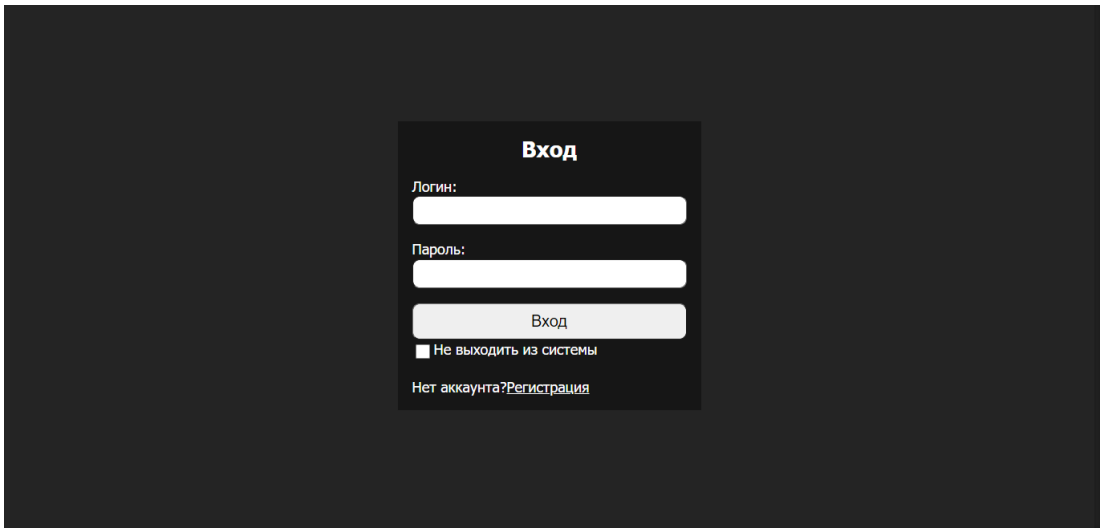


Рис. 2.24. Домашня сторінка застосунку

На домашній сторінці користувач може ввести логін та пароль щоб авторизуватись. При бажанні користувач може увімкнути галку «Не выходить из системы», якщо вона була увімкнута, то при наступному заході користувачеві не потрібно буде заново вводити пароль.

Якщо у користувача немає аккаунта, то він може натиснути на посилання реєстрації, інтерфейс сторінки реєстрації зображено на рис. 2.25.

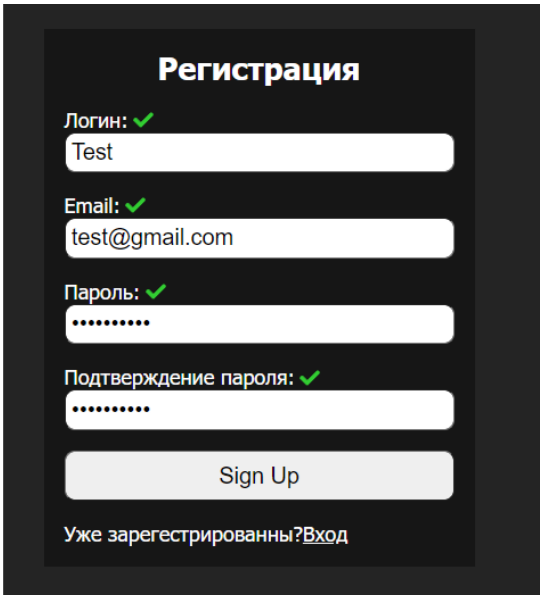


Рис. 2.25. Сторінка реєстрації застосунку

Якщо користувач ввів некоректні дані, то форма реєстрації має підказки до полів, а також індикатор. Якщо дані коректні, то над кожним з полів з’явиться зелена галка і кнопка реєстрації стане доступна, інакше, якщо дані некоректні – з’явиться червона галка і кнопка реєстрації становиться недоступною, приклад зображений на рис 2.26.

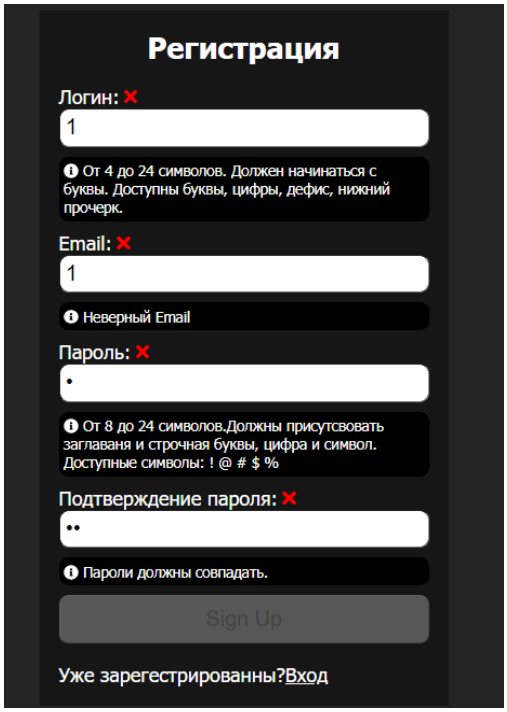


Рис. 2.26. Сторінка реєстрації з невалідними даними

Якщо користувач ввів усі дані вірно і натиснув кнопку реєстрації, і на сервері усе пройде без помилок, то у користувача з’явиться повідомлення про успішно реєстрацію (рис 2.27).

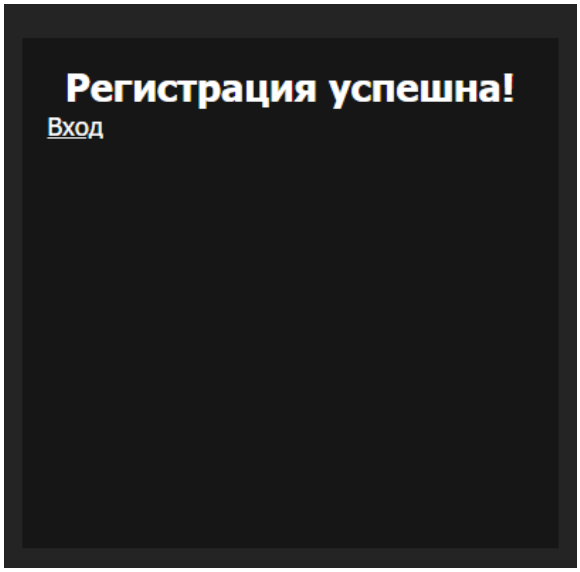


Рис. 2.27 Повідомлення про успішну реєстрацію

Якщо дані було введено коректні то користувач потрапить до головного меню (рис 2.28).

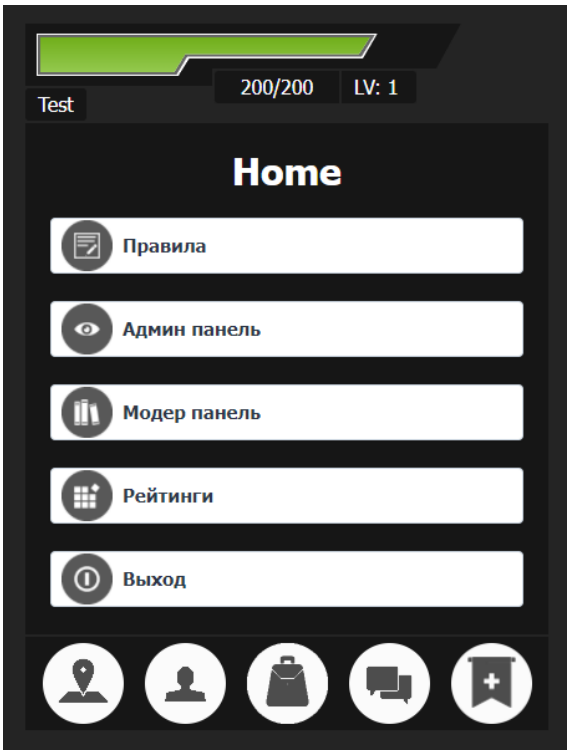


Рис. 2.28. Головне меню

На усіх сторінках додатку є так звана «шапка» сайту яка зображена на рис 2.29 і рис. 2.30, в якій є полоска що відображає кількість здоров'я що залишилася у процентах. Також у ній є ім'я персонажу, кількість здоров'я у цифрах, максимальна кількість здоров'я а також поточний рівень персонажу.

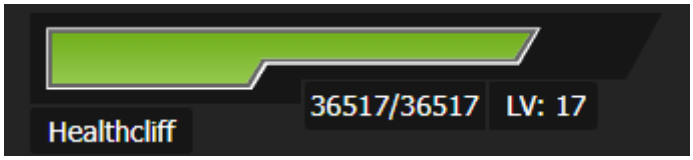


Рис. 2.29. Шапка

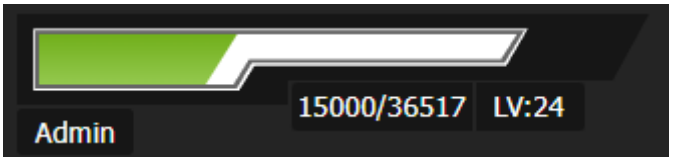


Рис. 2.30. Шапка (продовження)

Також на всіх сторінках є «підвал» (рис 2.31), через яке користувач може переходити на основні сторінки гри. Там знаходяться переходи до наступних сторіок: локація, профіль користувача, інвантар, чат, головне меню.

Рис. 2.31. Підвал

У головному меню (рис 2.28) знаходяться правила гри, адмін та модер панель, рейтинги і кнопка виходу. Щоб зайти до адмін або модер панелі треба щоб користувач володів необхідними привілеями. У разі якщо їх не буде, то користувачеві буде показано повідомлення яке зображене на рис. 2.32.

На сторінці рейтингів зображено 3 категорії: топ гравців по кількості перемог, топ гравців по уровню, і топ найбагатших граців. Кожен топ відібражає лише 3 кращих гравця у даній категорії. Сторінка рейтингів зображена на рис. 2.33.

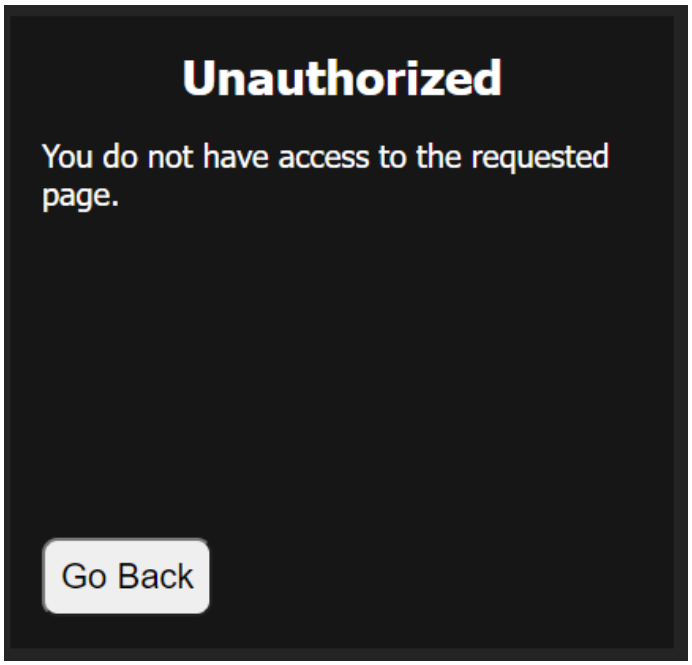


Рис. 2.32. Повідомлення за відсутності привілеїв

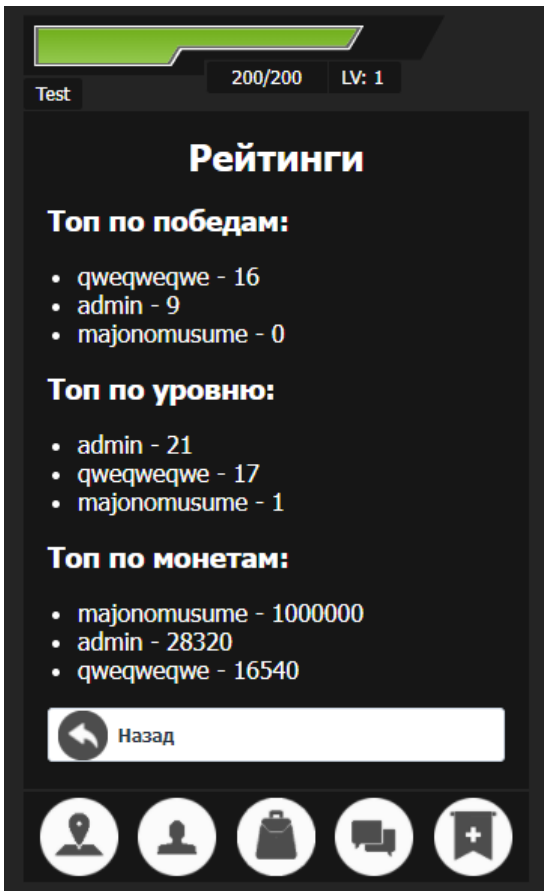


Рис. 2.33. Сторінка рейтингів

На сторінці локації користувачеві відображається поточна локація, її назва, зображення, магазини та інші локаціїї які доступні для перміщення. Приклади сторінок зображено на рис. 2.34 та рис. 2.35.


