МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему: **«РОЗРОБКА ІГРОВОЇ ІНФОРМАЦІЙНОЇ ПЛАТФОРМИ ЗАСОБАМИ REACT.JS ТА NODE.JS»**

Виконав: студент 4 курсу, групи 6.1212-пі-1
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми програмна інженерія
(назва освітньої програми)
В.В. Матвійчук
(ініціали та прізвище)

Керівник доцент кафедри програмної інженерії, к.ф.-м.н. Кривохата А.Г.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент доцент кафедри комп'ютерних наук, доцент, к.т.н. Решевська К.С.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2026

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти бакалавр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма програмна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2025 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Матвійчуку Віталію Вікторовичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка ігрової інформаційної платформи засобами React.js та Node.js

керівник роботи Кривохата Анастасія Григорівна, к.ф.-м.н.
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 02 » грудня 2025 року № 1741-с

2. Строк подання студентом роботи 01.06.2026 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Визначення вимог до інформаційної системи
2. Проєктування інформаційної системи «GenTixCraft Online».
3. Програмна реалізація інформаційної системи.

4. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою доповіді

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 09.12.2025 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	19.12.2025	
2.	Збір вихідних даних.	23.01.2026	
3.	Обробка методичних та теоретичних	20.02.2026	
	джерел.		
4.	Розробка першого та другого розділу.	06.04.2026	
5.	Розробка третього розділу.	15.05.2026	
6.	Оформлення та нормоконтроль	25.05.2026	
	кваліфікаційної роботи бакалавра.		
7.	Захист кваліфікаційної роботи.	15.06.2026	

Студент _____ В.В. Матвійчук
(підпис) (ініціали та прізвище)

Керівник роботи _____ А.Г. Кривохата
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота бакалавра «Розробка ігрової інформаційної платформи засобами React.js та Node.js»: 61 с., 54 рис., 17 джерел.

БАЗА ДАНИХ, ВЕБЗАСТОСУНОК, ІГРОВА ПЛАТФОРМА, ІНТЕРФЕЙС КОРИСТУВАЧА, КЛІЄНТ-СЕРВЕРНА АРХІТЕКТУРА, СИСТЕМА АВТОРИЗАЦІЇ, УПРАВЛІННЯ КОНТЕНТОМ, API, NODE.JS, REACT.JS.

Об’єкт дослідження – процес розробки ігрової платформи із використанням клієнт-серверних технологій.

Мета роботи – розробка ігрової інформаційної платформи з використанням React.js на стороні клієнта та Node.js на стороні сервера для забезпечення зручного доступу до новин, користувацьких сервісів і адміністрування.

Методи дослідження – методи об’єктно-орієнтованого програмування, методи моделювання програмного забезпечення, тестування вебзастосунку.

У кваліфікаційній роботі розроблено ігрову інформаційну платформу, орієнтовану на взаємодію користувачів із контентом, новинами та сервісами для ігрової спільноти. Вебзастосунок реалізовано з використанням бібліотеки React.js для побудови динамічного інтерфейсу користувача та середовища Node.js для серверної логіки й обробки запитів. Розроблено систему реєстрації та авторизації користувачів, модуль новин, панель керування адміністратора та інтеграцію з базою даних. Додатково реалізовано AI-асистента на базі Google Gemini API, який забезпечує генерацію відповідей українською мовою та взаємодію користувачів із системою через інтерфейс вебзастосунку. Запропоноване рішення забезпечує швидкодію, ефективність і зручність використання. Результати роботи можуть бути використані як основа для створення сучасних вебсервісів у сфері ігрових платформ та онлайн-спільнот.

SUMMARY

Bachelor’s qualification paper “Development of a Gaming Information Platform Using React.js and Node.js”: 61 pages, 54 figures, 17 references.

DATABASE, WEB APPLICATION, GAMING PLATFORM, USER INTERFACE, CLIENT-SERVER ARCHITECTURE, AUTHORIZATION SYSTEM, CONTENT MANAGEMENT, API, NODE.JS, REACT.JS.

The object of research is the process of developing a gaming platform using client-server technologies.

The purpose of the work is to develop a gaming information platform using React.js on the client side and Node.js on the server side to provide convenient access to news, user services, and administration tools.

Research methods include object-oriented programming methods, software modeling methods, and web application testing methods.