Рис. 2.34. Сторінка локації


Рис. 2.35. Сторінка локації

Локації поділені умовно на дві частини: околиці та міста. Головною відмінністю околіць від міста є тим що, в околицях мешкають монстри, з якими треба боротися, в околиці можна потрапити через ворота. На рис. 2.36 зображено околиці стартового міста.


Рис. 2.36. Локація околиць міста

Щоб перейти до поєдинку з монстром, треба натиснути кнопку «В бой», і відкриється сторінка поєдинку (рис. 2.37). На цій сторінці є кнопка атаки та втечі. Якщо втекти, то користувача поверне до локації. При атаці, сервер прораховує отриманий та нанесений урон, після цього воно записується у логах та відображається користувачеві. Також динамічно змінюється смужка здоров'я користувача і монстра.

При перемозі користувачеві відображається повідомлення про перемогу. Також відображається кількість отриманих очків досвіду (рис. 2.38).

У разі поразки користувачеві відкривається сторінка смерті (рис. 2.39) , де він може воскресити свого персонажа.

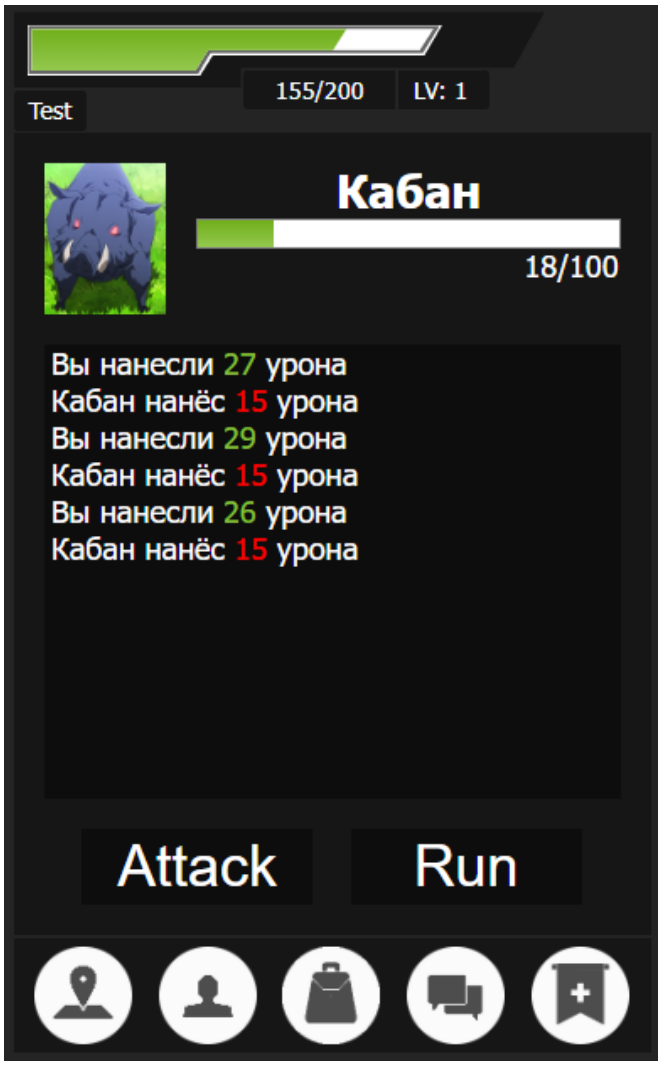


Рис. 2.37. Сторінка поєдинку

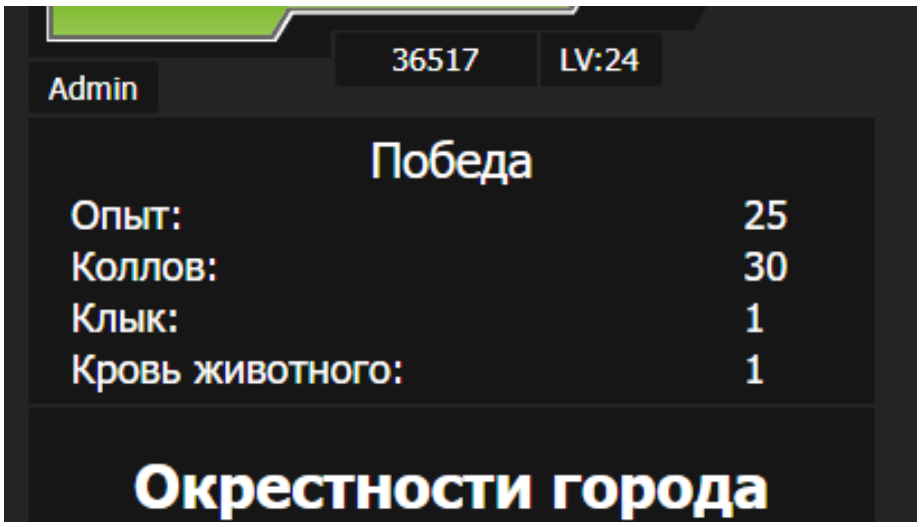


Рис. 2.38 Повідомлення при перемозі


Рис. 2.39 Повідомлення при смерті

Сторінка профілю (рис 2.40) містить в собі інформацію про персонажа. Його ім’я, аватар, рівень, поточна кількість очок досвіду, кількість очків навичок та грошей. А також тут відображаються характеристики персонажу: сила, захист, швидкість та витривалість. На цій же сторінці можна перейти до вкладки інвентарю, навичок та екіпірування.


Рис. 2.40. Сторінка профілю

На сторіні навичок (рис 2.41) гравець може почитати опис навичок та подивитись поточний рівень навичок. Якщо гравцю доступні нові навички, які даються за підвищення рівня персонажу, то він може їх розподілити. Розподілення відбувається нажмаючи на кнопки «+» або «-».

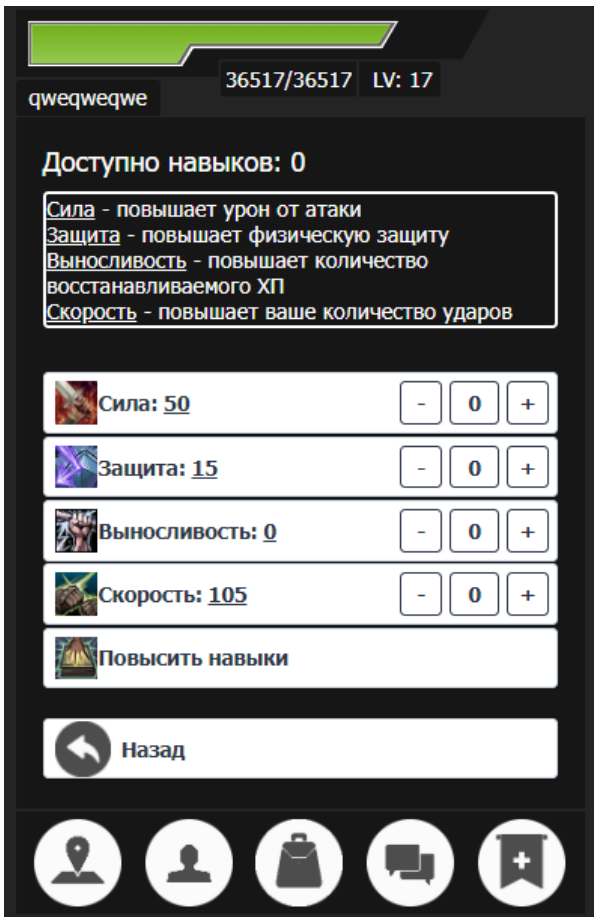


Рис. 2.41. Сторінка навичок

Сторінка екіпірування (рис. 2.42) відображає поточні предмети які екіпіровані на персонажі. При натисканні на предмет буде відображено характеристики які дає предмет і кнопка для зняття предмету (рис. 2.43).

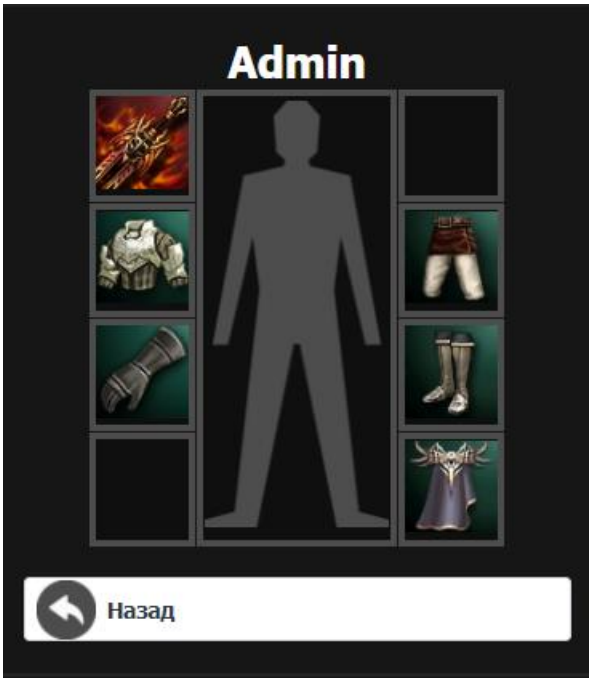


Рис. 2.42. Сторінка екіпірування

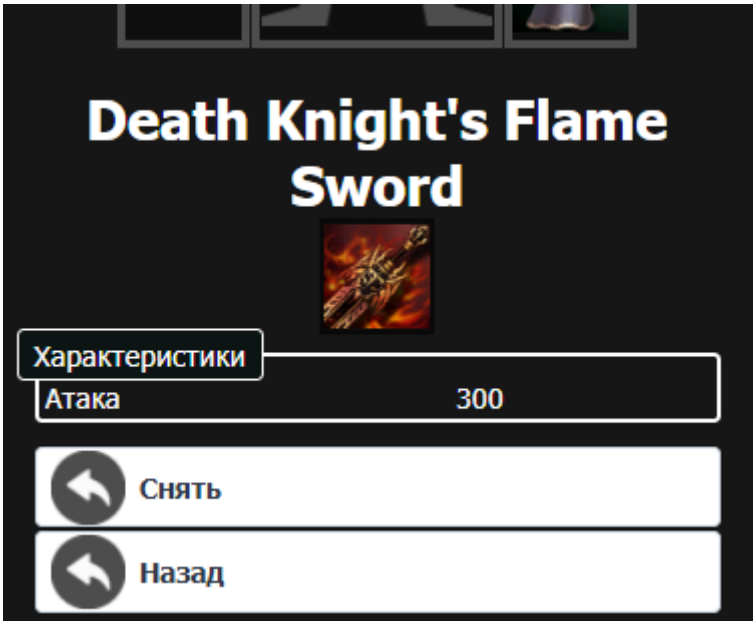

Рис. 2.43. Відображення обраного предмету

На сторінці інвентарю (рис. 2.44) зображено усі можливі категорії предметів. При натисканні на певну категорію буде відображено сторінку із списком предметів даної категорії (рис. 2.45). При натисканні на предмет буде зображено його характеристики, назва і зображеня (рис. 2.46).

Рис. 2.44. Сторінка інвентарю

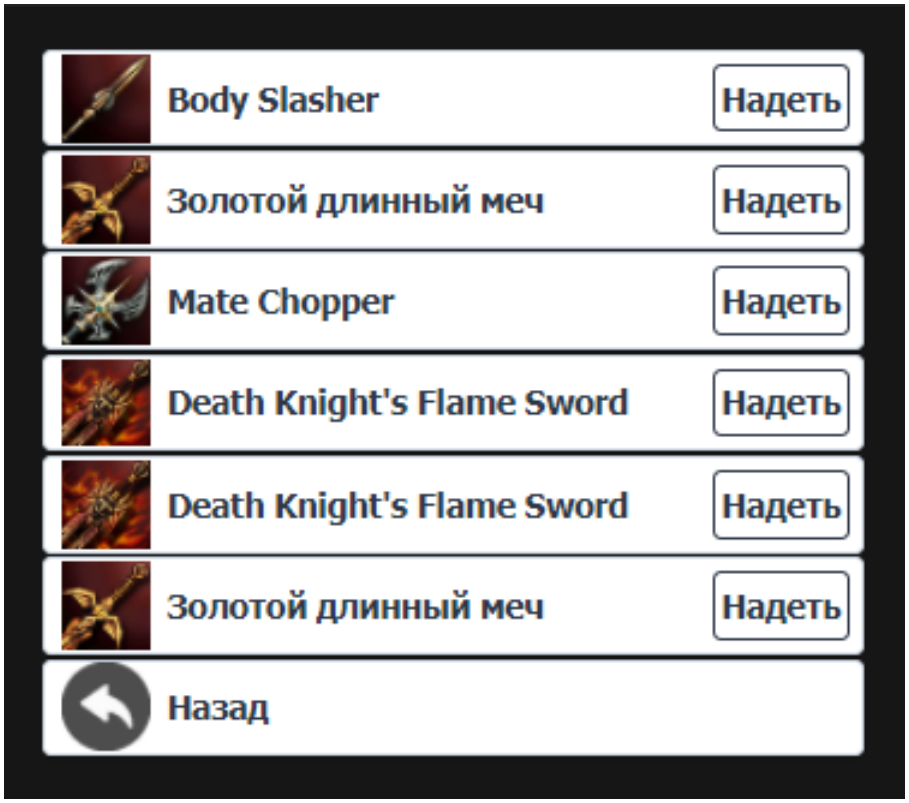


Рис. 2.45. Список предметів обраної категорії

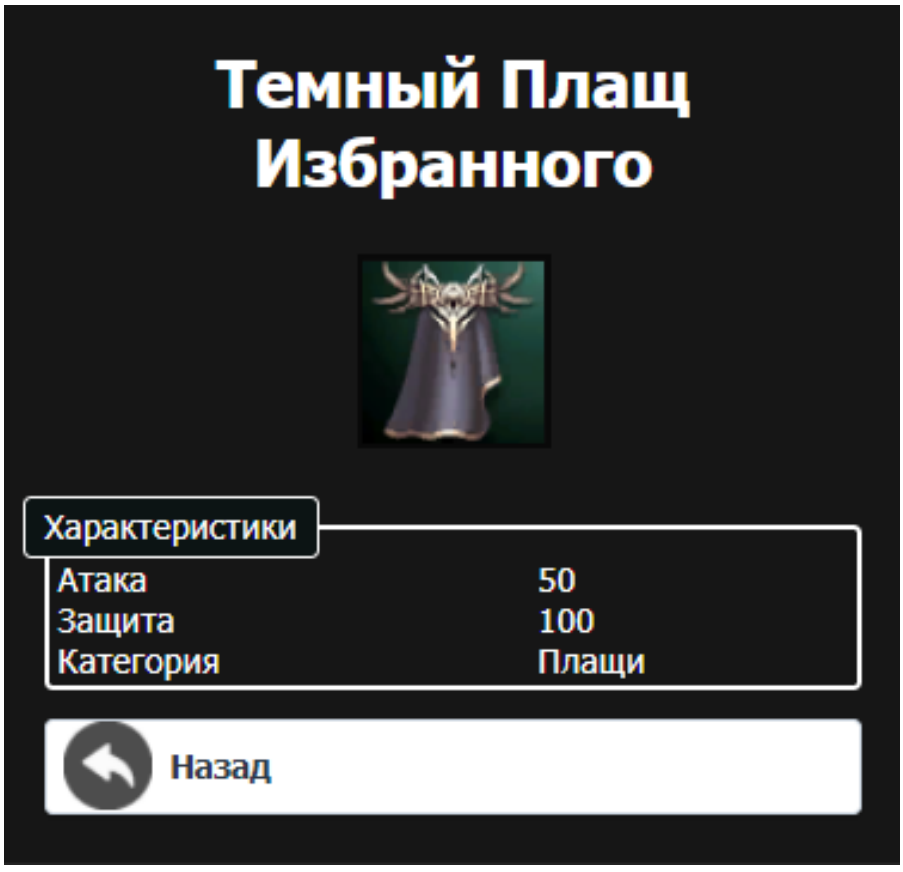


Рис. 2.46. Сторінка опису предмету

Також в грі присутній чат (рис. 2.47). В якому зображено останні сто повідомлень. При відправці повідомлення воно автоматично надсилається іншим користувачам. Тобто не потрібно оновлювати сторінку бо повідомлення відображаються у режимі реального часу.

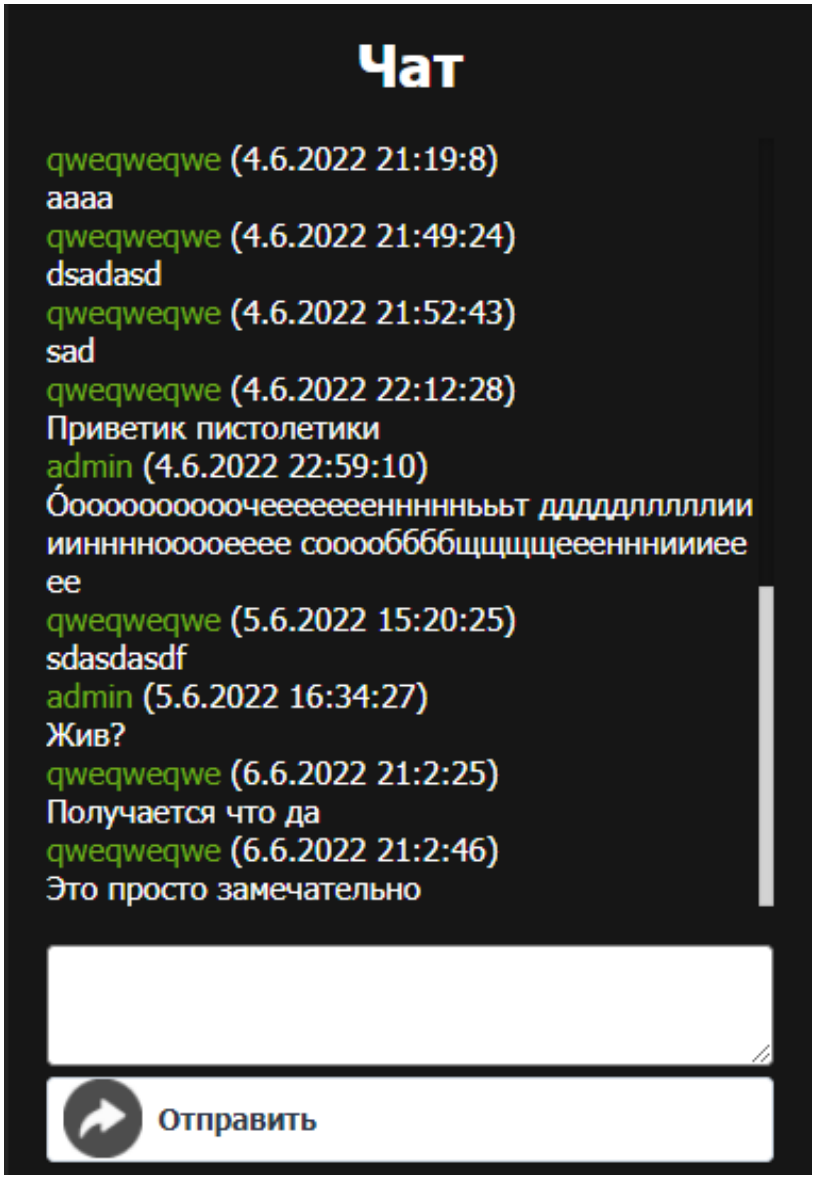

Рис. 2.47. Сторінка чату

# РОЗДІЛ 3
# ЕКОНОМІЧНИЙ РОЗДІЛ

## 3.1. Розрахунок трудомісткості та вартості розробки програмного продукту

Початкові дані:

1. передбачуване число операторів програми - 1940;
2. коефіцієнт складності програми – 1,7;
3. коефіцієнт корекції програми в ході її розробки – 0,06;
4. годинна заробітна плата програміста – 140 грн/год;
5. коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі – 1,2;
6. коефіцієнт кваліфікації програміста, обумовлений від стажу роботи з даної спеціальності – 1,1;
7. вартість машино-години ЕОМ – 14 грн/год

Нормування праці в процесі створення ПЗ істотно ускладнено в силу творчого характеру праці програміста. Тому трудомісткість розробки ПЗ може бути розрахована на основі системи моделей з різною точністю оцінки.

Трудомісткість розробки ПЗ можна розрахувати за формулою:

$$t = t_o + t_u + t_a + t_n + t_{отл} + t_д, \text{ людино-годин,} \qquad (3.1)$$

де $t_o$ – витрати праці на підготовку й опис поставленої задачі (приймається 50);

$t_u$– витрати праці на дослідження алгоритму рішення задачі;

$t_a$– витрати праці на розробку блок-схеми алгоритму;

$t_п$– витрати праці на програмування по готовій блок-схемі;

$t_{отл}$– витрати праці на налагодження програми на ЕОМ;

$t_д$– витрати праці на підготовку документації.

Складові витрати праці визначаються через умовне число операторів у ПЗ, яке розробляється.

Умовне число операторів (підпрограм):

Q=q·C·(1+p), де (3.2)

*q*– передбачуване число операторів;

*C*– коефіцієнт складності програми;

*p*– коефіцієнт кореляції програми в ході її розробки.

Q= 1940·1,7·(1+0,06) = 3495,98

Витрати праці на вивчення опису задачі $t_u$ визначається з урахуванням уточнення опису і кваліфікації програміста:

$$t_u = \frac{Q \bullet B}{(75 \ldots 85) \bullet K}, \text{ людино-годин,} \quad (3.3)$$

де *B* – коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі;

*K* – коефіцієнт кваліфікації програміста, обумовлений стажем роботи з даної спеціальності;

$$t_u = \frac{3495{,}98 \cdot 1{,}2}{78 \cdot 1{,}1} = 48{,}89, \text{ людино-годин.}$$

Витрати праці на розробку алгоритму рішення задачі:

$$t_a = \frac{Q}{(20 \ldots 25) \cdot k}, \text{ людино-годин.} \quad (3.4)$$

де *Q* – умовне число операторів програми;

*k* – коефіцієнт кваліфікації програміста.

Підставивши відповідні значення в формулу (3.4), людино-годин:

$$t_a = \frac{3495{,}98}{25 \cdot 1{,}1} = 127{,}12, \text{ людино-годин.}$$

Витрати на складання програми по готовій блок-схемі:

$$t_n = \frac{Q}{(20 \ldots 25) \cdot k}, \text{ людино-годин.} \quad (3.5)$$

$$t_n = \frac{3495{,}98}{25 \cdot 1{,}2} = 116{,}53, \text{ людино-годин.}$$

Витрати праці на налагодження програми на ЕОМ:

- за умови автономного налагодження одного завдання:

$$t_{отл} = \frac{Q}{(4..5) \cdot k}, \text{ людино-годин.} \quad (3.6)$$

$$t_{отл} = \frac{3495{,}98}{5 \cdot 1{,}1} = 635{,}63, \text{ людино-годин,}$$

- за умови комплексного налагодження завдання:

$$t_{\text{отл}}^{\text{к}} = 1{,}2 \cdot t_{\text{отл}}; \quad (3.7)$$

$$t_{\text{отл}}^{\text{к}} = 1{,}2 \cdot 635{,}63 = 762{,}75, \text{ людино-годин}$$

Витрати праці на підготовку документації:

$$t_{\partial} = t_{\partial p} + t_{\partial o}; \quad (3.8)$$

де $t_{др}$ – трудомісткість підготовки матеріалів і рукопису

$$t_{\partial} = \frac{Q}{(15 \ldots 20) \cdot K}; \quad (3.9)$$

$$t_{\partial} = \frac{3495{,}98}{18 \cdot 1{,}2} = 161{,}84, \text{ людино-годин.}$$

$t_{до}$– трудомісткість редагування, печатки й оформлення документації

$$t_{до} = 0{,}75 \cdot t_{\text{др}}; \quad (3.10)$$

$$t_{до} = 0{,}75 \cdot 161{,}84 = 121{,}38, \text{ людино-годин.}$$

$$t_{\partial} = 161{,}84 + 121{,}38 = 283{,}22, \text{ людино-годин.}$$

Отримаємо трудомісткість розробки програмного забезпечення:

t =50+48,89+127,12+116,53+635,63+283,22=1261,39, людино-годин.

У результаті ми розрахували, що в загальній складності необхідно 1261,39 людино-годин для розробки даного програмного забезпечення.