The qualification thesis presents the development of a gaming information platform focused on user interaction with content, news, and services for the gaming community. The web application was implemented using the React.js library for building a dynamic user interface and the Node.js environment for server-side logic and request processing. A user registration and authorization system, a news module, an administrator control panel, and database integration were developed. Additionally, an AI assistant based on the Google Gemini API was integrated into the platform to provide responses in Ukrainian and support interaction between users and the system through the web interface. The proposed solution ensures high performance, efficiency, and ease of use. The results of the work can serve as a foundation for creating modern web services in the field of gaming platforms and online communities.

ЗМІСТ

Завдання на кваліфікаційну роботу……..2
Реферат 4
Summary 5
Вступ..... 7
1 Визначення вимог до інформаційної системи 9
1.1 Опис предметної області 9
1.2 Огляд програмного забезпечення..... 10
1.3 Аналіз вимог 12
2 Проєктування інформаційної системи «GenTixCraft Online» 13
2.1 Концептуальне моделювання системи 13
2.2 Розробка логічної структури..... 15
2.3 Розробка бази даних..... 19
3 Програмна реалізація інформаційної системи 23
3.1 Реалізація аутентифікації та авторизації користувачів..... 23
3.2 Реалізація інтеграції вебсайту з ігровим сервером Minecraft..... 29
3.3 Реалізація панелі користувача 33
3.4 Реалізація модуля новин та магазину 37
3.5 Інтеграція AI-асистента..... 44
3.6 Реалізація панелі адміністратора для керування контентом 47
3.7 Огляд готового додатка та тестування..... 53
Висновки 59
Перелік посилань..... 60

ВСТУП

Сучасний етап розвитку інформаційних технологій характеризується стрімким зростанням ролі вебзастосунків у різних сферах діяльності людини, зокрема у сфері цифрових розваг та онлайн-спільнот. Особливого розвитку набули ігрові онлайн-платформи, які об’єднують ігрові спільноти, забезпечують доступ до новин, подій, обговорень та користувацьких сервісів. У порівнянні з традиційними статичними сайтами, сучасні вебзастосунки забезпечують динамічну взаємодію з користувачем, персоналізацію контенту та ефективну обробку великих обсягів даних. Це стало можливим завдяки використанню клієнт-серверної архітектури та сучасних JavaScript-фреймворків і серверних середовищ виконання.

Зокрема, бібліотека React.js дозволяє створювати високопродуктивні інтерфейси користувача, а середовище Node.js забезпечує ефективну обробку серверних запитів і взаємодію з базами даних. Поєднання цих технологій дозволяє розробити сучасну вебплатформу, орієнтовану на потреби користувачів та високі вимоги до швидкодії і зручності використання.

Мета дослідження – створення функціональної ігрової інформаційної платформи з використанням React.js на стороні клієнта та Node.js на стороні сервера для забезпечення зручного доступу до користувацьких сервісів.

Для досягнення поставленої мети передбачено реалізацію таких завдань:

- проаналізувати сучасні вебтехнології та засоби розробки, зокрема React.js, Node.js, Express.js, MySQL, JWT, Axios, REST API та Google Gemini API, що використовуються для створення ігрових інформаційних платформ;
- спроєктувати структуру програмної системи та бази даних;
- розробити користувацький інтерфейс із використанням React.js;
- реалізувати серверну частину застосунку на основі Node.js;
- впровадити систему реєстрації та авторизації користувачів;

– створити модулі керування контентом і новинами;

– протестувати функціональність та стабільність роботи вебплатформи.

Об’єкт дослідження – процес розробки вебплатформи для ігрових спільнот із використанням клієнт-серверних технологій.

Предмет дослідження – програмні модулі, API, серверні та клієнтські компоненти вебплатформи, реалізовані із використанням React.js, Node.js, Express.js, MySQL, JWT, Axios, Google Gemini API та RCON-взаємодії з ігровим сервером.

Для реалізації проєкту було використано мову програмування TypeScript, яка дозволяє створювати як клієнтську, так і серверну частини вебзастосунку в єдиному середовищі. Для побудови інтерфейсу користувача застосовано бібліотеку React.js, що забезпечує компонентний підхід та швидке оновлення даних. Серверну логіку реалізовано на основі Node.js із використанням REST API для обміну даними між клієнтом і сервером. Для зберігання інформації використано базу даних MySQL, яка забезпечує ефективну обробку користувацьких даних, новин і контенту платформи.

У результаті роботи створено повноцінну ігрову інформаційну платформу з системою авторизації, модулем новин, панеллю адміністратора та інтерактивним інтерфейсом користувача. Отримані результати мають практичну цінність і можуть бути використані як основа для розробки сучасних вебсервісів у сфері ігрових спільнот та інформаційних платформ.