## 3.2. Розрахунок витрат на створення програми

Витрати на створення ПЗ Кпо включають витрати на заробітну плату виконавця програми Зз/п і витрат машинного часу, необхідного на налагодження програми на ЕОМ.

$$К_{ПО} = З_{ЗП} + З_{МВ}, \text{ грн,} \quad (3.11)$$

де $З_{ЗП}$ – заробітна плата виконавців, яка визначається за формулою:

$$З_{ЗП} = t \cdot С_{ПР}, \text{ грн,} \quad (3.12)$$

де $t$ – загальна трудомісткість, людино-годин;

$С_{ПР}$– середня годинна заробітна плата програміста, грн/година

$$З_{ЗП} = 1261{,}39 \cdot 150 = 176\,594{,}6, \text{ грн.}$$

$З_{МВ}$– Вартість машинного часу, необхідного для налагодження програми на ЕОМ:

$$З_{МВ} = t_{отл} \cdot C_{М}\text{, грн,} \qquad (3.13)$$

де $t_{отл}$ – трудомісткість налагодження програми на ЕОМ, год.

$C_{МЧ}$– вартість машино-години ЕОМ, грн/год.

$$З_{МВ} = 768{,}15 \cdot 14 = 17659{,}46\text{, грн.}$$

$$К_{ПО} = 176594{,}6 + 17659{,}46 = 194\,254{,}06\text{, грн.}$$

Очікуваний період створення ПЗ:

$$T = \frac{t}{B_k \cdot F_p}\text{, мес.} \qquad (3.14)$$

де $B_k$- число виконавців;

$F_р$– місячний фонд робочого часу (при 40 годинному робочому тижні Fр=176 годин).

$$T = \frac{1261{,}39}{1 \cdot 176} = 7{,}71 \text{ міс.}$$

**Висновки**. На розробку даного програмного забезпечення піде 1261,39 людино-годин. Тобто, ймовірна очікувана тривалість розробки складатиме 7,71 місяці при стандартному 40-годинному робочому тижні і 176-годинному робочому місяці. Очікувані витрати на створення програмного забезпечення складатимуть 194 254,06 грн.

# ВИСНОВКИ

В даній кваліфікаційній роботі була розроблена клієнт-серверна WEB гра що складається з двух окремих повноцінних додатків у вигляді серверної частини реалізованної на платформі NodeJS, і кліентської частини, реалізованій на сучасній, популярній та швидкій біліотеці – React.

Основними задачами серверної частини э:

– надсилати код кліентської частини для подальшої взаємодії з сервером;

– при запиті від користувача, робити запит до бази даних, обробляти її да надавати належну відповідь у форматі JSON;

– мати налаштований SSE end-поінт, а також тримати актуальну інформацію щодо підключених користувачів на даний момент;

– при зміні ігрових даних, надислати цю інформацию користувачам, належним до цих самих данних;

– мати код, який легко масштабується, щоб потім «наповнювати» гру контентом.

Основними задачами клієнтської частини є:

– мати адаптивний та зручний інтерфейс;

– тримати keep-alive – з'єднання з серверним SSE end-поінтом;

– надавати запрос до серверу у належному форматі;

– корректно обробляти інформацію отриману як зі звичайних запросів, так від SSE.

Продукт повністю сумісний з більшістю браузерів, з десктопними та мобільними пристроями на усіх популярних платформах, тобто продукт є цілком адаптивним під мобільні пристрої. Додаток надає можливість користувачеві зареєструватися та надійно зберігає та шифрує інформацію, яка була надана користувачем. Кінцевий працездатний продукт готовий для розширення і наповнення контентом.

На началі користувачеві потрібно зареєструватися, обрати собі унікальне імя персонажу та пароль. Після реєстрації користувач попадає на стартову

локацію, і йому треба вирушати в околиці міста, де йому належить розвивати свого персонажа. Робиться це шляхом бороьби з монстрами і рейдовими боссами. У разі перемоги користувач буде отрмувати деяку кількість очків досвіду, а при накопиченні певної його кількості, він буде повишати свій рівень.

З кожним підвищенням рівня, користувач буде отримувати очки навичок, які користувач розподіляє меж параметрами персонажа, такі як: сила, захист, швидкість та витріваліть. Проте з кожним рівнем необхідна кількість очків досвіду росте, тож для більш ефективного розвитку персонажу, йому треба вирушати на наступний рівень, де мешкають більш сильніші монстри, за перемогу над котрими, користувачеві будуть надавати більшу кількість очків досвіду.

Практичне значення полягає в реалізації сучасними засобами прототипу браузерної гри, яка призначена для розважальних цілей.

Також у кваліфікаційній роботі було визначено трудомісткість розробленого програмного продукту (1261,39 люд-год), проведений підрахунок вартості роботи по створенню програми (194 254,06 грн) та розраховано час на його створення (7,71 місяців).

## СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ


1. Wieruch R. The road to react: your journey to master plain yet pragmatic react.js. Independently published, 2018. 226 p.

2. Banks A., Porcello E. Learning React: Modern Patterns for Developing React Apps 2nd Edition, 2020. 310 p.

3. Frain B. Responsive Web Design with HTML5 and CSS: Develop futureproof responsive websites using the latest HTML5 and CSS techniques, 3rd Edition, 2020. 408 p.

4. Carlos S. R. React 17 Design Patterns and Best Practices: Design, build, and deploy production-ready web applications using industry-standard practices, 3rd Edition, 2021. 394 p.

5. Nandaa A. Beginning API Development with Node.js: Build highly scalable, developer-friendly APIs for the modern web with JavaScript and Node.js, 2018. 254 p.

6. Bugl D. Learn React Hooks: Build and refactor modern React.js applications using Hooks, 2019. 774 p.

7. Flangan D. JavaScript: The Definitive Guide: Master the World's Most-Used Programming Language, 2020. 708 p.

8. Freeman A. Pro react 16. Apress, 2019. 745 p.

9. O'Reily. Head First JavaScript Programming: A Brain Friendly Guide, 2014.

10. Офіційний документація середовища розробки Visual Studio Code URL: https://code.visualstudio.com.

11. Офіційна документація React.JS URL : https://ru.reactjs.org/docs/getting-started.html.

12. Mikowski M., Powell J. Single Page Web Applications: JavaScript end-to-end, 2013. 432 p.

13. Emmit A. Scott, Jr. SPA Design and Architecture: Understanding single-page web applications, 2015. 275 p.

14. Klauzinski P., Moore J. Mastering JavaScript Single Page Application Development, 2016. 452 p.

15. Reed M. SQL: 2 Books in 1 - The Ultimate Beginner & Intermediate Guides to Mastering SQL Programming Quickly (Computer Programming), 2022. 427 p.

16. Reed M. SQL: The Ultimate Expert Guide to Learn SQL Programming Step-by-Step (Computer Programming), 2022. 120 p.

17. Madden N. API Security in Action, 2020. 576 p.

18. Elrom E. React and Libraries: Your Complete Guide to the React Ecosystem, 2021. 529 p.

19. Hossain M. CORS in Action: Creating and consuming cross-origin APIs, 2014. 240 p.

20. Hahn E. Express in Action: Writing, building, and testing Node.js applications, 2016. 256 p.

## ДОДАТОК А

# КОД ПРОГРАМИ

**Index.js**

```
const express = require('express')
const cors = require("cors")
const cookieParser = require("cookie-parser")
const mysql = require("mysql2")
const router = require("./router/index")
const errorMiddleware = require('./middlewares/error-middlewares');

const app = express()
const port = process.env.PORT || 5000

const dbHost = "localhost"
const dbName = "game";
const dbUser = "root";
const dbPass = "root";

app.use(express.json());
app.use(cookieParser());
app.use(cors({
   credentials: true,
   origin: "http://192.168.31.31:3000"
}));
app.use('/api', router);
app.use(errorMiddleware);

const start = async () => {
   try {
      await mysql.createConnection({ host: dbHost, user: dbUser, password: dbPass, database:
dbName});
      app.listen(port, () => console.log(`Running on port = ${port}`))
   } catch (e) {
      console.log(e);
   }
}
start();
```

**battle-controller.js**

```
const LocationService = require("../service/location-service");
const MobService = require("../service/mob-service");
const UserService = require("../service/user-service");
const ApiError = require('../exceptions/api-error');
const NotifService = require('../service/notifications-service');
const ItemService = require('../service/items-service');
const {validationResult} = require('express-validator');
const EventsController = require('./events-controller');

class CombatController {
```

```
    async getBattle (req, res, next) {
        const errors = validationResult(req);
        if (!errors.isEmpty()) {
            return next(ApiError.BadRequest('Не верные параметры', errors.array()));
        }
        if(!Number.isInteger(req.body?.id)) {
            return next(ApiError.BadRequest('Не верные параметры', errors.array()));
        }
        try {
            const mobInfo = await MobService.getMob(req.body?.id);
            return res.json(mobInfo);
        } catch (e) {
            next(e);
        }
    }

    async attack (req, res, next) {
        const errors = validationResult(req);
        if (!errors.isEmpty()) {
            return next(ApiError.BadRequest('Не верные параметры', errors.array()));
        }
        if(!Number.isInteger(req.body?.mobId)) {
            return next(ApiError.BadRequest('Не верные параметры', errors.array()));
        }
        try {
            let mobInfo = await MobService.getMob(req.body?.mobId);
            if(mobInfo.hp <= 0) {
                return next(ApiError.BadRequest('Моб уже мёртв', errors.array()));
            }
            const userInfo = await UserService.getUserGameInfo(req.user.id);
            if(mobInfo.location != userInfo.location) {
                return next(ApiError.BadRequest('Не верная локация', errors.array()));
            }
            const usersToNotify = await LocationService.getUsersInLocation(userInfo.location,
req.user.id);
            const userDamage = userInfo.attack + Math.floor(Math.random() * (userInfo.attack/5));
            mobInfo.hp -= userDamage;
            const response = {};
            response['mob'] = mobInfo;
            const mobDamage = Math.ceil(mobInfo.attack*(100/(100+userInfo.def)));
            if(mobInfo.hp > 0) {
                const successMob = await MobService.setMobHp(req.body?.mobId, mobInfo.hp);
                const successUser = await UserService.setUserHp(req.user.id, userInfo.hp -
mobDamage);
                if(successMob && successUser) {
                    // Не отсылаем пользователям частоту воскрешения
                    delete response.mob.respawnTime;
                    // Обновлённого моба отсылаем другим пользователям
                    await EventsController.sendEventsToUsers(usersToNotify, response);
                    // К обновлённому мобу добавляем нанесённый и полученный урон пользователя
                    response['log'] = {"user": userDamage, "enemy":mobDamage};
                    return res.json(response);
```

```
            } else {
                return next(ApiError.BadRequest('Произошла ошибка', errors.array()));
            }
        } else {
            mobInfo.hp = 0;
            response['mob'] = mobInfo;
            const successMob = await MobService.killMob(mobInfo);
            const successUser = await UserService.setUserHp(req.user.id, userInfo.hp -
mobDamage);
            const successXP = await UserService.addXP(req.user.id, mobInfo.xp);
            const successCol = await UserService.addCol(req.user.id, mobInfo.col);
            const successKills = await UserService.addKill(req.user.id);

            if(successMob && successXP && successCol && successUser && successKills) {
                const possbileDrops = JSON.parse(mobInfo.drops);
                // Сортируем по id из-за особенностей sql.
                // select ... IN (1,3,2) вернёт в порядке (1,2,3),
                // Что изменит порядок и предметы создадутся не правильно
                possbileDrops.sort((a, b) => (+(a["id"]) - ((b["id"]))));
                let dropIDarray = [];
                let dropAmountarray = [];
                possbileDrops.forEach(drop => {
                    if(Math.random() <= drop.chanse) {
                        dropIDarray.push(drop.id);
                        dropAmountarray.push(Math.floor(Math.random() * (drop.max - drop.min + 1) +
drop.min));
                    }
                });
                if(dropIDarray.length) {
                    const droptitles = await ItemService.addItems(req.user.id, dropIDarray,
dropAmountarray);
                    const drop = [];
                    droptitles.forEach(function(item, i) {
                        drop.push({title: droptitles[i], amount: dropAmountarray[i]} );
                    });
                    await EventsController.sendEventToUser(req.user.id, {drop: {drop, exp:
mobInfo.xp, col: mobInfo.col}});
                } else {
                    // Отсылаем только опыт
                    await EventsController.sendEventToUser(req.user.id, {drop: {drop:[], exp:
mobInfo.xp, col: mobInfo.col}});
                }
                if(mobInfo.type == "boss") {
                    if(mobInfo.openLocation > 0) {
                        await LocationService.openLocation(mobInfo.openLocation);

                    }
                }
                // Не отсылаем пользователям частоту воскрешения
                delete response.mob.respawnTime;
                // Обновлённого моба отсылаем другим пользователям
                EventsController.sendEventsToUsers(usersToNotify, response);
```

```
                // К обновлённому мобу добавляем нанесённый и полученный урон пользователя
                response['log'] = {"user": userDamage, "enemy":mobDamage};
                return res.json(response);
            } else {
                return next(ApiError.BadRequest('Произошла ошибка', errors.array()));
            }
        }
    } catch (e) {
        next(e);
    }
  }
}
module.exports = new CombatController();
```