Кваліфікаційна робота складається зі вступу, трьох розділів, висновків та переліку використаних джерел.

У першому розділі розглянуто теоретичні основи створення вебзастосунків і сучасні технології розробки інформаційних платформ. У другому розділі наведено проєктування архітектури системи, структури бази даних та основних модулів платформи. У третьому розділі описано практичну реалізацію вебплатформи з використанням React.js та Node.js, а також результати тестування.

1 ВИЗНАЧЕННЯ ВИМОГ ДО ІНФОРМАЦІЙНОЇ СИСТЕМИ

1.1 Опис предметної області

Основне призначення ігрової інформаційної платформи – забезпечення користувачів єдиним вебсередовищем для взаємодії з ігровою спільнотою, отримання актуальної інформації та використання персоналізованих сервісів. Платформа повинна надавати доступ до новин, облікових записів, статистики та профілю користувача.

Користувачами такої системи є гравці, які беруть участь в ігровому процесі, стежать за оновленнями, взаємодіють із контентом та керують власним профілем. Окрім звичайних користувачів, система передбачає наявність ролі адміністратора, яка відповідає за керування контентом, модерацію та оновлення інформації.

У перспективі можливе розширення функціоналу платформи через інтеграцію додаткових модулів, таких як рейтинги, соціальні взаємодії або ігрова аналітика.

Важливою особливістю розроблюваної системи є інтеграція ШІ-асистента у користувацький інтерфейс. Такий асистент може виконувати функції довідкової системи, допомагати користувачам орієнтуватися у можливостях платформи, надавати пояснення щодо функціоналу, а також спрощувати взаємодію із системою.

На поточному етапі розвитку цифрових сервісів існує велика кількість платформ, що підтримують ігрові спільноти. До них можна віднести офіційні ігрові портали та багатофункціональні сервіси спільнот, зокрема Hypixel [1], CurseForge [2] та Minecraft Forum [3]. Більшість схожих систем забезпечують доступ до новин, профілів користувачів та статистичних даних. Проте часто вони мають перевантажений інтерфейс, обмежений функціонал персоналізації або складну навігацію, що ускладнює взаємодію для нових користувачів.

У межах даної роботи пропонується створення вебплатформи, орієнтованої на простоту використання, сучасний інтерфейс та логічну структуру взаємодії. Система повинна забезпечувати реєстрацію та авторизацію користувачів, керування персональним профілем, відображення внутрішніх параметрів акаунта (наприклад, балансу), а також доступ до інформаційних модулів.

Таким чином, розробка сучасної, зручної та функціональної ігрової вебплатформи є актуальною задачею, що відповідає сучасним тенденціям розвитку вебтехнологій та потребам користувачів.

1.2 Огляд програмного забезпечення

Розробка ігрової інформаційної платформи потребує використання сучасних програмних засобів, що дозволяють забезпечити продуктивність, стабільність та зручність користувацького інтерфейсу. Сучасні вебзастосунки будуються на основі клієнт-серверної архітектури, яка забезпечує ефективний розподіл обчислювальних процесів та обробки даних.

Для реалізації клієнтської частини було обрано бібліотеку React.js [4]. Вона дозволяє створювати динамічні інтерфейси користувача використовуючи компонентний підхід. Така структура спрощує повторне використання елементів інтерфейсу, покращує організацію коду та забезпечує ефективне оновлення даних без перезавантаження сторінки.

Для підвищення структурованості програмного коду використано мову програмування TypeScript [5], яка забезпечує статичну типізацію. Це дозволяє зменшити кількість потенційних помилок під час розробки, покращує читабельність коду та спрощує подальший супровід системи.

Серверна частина реалізована на основі Node.js [6]. Дане середовище дозволяє ефективно обробляти HTTP-запити, керувати бізнес-логікою та забезпечувати взаємодію з базою даних, а також дозволяє обслуговувати значну кількість одночасних запитів без суттєвих втрат продуктивності.

Для організації обміну даними між клієнтом і сервером використовується REST API [7]. Такий підхід забезпечує стандартизовану взаємодію компонентів системи та спрощує структуру застосунку.

У якості системи керування базами даних обрано MySQL [8]. Реляційна модель даних дозволяє структуровано зберігати інформацію про користувачів, облікові записи, новини та інші сутності платформи.

Для забезпечення безпеки доступу до системи реалізовано механізм авторизації та автентифікації користувачів за допомогою JsonWebToken [9]. Це дозволяє обмежити доступ до персональних даних та гарантувати захист інформації.

Важливою складовою розроблюваної системи є інтеграція інтелектуального функціоналу. Для реалізації ШІ-асистента було використано хмарну платформу Google Cloud Console [10], яка надає інструменти для роботи з сучасними моделями машинного навчання. Застосування хмарної інфраструктури дозволяє забезпечити стабільність роботи сервісу та відсутність потреби у локальному розгортанні складних обчислювальних ресурсів.

У якості мовної моделі було обрано Gemini. Використання даної моделі дозволяє реалізувати функції інтерактивного асистента, який може надавати довідкову інформацію, допомагати користувачам у навігації платформою та підвищувати зручність взаємодії із системою.

Інтеграція хмарних ШІ-технологій у вебплатформу дозволяє розширити функціональні можливості системи, покращити користувацький досвід та відповідає сучасним тенденціям розвитку вебзастосунків.

Таким чином, використаний технологічний стек (React.js, TypeScript, Node.js, REST API, JsonWebToken, MySQL, Google Cloud Console, Gemini) є найоптимальнішим рішенням для створення сучасної ігрової вебплатформи, що відповідає вимогам продуктивності, безпеки та інтелектуальної взаємодії.

1.3 Аналіз вимог

Основна мета етапу аналізу вимог – встановлення чіткого розуміння потреб користувача, сформулювати очікувану поведінку системи та уникнути можливих суперечностей між замовником і виконавцем.

До функціональних вимог відносяться:

– можливість реєстрації та авторизації користувачів;
– перегляд та редагування персонального профілю;
– відображення внутрішніх параметрів акаунта (баланс, ідентифікатор тощо);
– доступ до інформаційних модулів платформи (новини, статистика);
– керування контентом через панель адміністратора (для адміністраторів).

До нефункціональних вимог відносяться:

– продуктивність – система повинна швидко реагувати на дії користувача;
– безпека – персональні дані мають бути захищені від несанкціонованого доступу;
– стабільність – платформа повинна коректно працювати при одночасному доступі декількох користувачів;
– масштабованість – архітектура повинна дозволяти розширення функціоналу;
– зручність використання – інтерфейс повинен бути інтуїтивно зрозумілим;
– кросбраузерність – коректне відображення у сучасних веббраузерах.

Отже, сформульовані вимоги визначають ключові характеристики майбутньої системи та слугують основою для її подальшого проєктування й реалізації.

2 ПРОЄКТУВАННЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ «GENTIXCRAFT ONLINE»

2.1 Концептуальне моделювання системи

На рисунку нижче наведено діаграму варіантів використання ігрової інформаційної платформи (див. рис. 2.1).