**chat-controller.js**

```
const ChatService = require('../service/chat-service');
const ApiError = require('../exceptions/api-error');
const UserService = require('../service/user-service');
const {validationResult} = require('express-validator');

class ChatController {
  async getMessages (req, res, next) {
    try {
      const messages = await ChatService.getMessages();
      if(messages) return res.json(messages);
    } catch (e) {
      next(e);
    }
  }

  async sendMessage (req, res, next) {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return next(ApiError.BadRequest('Не верные параметры', errors.array()));
    }
    const text = req.body?.message;
    if(!text) {
      return next(ApiError.BadRequest('Сообщение отсутсвует', errors.array()));
    }
    try {
      const userName = await UserService.getUserName(req.user.id);
      const success = await ChatService.sendMessage(userName, text);
      if(success) {
        return res.json(true);
      } else {
        return next(ApiError.BadRequest('Произошла ошибка'));
      }
    } catch (e) {
      next(e);
    }
  }
}
```

```
module.exports = new ChatController();
```

**events-controller.js**

```
const {validationResult} = require('express-validator');
var clients = [];

class EventsController {
   async eventsHandler(request, response, next) {
      const clientId = request.user.id;
      const prevConnection = clients.find(client => client.id == clientId)
      if(prevConnection) {
         prevConnection.response.end((`data: "Forbidden"\n\n`));
      }
      const headers = {
         'Content-Type': 'text/event-stream',
         'Connection': 'keep-alive',
         'Cache-Control': 'no-cache'
      };
      response.writeHead(200, headers);
      const newClient = {
         id: clientId,
         response
      };
      clients.push(newClient);
      console.log(`${clientId} Opened connection`);

      request.on('close', () => {
         console.log(`${clientId} Connection closed`);
         clients = clients.filter(client => client.id !== clientId);
      });

      setInterval(() => response.write(`hb\n\n`),20000);
   }

   async sendEventsToAll(data) {
      clients.forEach(client => client.response.write(`data: ${data}\n\n`))
   }

   async sendEventsToUsers(users, data) {
      users.forEach(userId => {
         clients.forEach(client => {
            if(userId === client.id) {
               client.response.write(`data: ${JSON.stringify(data)}\n\n`);
            }
         })
      })
   }

   async sendEventToUser(userId, data) {
      try{
         const client = clients.find(user => user.id == userId);
         if(client) client.response.write(`data: ${JSON.stringify(data)}\n\n`);
```

```
        } catch (err) {
            console.log(err);
        }
    }
}
module.exports = new EventsController();
```

**items-controller.js**

```
const ItemsService = require("../service/items-service");
const ApiError = require('../exceptions/api-error');
const {validationResult} = require('express-validator');
const UserService = require("../service/user-service");

class ItemsController {
    async getItems (req, res, next) {
        const errors = validationResult(req);
        if (!errors.isEmpty()) {
            return next(ApiError.BadRequest('Не верные параметры', errors.array()));
        }
        if(!req.body?.type instanceof String) {
            return next(ApiError.BadRequest('Не верные параметры', errors.array()));
        }
        try {
            const items = await ItemsService.getItems(req.user.id, req.body?.type);
            if(items) {
                return res.json(items);
            }
        } catch (e) {
            next(e);
        }
    }

    async equip(req, res, next) {
        const errors = validationResult(req);
        if (!errors.isEmpty()) {
            return next(ApiError.BadRequest('Не верные параметры', errors.array()));
        }
        if(!Number.isInteger(req.body?.itemId)) {
            return next(ApiError.BadRequest('Не верные параметры', errors.array()));
        }
        try {
            const newItem = await ItemsService.getItem(req.body?.itemId, req.user.id);
            const currentEquipment = await ItemsService.getEquipment(req.user.id);
            if(currentEquipment[newItem.category] == req.body?.itemId) {
                return res.json({error: "already equipped"});
            } else {
                if(currentEquipment[newItem.category] != null) {
                    const currentItem = await ItemsService.getItem(currentEquipment[newItem.category],
req.user.id);
                    await ItemsService.uneqip(req.user.id, currentItem);
                }
            }
```

```
            const success = await ItemsService.equip(req.user.id, newItem);
            return res.json(success);
        } catch (e) {
            next(e);
        }
    }

    async unequip(req, res, next) {
        const errors = validationResult(req);
        if (!errors.isEmpty()) {
            return next(ApiError.BadRequest('Не верные параметры', errors.array()));
        }
        if(!Number.isInteger(req.body?.itemId)) {
            return next(ApiError.BadRequest('Не верные параметры', errors.array()));
        }
        try {
            const item = await ItemsService.getItem(req.body?.itemId, req.user.id);
            if(!item) {
                return next(ApiError.BadRequest('Предмет не существует'));
            }
            const currentEquipment = await ItemsService.getEquipment(req.user.id);
            if(currentEquipment[item.category] == req.body?.itemId) {
                const success = await ItemsService.uneqip(req.user.id, item);
                return res.json(success);
            } else {
                return next(ApiError.BadRequest('Предмет не экипирован'));
            }
        } catch (e) {
            next(e);
        }
    }

    async getItem(req, res, next) {
        const errors = validationResult(req);
        if (!errors.isEmpty()) {
            return next(ApiError.BadRequest('Не верные параметры', errors.array()));
        }
        if(!Number.isInteger(req.body?.id)) {
            return next(ApiError.BadRequest('Не верные параметры', errors.array()));
        }
        try {
            const itemInfo = await ItemsService.getItemInfo(req.body?.id, req.user.id);
            return res.json(itemInfo);
        } catch (e) {
            next(e);
        }
    }

    async use(req, res, next) {
        const errors = validationResult(req);
        if (!errors.isEmpty()) {
            return next(ApiError.BadRequest('Не верные параметры', errors.array()));
```

```
        }
        if(!Number.isInteger(req.body?.itemId)) {
            return next(ApiError.BadRequest('Не верные параметры', errors.array()));
        }
        try {
            const itemInfo = await ItemsService.getItem(req.body?.itemId, req.user.id);
            if(!itemInfo) {
                return next(ApiError.BadRequest('Предмет не существует', errors.array()));
            }
            if(itemInfo.category == "potion") {
                if(itemInfo.hp > 0 ){
                    const userInfo = await UserService.getUserStats(req.user.id);
                    let finalHp = userInfo.hp + itemInfo.hp;
                    if(finalHp > userInfo.maxHp) finalHp = userInfo.maxHp;
                    const successHP = await UserService.setUserHp(req.user.id, userInfo.maxHp);
                    if(itemInfo.amount <= 1) {
                        await ItemsService.deleteItem(req.user.id, req.body.itemId);
                    } else {
                        await ItemsService.removeAmount(req.user.id, req.body.itemId, 1);
                    }
                    if(successHP) {
                        return res.json({success:true, hp:finalHp, newAmount:itemInfo.amount-1})
                    } else {
                        return next(ApiError.BadRequest('Произошла непредвиденная ошибка',
errors.array()));
                    }
                } else {
                    return next(ApiError.BadRequest('Невозможно использовать', errors.array()));
                }
            } else {
                return next(ApiError.BadRequest('Невозможно использовать', errors.array()));
            }

        } catch (e) {
            next(e);
        }
    }
}
module.exports = new ItemsController();
```

**location-controller.js**

```
const LocationService = require("../service/location-service");
const MobService = require("../service/mob-service");
const ApiError = require('../exceptions/api-error');
const {validationResult} = require('express-validator');

class LocationController {
    async getLocation (req, res, next) {
        try {
            var locationInfo = await LocationService.getUserLocation(req.user.id);
            if(locationInfo.type === 2) {
                const mobs = await MobService.getMobsFromLocation(locationInfo.id);
```

```
            locationInfo['mobs'] = mobs;
        }
        return res.json(locationInfo)
    } catch (e) {
        next(e);
    }
}

async navigate (req, res, next) {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
        return next(ApiError.BadRequest('Не верные параметры', errors.array()));
    }
    // If request ID is a number
    if(!Number.isInteger(req.body?.locationId)) {
        return next(ApiError.BadRequest('Не верные параметры', errors.array()));
    }

    // If target location exists
    const targetLocation = await LocationService.getLocation(req.body.locationId);
    if(!targetLocation) {
        return next(ApiError.BadRequest('Локация не существует', errors.array()));
    }
    if(!targetLocation.opened) {
        return next(ApiError.BadRequest('Локация закрыта', errors.array()));
    }
    //If possible to get from current location
    const currentLocation = await LocationService.getUserLocation(req.user.id);
    const testingLinks = currentLocation.links?.find(i => i.id === req.body.locationId);
    if(!testingLinks) {
        const testingSplit = currentLocation.split?.find(i => i === req.body.locationId);
        if(!testingSplit) {
            return next(ApiError.BadRequest('Локация не досягаемая', errors.array()));
        }
    }
    try {
        const success = await LocationService.setUserLocation(req.user.id, req.body.locationId);
        if(success) {
            const locationInfo = await LocationService.getUserLocation(req.user.id);
            if(locationInfo.type === 2) {
                const mobs = await MobService.getMobsFromLocation(locationInfo.id);
                if(mobs) locationInfo['mobs'] = mobs;
            }
            return res.json(locationInfo);
        } else {
            return next(ApiError.BadRequest('Произошла ошибка обработки', errors.array()));
        }
    } catch (e) {
        next(e);
    }
}
```

```
}
module.exports = new LocationController();
```

**shop-controller.js**

```
const ShopService = require('../service/shop-service');
const UserService = require('../service/user-service');
const ItemService = require('../service/items-service');
const ApiError = require('../exceptions/api-error');
const LocationService = require('../service/location-service');
const {validationResult} = require('express-validator');

class ShopController {
    async getShop (req, res, next) {
        const errors = validationResult(req);
        if (!errors.isEmpty()) {
            return next(ApiError.BadRequest('Не верные параметры', errors.array()));
        }
        try {
            const userLocation = await LocationService.getUserLocationId(req.user.id);
            const shopInfo = await ShopService.getShop(req.body?.shopId);
            if(!shopInfo || userLocation != shopInfo.location) {
                throw ApiError.BadRequest(`Магазин не существует`)
            }
            if(req.body.buy >= 1) {
                const items = JSON.parse(shopInfo.items);
                const hasItemId = items.map((data) => {
                    return data.items.find(c => c == req.body.buy);
                })
                if(!hasItemId) {
                    throw ApiError.BadRequest(`Предмет не существует`)
                }
                const itemInfo = await ItemService.getItemTypeInfo(req.body.buy);
                console.log(itemInfo);
                const isPayed = await UserService.removeCol(req.user.id, itemInfo.price);
                if(isPayed) {
                    const success = await ItemService.addItems(req.user.id, [req.body.buy], [1]);
                    if(success) {
                        return res.json({success:true});
                    } else {
                        throw ApiError.BadRequest("Непредвиденная ошибка");
                    }

                } else {
                    return res.json({success:false});
                }
            } else if(req.body.sell >= 1) {
                const itemtypeid = await ItemService.getItemTypeById(req.body.sell, req.user.id);
                if(!itemtypeid) {
                    throw ApiError.BadRequest(`Предмет не существует`)
                }
                const itemInfo = await ItemService.getItemType(itemtypeid);
                if(itemInfo.stackable == '1') {
```

```
            // IF ITEM STACKABLE
        } else {
            const currentEquipment = await ItemService.getEquipmentIds(req.user.id);
            console.log("searching", req.body.sell, "in", currentEquipment);
            if(currentEquipment.find(o => o == req.body.sell)) {
                return res.json({fail:"Предмет экипирован"});
            }
            const IsDeleted = await ItemService.deleteItem(req.user.id, req.body.sell);
            if(IsDeleted) {
                await UserService.addCol(req.user.id, Math.floor(itemInfo.price*0.75));
                return res.json({success: true});
            } else {
                throw ApiError.BadRequest(`Произошла ошибка`);
            }
        }
    }
    // Если запросили айтем то отсылаем информацию
    if(req.body.itemId >= 1) {
        if(req.body?.selling === true) {
            const itemInfo = await ItemService.getItemInfo(req.body.itemId, req.user.id);
            const itemTypeInfo = await ItemService.getItemTypeInfo(itemInfo.itemtypeid);
            itemTypeInfo.price = Math.floor(itemTypeInfo.price*0.75);
            itemTypeInfo["amount"] = itemInfo.amount;
            return res.json(itemTypeInfo);
        } else {
            const itemInfo = await ItemService.getItemTypeInfo(req.body.itemId);
            return res.json(itemInfo);
        }
    }
    // Если выбранна категория то отсылаем содержимое
    if(req.body.category >= 0) {
        if(req.body?.selling === true) {
            const items = await ItemService.getItems(req.user.id,
JSON.parse(shopInfo.items)[req.body.category].category);
            if(items.hasOwnProperty("error")) {
                return res.json({fail:"Нет предметов на продажу"});
            }
            return res.json(items);
        } else {
            const itemIds = JSON.parse(shopInfo.items)[req.body.category].items;
            const items = await ItemService.getItemTypesInfo(itemIds);
            return res.json(items);
        }
    } else {
        // Если НЕ выбранна категория то отсылаем категории
        const info = {};
        let categories = JSON.parse(shopInfo.items);
        categories = categories.map((category)=> {
            return({title: category.title, category: category.category})
        });
        info["categories"] = categories;
        info["image"] = shopInfo.image;
```

```
            return res.json(info);
        }

    } catch (e) {
        next(e);
    }
}

}
module.exports = new ShopController();
```