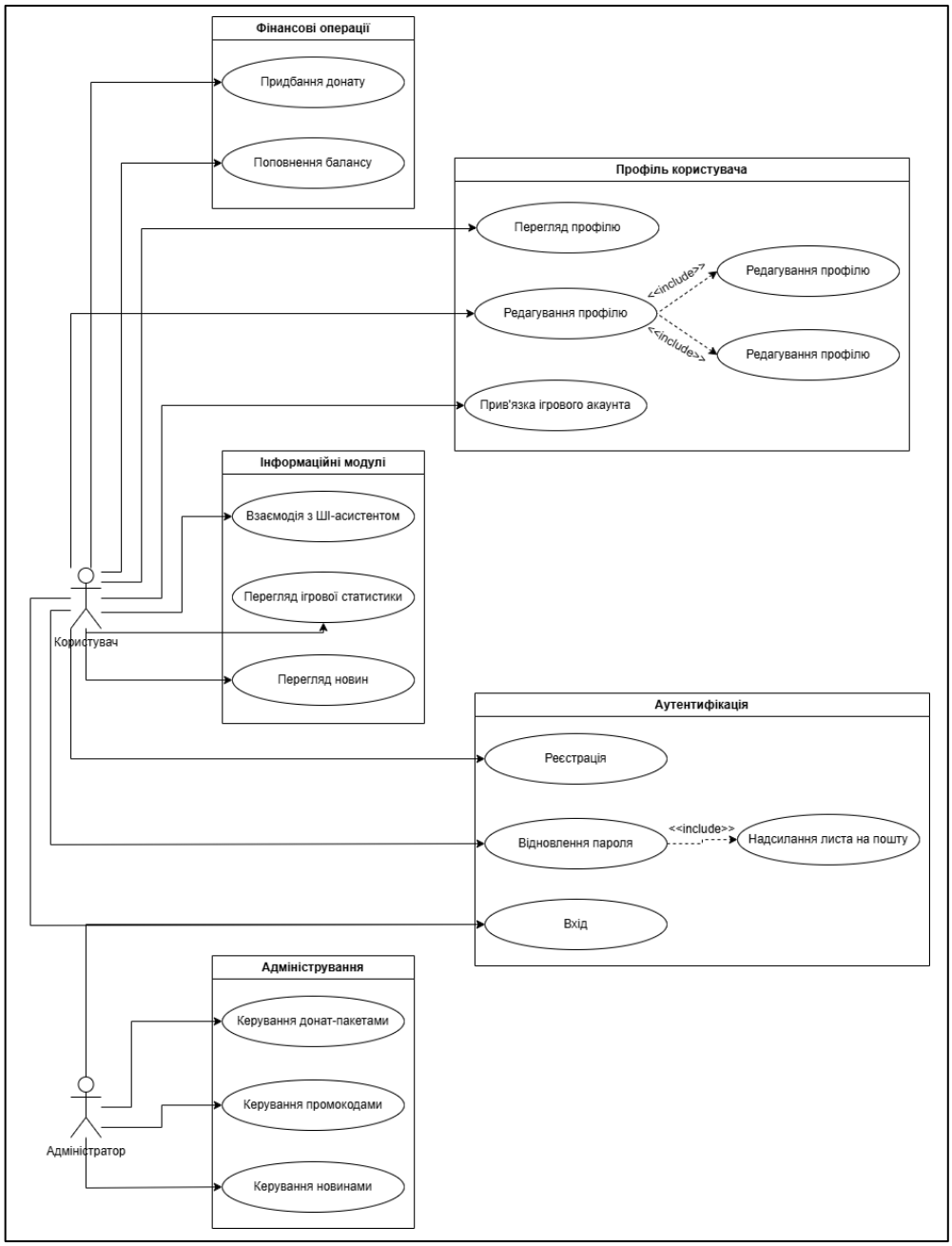


Рисунок 2.1 – Діаграма варіантів використання

На діаграмі зображено два основних актори: «Користувач» та «Адміністратор», які взаємодіють із різними функціональними модулями системи. Діаграма відображає основні сценарії використання платформи та логічний поділ функціональності на декілька підсистем: аутентифікація, профіль користувача, фінансові операції, інформаційні модулі та адміністрування.

Актор «Користувач» є основним учасником взаємодії із системою. Він має можливість користуватися більшістю функцій платформи, пов’язаних із керуванням власним профілем, переглядом інформації та виконанням фінансових операцій.

У гілці «Аутентифікація» реалізовано базові сценарії доступу до системи:

– реєстрація – створення нового облікового запису користувача;
– вхід – авторизація користувача у системі;
– відновлення пароля – відновлення доступу до акаунта у випадку втрати пароля.

Після успішної авторизації користувач отримує доступ до модуля «Профіль користувача», який містить такі варіанти використання:

– перегляд профілю – відображення інформації про користувача;
– редагування профілю – зміна персональних даних;
– прив’язка ігрового акаунта – зв’язування профілю із ігровим акаунтом.

Функція «Редагування профілю» включає додаткові сценарії редагування окремих параметрів, що на діаграмі позначено відношенням <<include>>.

Окремий модуль «Фінансові операції» відповідає за взаємодію користувача з внутрішньою економікою платформи. До нього входять такі сценарії:

– придбання донату – покупка внутрішніх ігрових пакетів або бонусів;
– поповнення балансу – додавання коштів на внутрішній рахунок користувача.

Модуль «Інформаційні модулі» забезпечує доступ користувача до інформаційних сервісів системи:

– взаємодія з ШІ-асистентом – отримання допомоги або консультацій щодо роботи платформи;
– перегляд ігрової статистики – перегляд показників та результатів гри;
– перегляд новин – ознайомлення з останніми новинами та оновленнями платформи.

Окрім користувача, у системі присутній актор «Адміністратор», який має розширені можливості керування платформою. Його функціональність представлена у модулі «Адміністрування», який включає:

– керування донат-пакетами – створення, редагування та видалення пакетів донату;
– керування промокодами – створення та контроль промокодів;
– керування новинами – публікація та редагування інформаційних матеріалів.

Таким чином, діаграма варіантів використання відображає основні сценарії взаємодії користувачів із системою та демонструє структуру функціональних можливостей платформи.

2.2 Розробка логічної структури

На етапі проєктування інформаційної системи було розроблено компонентну структуру програмного забезпечення. Для цього використано діаграму компонентів (див. рис. 2.2), яка дозволяє відобразити архітектуру системи, основні програмні модулі та взаємозв’язки між ними.

Компонентна діаграма демонструє логічний поділ системи на окремі модулі, що виконують певні функції та взаємодіють між собою через визначені інтерфейси. Такий підхід дозволяє підвищити модульність системи, спростити її підтримку та забезпечити можливість подальшого масштабування.

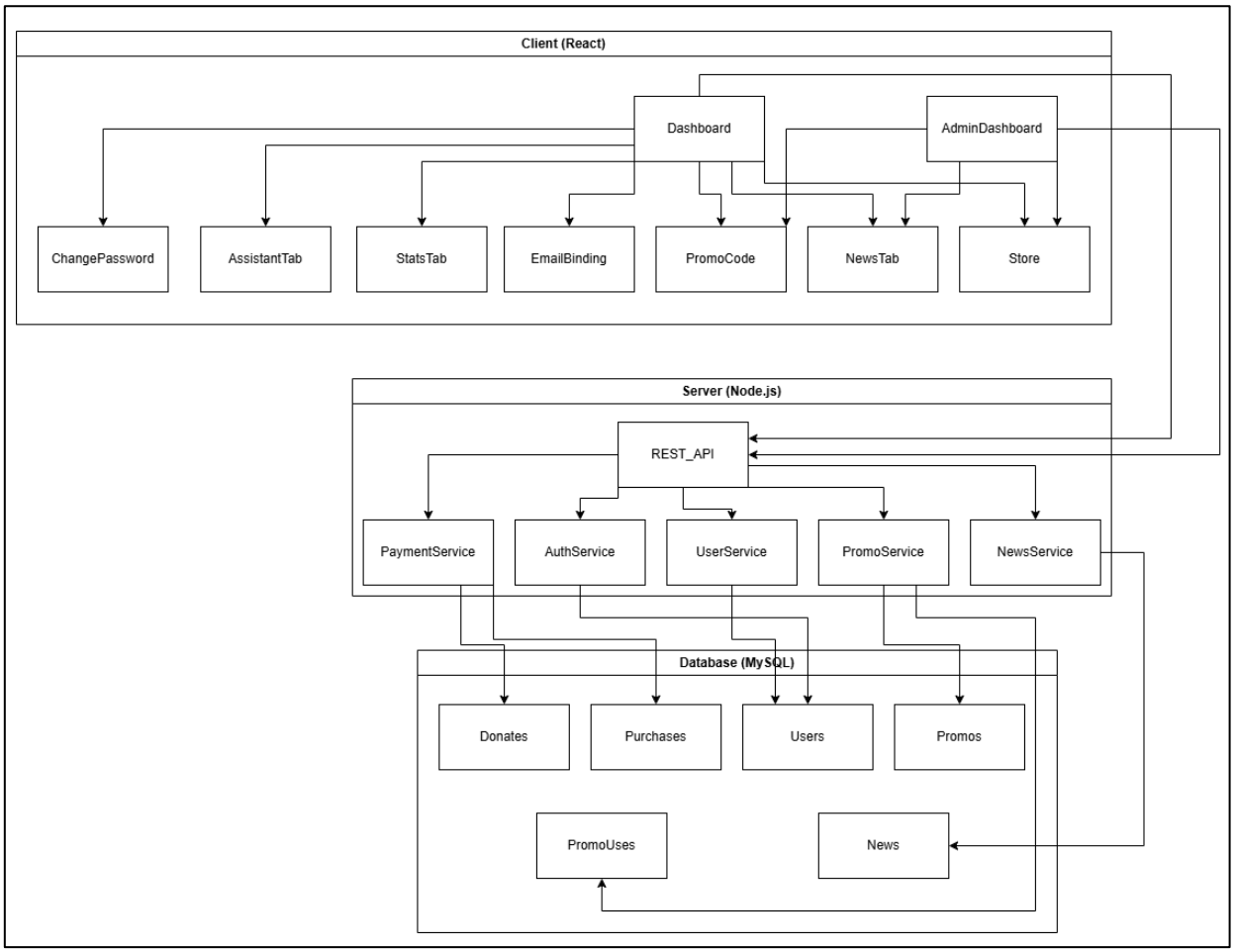


Рисунок 2.2 – Діаграма компонентів

Архітектура розробленої інформаційної системи побудована за принципом клієнт-серверної взаємодії та складається з трьох основних рівнів:

– клієнтської частини;

– серверної частини;

– системи зберігання даних.