**user-controller.js**

```
const UserService = require("../service/user-service");
const ItemsService = require("../service/items-service");
const {validationResult} = require('express-validator');
const ApiError = require('../exceptions/api-error');

class UserController {
    async registration (req, res, next) {
        try {
            const errors = validationResult(req);
            if (!errors.isEmpty()) {
                return next(ApiError.BadRequest('Ошибка валидации', errors.array()));
            }
            const {email, username, password} = req.body;
            const userData = await UserService.register(username, password, email);
            res.cookie('refreshToken', userData.refreshToken, {maxAge: 24 * 60* 60 * 1000, httpOnly:
true})
            return res.sendStatus(200);

        } catch (e) {
            next(e);
        }
    }

    async login (req, res, next) {
        try {
            const {username, password} = req.body;
            const userData = await UserService.login(username, password);
            res.cookie('refreshToken', userData.refreshToken, {maxAge: 365 * 24 * 60* 60 * 1000,
httpOnly: true})
            return res.json({"accessToken": userData.accessToken, "user": userData.user});
        } catch (e) {
            next(e);
        }
    }

    async logout (req, res, next) {
        try {
            const {refreshToken} = req.cookies;
            const token = await UserService.logout(refreshToken);
            res.clearCookie('refreshToken');
```

```
            return res.json(token);
        } catch (e) {
            next(e);
        }
    }

    async refresh (req, res, next) {
        try {
            const {refreshToken} = req.cookies;
            const userData = await UserService.refresh(refreshToken);
            return res.json(userData);
        } catch (e) {
            next(e);
        }
    }
    async revive (req, res, next) {
        try {
            const users = await UserService.revive(req?.user?.id);
            return res.json(users)
        } catch (e) {
            next(e);
        }
    }

    async getUsers (req, res, next) {
        try {
            const users = await UserService.getAllUsers();
            return res.json(users)
        } catch (e) {
            next(e);
        }
    }

    async getProfile (req, res, next) {
        try {
            const user = await UserService.getUserGameInfo(req?.user?.id);
            return res.json(user)
        } catch (e) {
            next(e);
        }
    }

    async getEquipment (req, res, next) {
        try {
            const user = await ItemsService.getEquipmentInfo(req?.user?.id);
            return res.json(user)
        } catch (e) {
            next(e);
        }
    }

    async getSkills (req, res, next) {
```

```
        try {
            const user = await UserService.getSkills(req?.user?.id);
            return res.json(user)
        } catch (e) {
            next(e);
        }
    }

    async getRatings (req, res, next) {
        try {
            const user = await UserService.getRatings();
            return res.json(user);
        } catch (e) {
            next(e);
        }
    }

    async skills (req, res, next) {
        const errors = validationResult(req);
        if (!errors.isEmpty()) {
            return next(ApiError.BadRequest('Ошибка валидации', errors.array()));
        }
        if(!Number.isInteger(req.body?.att) || !Number.isInteger(req.body?.def)) {
            return next(ApiError.BadRequest('Не верные параметры', errors.array()));
        }
        if(!Number.isInteger(req.body?.spd) || !Number.isInteger(req.body?.endur)) {
            return next(ApiError.BadRequest('Не верные параметры', errors.array()));
        }
        try {
            const {att, def, spd, endur} = req.body;
            if(att < 0 || spd < 0 || def < 0 || endur < 0) {
                return next(ApiError.BadRequest('Недопустимые значения', errors.array()));
            }
            if(att + spd + def + endur == 0 ) {
                return next(ApiError.BadRequest('Неверное количество навыков', errors.array()));
            }
            const currentSkills = await UserService.getSkills(req?.user?.id);
            if(att + spd + def + endur > currentSkills.skills ) {
                return next(ApiError.BadRequest('Ошибка валидации', errors.array()));
            }
            const success = await UserService.increaseSkills(req?.user?.id, att, def, spd, endur,
att+def+spd+endur)
            return res.json({success: success});
        } catch (e) {
            next(e);
        }
    }

}
module.exports = new UserController();
```

**api-error.js**

```
module.exports = class ApiError extends Error {
    status;
    errors;

    constructor(status, message, errors = []) {
        super(message);
        this.status = status;
        this.errors = errors;
    }

    static UnauthorizedError() {
        return new ApiError(401,"Пользователь не авторизован")
    }

    static ForbiddenError() {
        return new ApiError(403,"Доступ запрещён")
    }

    static BadRequest(message, errors = []) {
        return new ApiError(400, message, errors)
    }
}
```

**auth-middleware.js**

```
const ApiError = require('../exceptions/api-error');
const tokenService = require('../service/token-service');

module.exports = function (req, res, next) {
    try {
        const AuthorizationHeader = req.headers.authorization;
        if(!AuthorizationHeader) {
            return next(ApiError.UnauthorizedError());
        }
        const accessToken = AuthorizationHeader.split(' ')[1];
        if(!accessToken) {
            return next(ApiError.ForbiddenError());
        }
        const userData = tokenService.validateAccessToken(accessToken);
        if(!userData) {
            return next(ApiError.ForbiddenError());
        }
        if(userData.hp <= 0 && req.route.path != "/revive") {
            console.log(req.route.path);
            return next(ApiError.ForbiddenError());
        }
        req.user = userData;
        next();
    } catch (error) {
        console.log(error);
        return next(ApiError.UnauthorizedError());
    }
}
```

**error-middlewares.js**

```
const ApiError = require('../exceptions/api-error')

module.exports = function (err, req, res, next) {
    console.log(err);
    if (err instanceof ApiError) {
        return res.status(err.status).json({message: err.message, errors: err.errors})
    }
    return res.status(500).json({message: 'Непредвиденная ошибка'})
}
```

**query-model.js**

```
const { json } = require("body-parser");
const mysql = require("mysql2")

const dbHost = "localhost"
const dbName = "game";
const dbUser = "root";
const dbPass = "root";
const connection = mysql.createConnection({ host: dbHost, user: dbUser, password: dbPass,
database: dbName});

function execute(querytext) {
    return new Promise(function(resolve) {
        try {
            connection.query(querytext, function (error, results) {
                if(results?.length == 0) resolve(null);
                resolve(results);
            });
        } catch (e) {
            console.log(e);
        }
        }
    );

}

module.exports = {execute};
```

**token-model.js**

```
const queryModel = require('./query-model')

async function getRefreshTokenById(userid){
    const query = `SELECT refresh_token FROM USERS WHERE id = "${userid}"`;
    const res = await queryModel.execute(query)
    if(res) {
        return res;
    } else {
        return null;
    }
}
```

```
async function findRefreshToken(refreshToken) {
    const query = `SELECT id FROM users WHERE refresh_token = "${refreshToken}"`;
    const res = await queryModel.execute(query)
    if(res) {
        return res[0].id;
    } else {
        return null;
    }
}

async function saveRefreshToken(userId, refreshToken) {
    const query = `UPDATE users SET refresh_token = "${refreshToken}" WHERE id =
"${userId}"`;
    const res = await queryModel.execute(query);
    if(res) {
        return res;
    } else {
        return null;
    }
}

async function saveAccessToken(userId, accessToken) {
    const query = `UPDATE users SET access_token = "${accessToken}" WHERE id =
"${userId}"`;
    const res = await queryModel.execute(query);
    if(res) {
        return res;
    } else {
        return null;
    }
}

async function removeToken(refreshToken) {
    const query = `UPDATE users SET refresh_token = "", access_Token = "" WHERE
refresh_token = "${refreshToken}"`;
    const res = await queryModel.execute(query)
    if(res) {
        return res;
    } else {
        return null;
    }
}


module.exports = {getRefreshTokenById, saveRefreshToken, saveAccessToken, removeToken,
findRefreshToken};
```

**user-model.js**

```
const queryModel = require('./query-model')

async function getIdByName(name){
    const query = `SELECT id FROM USERS WHERE username = "${name}"`;
```

```
    const res = await queryModel.execute(query)
    if(res) {
        return res[0].id;//Getting id from first item
    } else {
        return null;
    }
}

async function findByEmail(email){
    const query = `SELECT id FROM USERS WHERE email = "${email}"`;
    const res = await queryModel.execute(query)
    if(res) {
        return res[0].id;//Getting id from first item
    } else {
        return null;
    }
}

async function getInfoById(userId){
    const query = `SELECT id, username, role, speed + sspd AS speed, email, hp, maxHp, lvl FROM
USERS WHERE id = "${userId}"`;
    const res = await queryModel.execute(query)
    if(res) {
        return res;
    } else {
        return null;
    }
}
async function getPassword(userId){
    const query = `SELECT password FROM USERS WHERE id = "${userId}"`;
    const res = await queryModel.execute(query);
    if(res) {
        return res[0].password;
    } else {
        return null;
    }
}

async function create(login, pass, email){
    const query = `INSERT into users (username, password, email) VALUES ("${login}",
"${pass}", "${email}")`;
    const res = await queryModel.execute(query)
    if(res) {
        return res.insertId;
    } else {
        return null;
    }
}

async function getAllUsers(){
    const query = `SELECT * FROM users`;
    const res = await queryModel.execute(query)
```

```
    if(res) {
        return res;
    } else {
        return null;
    }
}
module.exports = {getIdByName, create, getInfoById, findByEmail, getPassword, getAllUsers};
```

**index.js**

```
const Router = require('express').Router;
const express = require('express');
const UserController = require('../controllers/user-controller');
const LocationController = require('../controllers/location-controller');
const BattleController = require('../controllers/battle-controller');
const EventsController = require('../controllers/events-controller');
const ItemsController = require("../controllers/items-controller");
const ShopController = require('../controllers/shop-controller');
const RealtimeService = require('../service/realtime-service');
const ChatController = require('../controllers/chat-controller');
const router = new Router();
const {body} = require('express-validator');
const authMiddleware = require('../middlewares/auth-middleware');
const cron = require('node-cron');

router.post('/login',
    body('password').isLength({min: 6,max: 30}),
    UserController.login
);
router.post('/register',
    body('email').isEmail(),
    body('password').isLength({min: 6,max: 30}),
    UserController.registration
);
router.get('/logout',   UserController.logout);
router.get('/refresh',  UserController.refresh);
router.get('/revive',  authMiddleware, UserController.revive);
router.get('/getUsers', authMiddleware, UserController.getUsers);
router.get('/getRatings', authMiddleware, UserController.getRatings);
router.get('/chat', authMiddleware, ChatController.getMessages);
router.post('/message', authMiddleware, ChatController.sendMessage);
router.post('/game',    authMiddleware, LocationController.getLocation);
router.post('/navigate', authMiddleware, LocationController.navigate);
router.post('/getItems', authMiddleware, ItemsController.getItems);
router.post('/getItem', authMiddleware, ItemsController.getItem);
router.post('/equip', authMiddleware, ItemsController.equip);
router.post('/use', authMiddleware, ItemsController.use);
router.post('/unequip', authMiddleware, ItemsController.unequip);
router.post('/battle', authMiddleware, BattleController.getBattle);
router.post('/attack', authMiddleware, BattleController.attack);
router.get('/events', authMiddleware, EventsController.eventsHandler);
router.post('/equipment', authMiddleware, UserController.getEquipment);
router.post('/profile', authMiddleware, UserController.getProfile);
```

```
router.post('/getSkills', authMiddleware, UserController.getSkills);
router.post('/skills', authMiddleware, UserController.skills);
router.post('/shop', authMiddleware, ShopController.getShop);
router.use(express.static('public'));

cron.schedule('*/20 * * * * *', RealtimeService.Healing);

module.exports = router;
```