Клієнтська частина реалізована за допомогою бібліотеки React.js та включає набір інтерфейсних компонентів, які забезпечують взаємодію користувача з системою. Основним компонентом клієнтської частини є Dashboard, що виступає центральною точкою доступу до функціоналу платформи.

До складу клієнтських компонентів входять:

– statsTab – модуль перегляду ігрової статистики користувача;

– newsTab – модуль перегляду новин ігрового сервера;

– store – модуль придбання донат-пакетів;
– promoCode – модуль активації промокодів;
– emailBinding – модуль прив’язки електронної пошти;
– changePassword – модуль зміни пароля;
– assistantTab – модуль взаємодії з AI-асистентом.

Окремим компонентом клієнтської частини є AdminDashboard, який використовується адміністраторами системи для керування контентом.

Панель адміністратора забезпечує управління такими компонентами системи:

– donate – керування донат-пакетами;
– promo – керування промокодами;
– news – керування новинами платформи.

Серверна частина системи реалізована на основі Node.js та відповідає за обробку HTTP-запитів, реалізацію бізнес-логіки, автентифікацію користувачів, а також взаємодію з базою даних.

Обмін даними між клієнтською та серверною частинами здійснюється через REST API, що забезпечує стандартизований спосіб доступу до ресурсів системи.

Для більш повного відображення архітектури інформаційної системи наведено діаграму розгортання (див. рис. 2.3), яка демонструє фізичне розміщення програмних компонентів та їх взаємодію у середовищі виконання.

У розробленій системі виділено такі основні вузли:

– клієнтський пристрій (Client Device) – персональний комп’ютер або мобільний пристрій користувача, на якому через веббраузер здійснюється доступ до системи;
– вебклієнт (React Application) – інтерфейс користувача, що виконується у браузері та забезпечує взаємодію з сервером;
– сервер застосунку (Node.js Server) – обробляє HTTP-запити, реалізує бізнес-логіку системи, автентифікацію користувачів та взаємодію з базою даних;

– база даних (MySQL Database) – забезпечує зберігання даних користувачів, транзакцій, донатів, новин та промокодів;
– зовнішні сервіси (External Services) – сторонні API, зокрема AI-асистент, який використовується для взаємодії з користувачем.