**chat-service.js**

```
const queryModel = require('../models/query-model');

class ChatService {
   async getMessages() {
      const query = `SELECT * FROM (
                     SELECT * FROM messages ORDER BY id DESC LIMIT 50
                  ) t1 ORDER BY t1.id`
      const res = await queryModel.execute(query);
      if(res) {
         return res;
      } else {
         console.log("error getMessages");
         return null;
      }
   }

   async sendMessage(userName, text) {

      const query = `INSERT INTO messages (username, text, time) VALUES ('${userName}',
'${text}','${Math.floor(Date.now() / 1000)}')`
      const res = await queryModel.execute(query);
      if(res.insertId) {
         return res.insertId;
      } else {
         console.log("error sendMessage");
         return null;
      }
   }
}
module.exports = new ChatService();
```

**items-service.js**

```
const queryModel = require('../models/query-model');

class ItemService {
   async addItems(userId, items, amounts) {
      let ItemsAddAmount = [];
      let ItemsToCreate = [];
      let titles = [];
      const infoQuery = `SELECT * FROM
            (SELECT * FROM itemtypes WHERE itemtypeid IN (${items.toString()})) t1
         LEFT JOIN
```

```
            (SELECT itemtypeid,amount FROM items WHERE itemtypeid IN (${items.toString()})
AND userId = ${userId}) t2
          ON (t2.itemtypeid = t1.itemtypeid)`;

        const existingitems = await queryModel.execute(infoQuery);
        for(let i = 0; i < items.length; i++) {
          titles.push(existingitems[i]?.title);
            //Предмет существует, стакается
          if(items[i] == existingitems[i]?.itemtypeid && existingitems[i]?.stackable == 1) {
            ItemsAddAmount.push({"id":items[i], "amount": amounts[i]});

            //Предмет существует, не стакается
          } else if (items[i] == existingitems[i]?.itemtypeid && existingitems[i]?.stackable == 0) {
            ItemsToCreate.push({"id":items[i], "amount": 1, "category": existingitems[i]?.category,
"title": titles[i], "attack": existingitems[i]?.attack, "def": existingitems[i]?.def, "hp":
existingitems[i]?.hp, "speed": existingitems[i]?.speed});

            //Предмет не существует, стакается
          } else if (existingitems[i]?.stackable == 1){
            ItemsToCreate.push({"id":items[i], "amount": amounts[i], "category":
existingitems[i]?.category, "title": titles[i], "attack": existingitems[i]?.attack, "def":
existingitems[i]?.def, "hp": existingitems[i]?.hp, "speed": existingitems[i]?.speed});

            //Предмет не существует, не стакается
          } else {
            ItemsToCreate.push({"id":items[i], "amount": 1, "category": existingitems[i]?.category,
"title": titles[i], "attack": existingitems[i]?.attack, "def": existingitems[i]?.def, "hp":
existingitems[i]?.hp, "speed": existingitems[i]?.speed});
          }
        }
        if(ItemsAddAmount.length) {
          console.log("adding", ItemsAddAmount.length, "items");
          let itemsStr = "";
          ItemsAddAmount.map((i) => {
                var item = 'when itemtypeid = '+i.id+' then amount + ' +i.amount;
                itemsStr = itemsStr? itemsStr+' '+item : item;
                }
              );
          const addQuery =  `UPDATE items SET amount = (case ${itemsStr} end)
                        WHERE itemtypeid in (${items.toString()}) AND userid = ${userId}`;

          await queryModel.execute(addQuery);
        }
        if(ItemsToCreate.length) {
          console.log("creating", ItemsToCreate.length, "items");
          let itemsStr = "";
          ItemsToCreate.map((i) => {
                var item = '(' + userId+','+i.id+','+i.amount+',"'+i.category+ '","'+ i.title +'",' + i.attack
+',' +i.def+',' +i.hp +','+i.speed + ')';
                itemsStr = itemsStr? itemsStr+', '+item : item;
                }
              );
```

```
            const createQuery = `INSERT INTO items (userId, itemtypeid, amount, category, title,
attack, def, hp, speed) VALUES ${itemsStr}`;
            await queryModel.execute(createQuery);
        }
        if(existingitems) {
            return titles;
        } else {
            console.log("error addItems");
            return null;
        }
    }
    async getItems(userId, category) {
        const query = `SELECT itemtypeid, amount, title, itemId FROM items WHERE userId =
${userId} AND category = "${category}"`;
        const res = await queryModel.execute(query);
        if(res) {
            return res;
        } else if(res==null) {
            return ({error: "Нет предметов"});
        } else {
            console.log("error getItems");
            return null;
        }
    }

    async getItemType(typeId) {
        const query = `SELECT * FROM itemtypes WHERE itemtypeid = "${typeId}"`;
        const res = await queryModel.execute(query);
        if(res) {
            return res[0];
        } else if(res==null) {
            console.log("Не существующий тип");
            return null;
        } else {
            console.log("error getItems");
            return null;
        }
    }

    async getItemTypeInfo(typeId) {
        const query = `SELECT title, itemtypeid, attack, def, hp, speed, price, stackable, category
FROM itemtypes WHERE itemtypeid = "${typeId}"`;
        const res = await queryModel.execute(query);
        if(res) {
            return res[0];
        } else if(res==null) {
            console.log("Не существующий тип");
            return null;
        } else {
            console.log("error getItems");
            return null;
        }
```

```
    }

    async getItemInfo(itemId, ownerId) {
        const query = `SELECT itemtypeid, title, amount, attack, def, hp, itemtypeid, speed FROM
items WHERE itemId = ${itemId} AND userId=${ownerId}`;
        const res = await queryModel.execute(query);
        if(res) {
            return res[0];
        } else {
            console.log("error getItemInfo");
            return null;
        }
    }
    async getItem(itemId, ownerId) {
        const query = `SELECT * FROM items WHERE itemId = ${itemId} AND
userId=${ownerId} AND amount > 0`;
        const res = await queryModel.execute(query);
        if(res) {
            return res[0];
        } else {
            console.log("error getItem");
            return null;
        }
    }

    async getItemsInfo(itemsId) {
        const query = `SELECT * FROM items WHERE itemId IN (${itemsId})`;
        const res = await queryModel.execute(query);
        if(res) {
            return res;
        } else {
            console.log("error getItem");
            return null;
        }
    }

    async getItemTypesInfo(itemIds) {

        const query = `SELECT itemtypeid, title, category, price FROM itemtypes WHERE
itemtypeid IN (${itemIds.toString()})`;
        const res = await queryModel.execute(query);
        if(res) {
            return res;
        } else {
            console.log("error getItems");
            return null;
        }
    }

    async getEquipment(userId) {
        const query1 = `SELECT * FROM equipment WHERE userId = ${userId}`;
        const res1 = await queryModel.execute(query1);
```

```
    if(res1) {
        return res1[0];
    } else {
        const query2 = `INSERT INTO equipment (userId) values (${userId})`;
        const res2 = await queryModel.execute(query2);
        return getEquipment(userId);
    }
}

async getEquipmentInfo(userId) {
    const query1 = `SELECT * FROM equipment WHERE userId = ${userId}`;
    const res1 = await queryModel.execute(query1);
    if(res1) {
        //Checking if not null
        let equipmentIds = [];
        Object.keys(res1[0]).forEach(function(k){
            if(res1[0][k]) {
                equipmentIds.push(res1[0][k]);
            }
        });
        if(equipmentIds.length > 0) {
            return this.getItemsInfo(equipmentIds);
        } else {
            return res1[0];
        }
    } else {
        await queryModel.execute(`INSERT INTO equipment (userId) values (${userId})`);
        return getEquipmentInfo(userId);
    }
}

async getEquipmentIds(userId) {
    const query1 = `SELECT * FROM equipment WHERE userId = ${userId}`;
    const res1 = await queryModel.execute(query1);
    if(res1) {
        //Checking if not null
        let equipmentIds = [];
        Object.keys(res1[0]).forEach(function(k){
            if(res1[0][k]) {
                equipmentIds.push(res1[0][k]);
            }
        });
        if(equipmentIds.length > 0) {
            return equipmentIds;
        } else {
            return null;
        }
    } else {
        await queryModel.execute(`INSERT INTO equipment (userId) values (${userId})`);
        return getEquipmentInfo(userId);
    }
}
```

```
  async uneqip(userId, item) {
    const query1 =  `UPDATE equipment SET ${item.category} = null WHERE
userId=${userId}`
    const res1 = await queryModel.execute(query1);
    const query2 =  `UPDATE users SET attack = attack - ${item.attack},
            maxhp = maxhp - ${item.hp}, def = def - ${item.def}, speed = speed -
${item.speed}
            WHERE id=${userId}`
    const res2 = await queryModel.execute(query2);
    if(res1 && res2) {
      return true;
    } else {
      console.log("error uneqip");
      return false;
    }
  }

  async equip(userId, item) {
    const query1 = `UPDATE equipment SET ${item.category} = ${item.itemId} WHERE
userId=${userId}`;
    const res1 = await queryModel.execute(query1);
    const query2 =  `UPDATE users SET attack = attack + ${item.attack},
             maxhp = maxhp + ${item.hp}, def = def + ${item.def}, speed = speed +
${item.speed}
            WHERE id=${userId}`;
    const res2 = await queryModel.execute(query2);
    if(res1 && res2) {
      return true;
    } else {
      console.log("error equip", query1, res1, query2, res2);
      return false;
    }
  }
  async deleteItem(userId, itemId) {
    const query = `DELETE FROM items WHERE itemId = ${itemId} AND userId=${userId}`;
    const res = await queryModel.execute(query);
    if(res) {
      return true;
    } else {
      console.log("error deleteItem");
      return false;
    }
  }
  async removeAmount(userId, itemId, amount) {
    const query = `UPDATE items SET amount = amount - ${amount} WHERE itemId =
${itemId} AND userId=${userId}`;
    const res = await queryModel.execute(query);
    if(res) {
      return true;
    } else {
      console.log("error removeAmount");
```

```
            return false;
        }
    }
}
module.exports = new ItemService();
```

**location-service.js**

```
const queryModel = require('../models/query-model');

class UserService {
    async getUserLocation(userId) {
        const locationId = await this.getUserLocationId(userId);
        const query = `SELECT * FROM locations WHERE id = ${locationId} `;
        const res = await queryModel.execute(query);
        if(res) {
            var data;
            data = JSON.parse(res[0].items);
            res[0].items = JSON.parse(res[0].items);
            data['title'] = res[0].title;
            data['image'] = res[0].image;
            data['type'] = res[0].type;
            data['id'] = res[0].id;
            return data;
        } else {
            console.log("error");
            return null;
        }
    }

    async getUserLocationId(userId) {
        const query = `SELECT location FROM users WHERE id = ${userId} `;
        const res = await queryModel.execute(query);
        if(res) {
            return res[0].location;
        } else {
            console.log("error");
            return null;
        }
    }

    async setUserLocation(userId, locationId) {
        const query = `UPDATE users SET location = "${locationId}" WHERE id = "${userId}"`;
        const res = await queryModel.execute(query);
        if(res.affectedRows) {
            return true;
        } else {
            console.log("error");
            return false;
        }
    }

    async getLocation(locationId) {
```

```
        const query = `SELECT * FROM locations WHERE id = "${locationId}"`;
        const res = await queryModel.execute(query);
        if(res) {
            return res[0];
        } else {
            console.log("error");
            return false;
        }
    }

    async openLocation(locationId) {
        const query = `UPDATE locations SET opened = 1 WHERE id = ${locationId}`;
        const res = await queryModel.execute(query);
        if(res) {
            return true;
        } else {
            console.log("error");
            return false;
        }
    }

    async canNavigate(userId, locationId) {
        const query = `SELECT title FROM locations WHERE id = "${locationId}"`;
        const res = await queryModel.execute(query);
        if(res) {
            return true;
        } else {
            console.log("error");
            return false;
        }
    }

    async getUsersInLocation(locationId, excludeId=0) {
        const query = `SELECT id FROM users WHERE location = "${locationId}"`;
        const res = await queryModel.execute(query);
        if(res) {
            var output = [];
            for (var i=0; i < res.length ; ++i) {
                if(res[i]["id"] != excludeId) output.push(res[i]["id"]);
            }
            return output;
        } else {
            console.log("error");
            return false;
        }
    }
}
module.exports = new UserService();
```

**mob-service.js**

```
const queryModel = require('../models/query-model');
```

```
class MobService {
   async getMobsFromLocation(locationId) {
      const query = `SELECT id, name, hp, maxHp, image, timerevive, type FROM mobs WHERE
location = ${locationId} LIMIT 10`;
      const res = await queryModel.execute(query);
      if(res) {
         let deadMobs = [];
         const currentTime = (Date.now() / 1000 | 0);
         res.forEach(mob => {
            if(mob.timerevive <= currentTime && mob.hp <= 0) {
               deadMobs.push(mob.id);
            }
         });
         if(deadMobs.length != 0) {
            const queryForDead = `UPDATE mobs SET hp = maxHp WHERE id IN
(${deadMobs.toString()})`;
            await queryModel.execute(queryForDead);
            return this.getMobsFromLocation(locationId);
         } else {
            return res;
         }
      } else {
         return null;
      }
   }

   async getMob(mobId) {
      const query = `SELECT * FROM mobs WHERE id = ${mobId} `;
      const res = await queryModel.execute(query);
      if(res) {
         const currentTime = (Date.now() / 1000 | 0);
         if(res[0].timerevive <= currentTime && res[0].hp <= 0) {
            const queryForDead = `UPDATE mobs SET hp = maxHp WHERE id = "${mobId}"`;
            await queryModel.execute(queryForDead);
            return this.getMob(mobId);
         } else {
            return res[0];
         }
      } else {
         console.log("error getMob");
         return null;
      }
   }

   async setMobHp(mobId, mobHp) {
      const query = `UPDATE mobs SET hp = "${mobHp}" WHERE id = "${mobId}"`;
      const res = await queryModel.execute(query);
      if(res) {
         return true;
      } else {
         console.log("error setMobHp");
         return null;
```

```
        }
    }
    async killMob(mobInfo) {
        const query = `UPDATE mobs SET hp = 0, timerevive=${(Date.now() / 1000 | 0) +
mobInfo.respawntime} WHERE id = "${mobInfo.id}"`;
        const res = await queryModel.execute(query);
        if(res) {
            return true;
        } else {
            console.log("error killMob");
            return null;
        }
    }
}
module.exports = new MobService();
```

**realtime-service.js**

```
const UserService = require("./user-service");
const EventsController = require('../controllers/events-controller');

class RealtimeService {
    async Healing() {
        const damagedUsers = await UserService.getDamagedUsers();
        if(damagedUsers) {

            let users = [];
            let newHP = [];
            damagedUsers.map((user) => {

                user.hp += (user.endurance+user.sendur)*2;
                if(user.hp > user.maxHp) user.hp = user.maxHp;
                users.push(user.id);
                newHP.push(user.hp);
                EventsController.sendEventToUser(user.id, {healing:user.hp});
            })
            await UserService.setUsersHp(users, newHP);
        }
    }
}
module.exports = new RealtimeService();
```

**token-service.js**

```
const jwt = require("JsonWebToken");
const tokenModel = require('../models/token-model');

JWT_ACCESS_SECRET = "SAO-ACCHY6RX-333";
JWT_REFRESH_SECRET = "SAO-REFHY6RX-444";

class TokenService {
    generateTokens(payload) {
        const accessToken = jwt.sign(...payload, JWT_ACCESS_SECRET, {expiresIn:'1d'})
        const refreshToken = jwt.sign(...payload, JWT_REFRESH_SECRET, {expiresIn:'365d'})
```

```
        return {
            accessToken,
            refreshToken
        }
    }

    async saveRefreshToken(userId, token) {
        await tokenModel.saveRefreshToken(userId,token);
    }
    async saveAccessToken(userId, token) {
        await tokenModel.saveAccessToken(userId,token);
    }

    async removeToken(refreshToken) {
        const tokenData = await tokenModel.removeToken(refreshToken);
        return tokenData;
    }

    async findRefreshToken(refreshToken) {
        const tokenData = await tokenModel.findRefreshToken(refreshToken);
        return tokenData;
    }

    validateAccessToken(token) {
        try {
            const userData = jwt.verify(token, JWT_ACCESS_SECRET);
            return userData;
        } catch (error) {
            return null;
        }
    }

    validateRefreshToken(token) {
        try {
            const userData = jwt.verify(token, JWT_REFRESH_SECRET);
            return userData;
        } catch (error) {
            return null;
        }
    }
}
module.exports = new TokenService();
```

**user-service.js**

```
const UserModel = require('../models/user-model');
const bcrypt = require("bcrypt");
const tokenService = require('./token-service');
const ApiError = require('../exceptions/api-error');
const queryModel = require('../models/query-model');
const EventsController = require('../controllers/events-controller');

class UserService {
```

```
async register(username, password, email) {
    const candidate = await UserModel.getIdByName(username);
    if (candidate) {
        throw ApiError.BadRequest(`Пользователь с таким логином уже существует`)
    }
    const hashPassword = await bcrypt.hash(password, 3);
    const resultId = await UserModel.create(username, hashPassword, email);
    const result = await UserModel.getInfoById(resultId);
    const tokens = tokenService.generateTokens(result);

    await tokenService.saveRefreshToken(resultId, tokens.refreshToken);
    await tokenService.saveAccessToken(resultId, tokens.accessToken);
    return { ...tokens, user: result }
}

async login(username, password) {
    const userId = await UserModel.getIdByName(username);
    if (!userId) {
        throw ApiError.BadRequest(`Пользователь с таким логином не был найден`)
    }
    const userPass = await UserModel.getPassword(userId);
    const isPassEquals = await bcrypt.compare(password, userPass);
    if (!isPassEquals) {
        throw ApiError.BadRequest(`Неверный пароль`)
    }
    const result = await UserModel.getInfoById(userId);
    const tokens = tokenService.generateTokens(result);

    await tokenService.saveRefreshToken(userId, tokens.refreshToken);
    await tokenService.saveAccessToken(userId, tokens.accessToken);
    return { ...tokens, user: result }
}

async logout(refreshToken) {
    await tokenService.removeToken(refreshToken);
}

async refresh(refreshToken) {
    if (!refreshToken) {
        throw ApiError.UnauthorizedError();
    }
    const userData = tokenService.validateRefreshToken(refreshToken);
    const userIdWithToken = await tokenService.findRefreshToken(refreshToken);
    if(!userData || !userIdWithToken) {
        throw ApiError.UnauthorizedError();
    }

    const result = await UserModel.getInfoById(userIdWithToken);
    const tokens = tokenService.generateTokens(result);
    await tokenService.saveAccessToken(userIdWithToken, tokens.accessToken);
    return { accessToken: tokens.accessToken, user: result, role: result[0].role}
}
```

```
    async getAllUsers() {
        const users = await UserModel.getAllUsers()
        return users;
    }



    async getUserStats(userId) {
        const query = `SELECT hp, maxHp, attack FROM users WHERE id = "${userId}"`;
        const res = await queryModel.execute(query);
        if(res) {
            return res[0];
        } else {
            console.log("error getUserStats");
            return false;
        }
    }

    async getUserExp(userId) {
        const query = `SELECT xp, lvl FROM users WHERE id = "${userId}"`;
        const res = await queryModel.execute(query);
        if(res) {
            return res[0];
        } else {
            console.log("error getUserExp");
            return false;
        }
    }

    async getUserGameInfo(userId) {
        const query = `SELECT location, hp, maxHp, lvl, attack, skills, def, col, xp, speed, endurance
FROM users WHERE id = "${userId}"`;
        const res1 = await queryModel.execute(query);
        const res2 = await this.getSkills(userId);
        if(res1 && res2) {
            res1[0].attack += res2.satt;
            res1[0].def += res2.sdef;
            res1[0].speed += res2.sspd;
            res1[0].endurance += res2.sendur;
            res1[0]['xpLeft'] = this.nextLevel(res1[0].lvl+1);
            return res1[0];
        } else {
            console.log("error getUserGameInfo");
            return false;
        }
    }

    async getDamagedUsers() {
        const query = `SELECT id, hp, maxHp, endurance, sendur FROM users WHERE hp < maxHp
AND hp > 0`;
        const res = await queryModel.execute(query);
```

```
        if(res) {
            return res;
        } else {
            return null;
        }
    }

    async getUserName(userId) {
        const query = `SELECT username FROM users WHERE id=${userId}`;
        const res = await queryModel.execute(query);
        if(res) {
            return res[0].username;
        } else {
            console.log("error getUserName");
            return null;
        }
    }

    async increaseDeaths(userId) {
        const query = `UPDATE users SET deaths = deaths + 1 WHERE id = ${userId}`;
        const res = await queryModel.execute(query);
        if(res) {
            return true;
        } else {
            console.log("increaseDeaths");
            return null;
        }
    }

    async setUserHp(userId, hp) {
        const query = `UPDATE users SET hp = "${hp}" WHERE id = "${userId}"`;
        if(hp <= 0) await this.increaseDeaths(userId);
        const res = await queryModel.execute(query);
        if(res) {
            return true;
        } else {
            console.log("error setUserHp");
            return null;
        }
    }

    async setUsersHp(users, newHP) {
        let userStr = "";
        for(let i = 0; i < users.length; i++) {
            let item = '(' + users[i]+','+newHP[i]+')';
            userStr = userStr? userStr+', '+item : item;
        }
        const query = `INSERT INTO users (id, hp) VALUES ${userStr} ON DUPLICATE KEY
UPDATE hp = VALUES(hp)`;
        const res = await queryModel.execute(query);
        if(res) {
            return true;
```

```
        } else {
            console.log("error setUsersHp");
            return null;
        }
    }

    async revive(userId) {
        const query = `UPDATE users SET hp = maxHp, location = 4 WHERE id = "${userId}"`;
        const res = await queryModel.execute(query);
        if(res) {
            return true;
        } else {
            console.log("error revive");
            return null;
        }
    }
    nextLevel(lvl) {
        return Math.round( Math.pow((lvl/0.1),2));
    }
    async addXP(userId, exp) {
        let query = `UPDATE users SET xp = xp + ${exp} WHERE id = "${userId}"`;
        const userExp = await this.getUserExp(userId);
        if(userExp.xp + exp >= this.nextLevel(userExp.lvl+1)) {
            query = `UPDATE users SET xp = xp + ${exp}, skills=skills+10, lvl = lvl+1, maxHp =
maxHp+200 WHERE id = "${userId}"`;
            await EventsController.sendEventToUser(userId, {notification: "LvlUp"})
        }
        const res = await queryModel.execute(query);
        if(res) {
            return true;
        } else {
            console.log("error addXP");
            return null;
        }
    }

    async addCol(userId, col) {
        let query = `UPDATE users SET col = col + ${col} WHERE id = "${userId}"`;
        const res = await queryModel.execute(query);
        if(res) {
            return true;
        } else {
            console.log("error addCol");
            return null;
        }
    }

    async removeCol(userId, col) {
        const query = `UPDATE users SET col = col - ${col} WHERE id = "${userId}" AND col >=
${col}`;
        const res = await queryModel.execute(query);
        if(res.affectedRows) {
```

```
            return true;
        } else {
            console.log("error removeCol");
            return false;
        }
    }

    async getSkills(userId) {
        const query = `SELECT skills, satt, sdef, sendur, sspd FROM users WHERE id = ${userId}`;
        const res = await queryModel.execute(query);
        if(res) {
            return res[0];
        } else {
            console.log("error getSkills");
            return null;
        }
    }

    async increaseSkills(userId, att, def, spd, endur, total) {
        const query = `UPDATE users SET satt = satt + ${att}, sdef=sdef+${def},
sspd=sspd+${spd},sendur=sendur+${endur}, skills=skills-${total} WHERE id = ${userId}`;
        const res = await queryModel.execute(query);
        if(res.affectedRows) {
            return true;
        } else {
            console.log("error increaseSkills");
            return false;
        }
    }

    async getRatings() {
        const winsquery = `SELECT username, wins FROM users ORDER BY wins DESC LIMIT 3`;
        const wins = await queryModel.execute(winsquery);
        const lvlquery = `SELECT username, lvl FROM users ORDER BY lvl DESC LIMIT 3`;
        const lvl = await queryModel.execute(lvlquery);
        const colquery = `SELECT username, col FROM users ORDER BY col DESC LIMIT 3`;
        const col = await queryModel.execute(colquery);
        if(wins) {
            return ({wins, lvl, col});
        } else {
            console.log("error getRatings");
            return null;
        }
    }
}
module.exports = new UserService();
```

**ДОДАТОК Б**

# ВІДГУК КЕРІВНИКА ЕКОНОМІЧНОГО РОЗДІЛУ

**ДОДАТОК В**

## Перелік файлів на диску

| Ім’я файлу | Опис |
|---|---|
| Пояснювальні документи | |
| Кваліфікаційна робота Міхно.docx | Пояснювальна записка до кваліфікаційної роботи. Документ Word. |
| Кваліфікаційна робота Міхно.pdf | Пояснювальна записка до кваліфікаційної роботи в форматі PDF |
| Програма | |
| Міхно.rar | Архів. Містить коди програми і скомпільовану програму |
| Презентація | |
| Міхно.ppt | Презентація кваліфікаційної роботи |