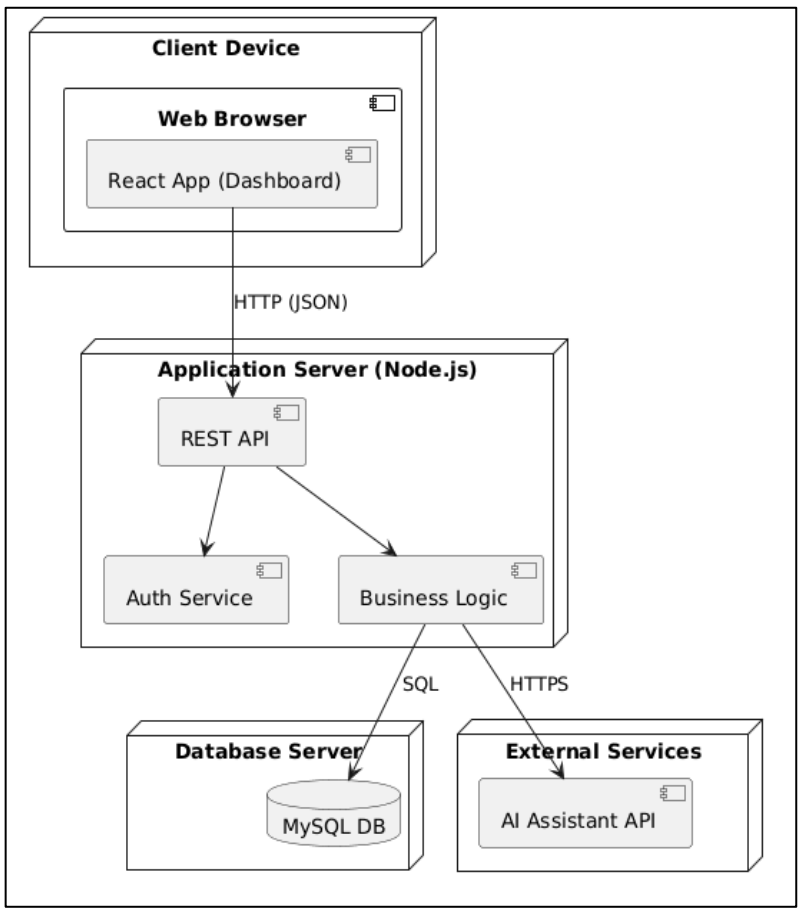


Рисунок 2.3 – Діаграма розгортання

Взаємодія між компонентами відбувається наступним чином: користувач через браузер взаємодіє з клієнтським вебзастосунком (React) → клієнтський застосунок надсилає HTTP-запити до серверу (Node.js) через REST API → сервер обробляє запити, виконує необхідну бізнес-логіку та звертається до бази даних → база даних повертає результати запитів серверу → сервер надсилає відповідь клієнту у форматі JSON → у випадку використання AI-функціоналу сервер звертається до зовнішнього API.

Така архітектура відповідає сучасним підходам до розробки вебзастосунків та забезпечує гнучкість і надійність системи.

Розроблена компонентна структура дозволяє чітко розділити функціональні обов’язки між модулями системи, що спрощує її подальший розвиток, тестування та підтримку.

2.3 Розробка бази даних

Для зберігання інформації використовується реляційна база даних MySQL, у якій зберігаються дані користувачів, донат-пакетів, покупок, промокодів та новин.

Для забезпечення зберігання та обробки даних у системі було спроєктовано реляційну базу даних. Структура бази даних представлена у вигляді ER-моделі (див. рис. 2.4), яка відображає основні сутності системи, їх атрибути та взаємозв’язки між ними.

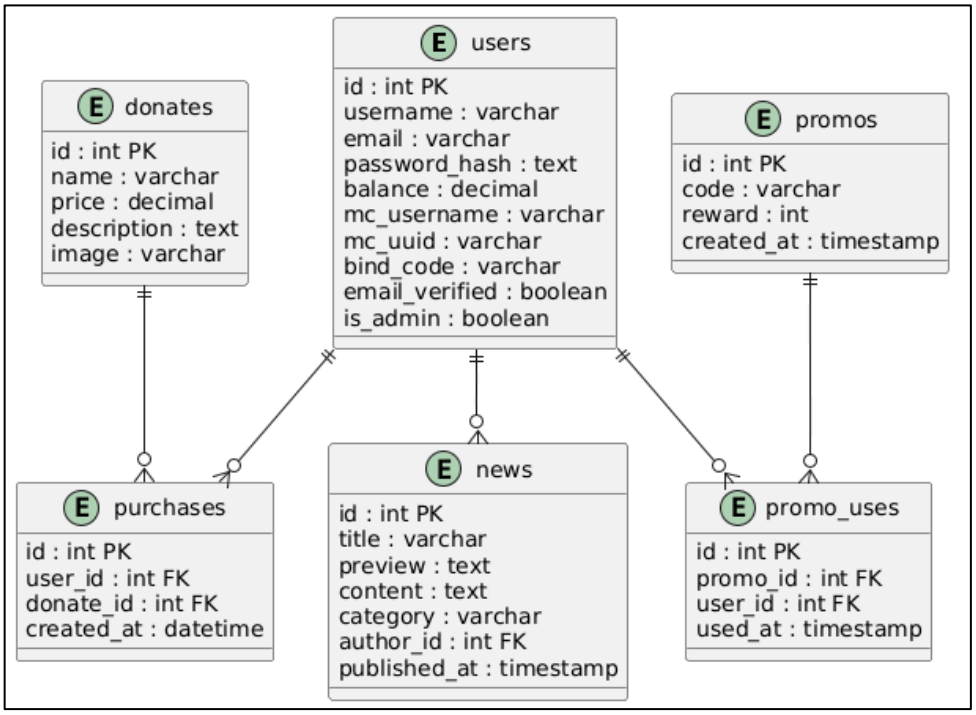


Рисунок 2.4 – Діаграма бази даних

Основними сутностями бази даних є: Users, Donates, Purchases, Promos, Promo_Uses та News.

Сутність Users зберігає інформацію про зареєстрованих користувачів платформи. Вона містить основні дані облікового запису, а також інформацію про баланс користувача та прив’язку ігрового акаунта. Основні атрибути сутності:

– id – унікальний ідентифікатор користувача;
– username – ім’я користувача;
– email – електронна адреса;
– password_hash – хешований пароль;
– balance – баланс внутрішньої валюти;
– mc_username – ігровий нікнейм користувача;
– mc_uuid – унікальний ідентифікатор ігрового акаунта;
– bind_code – код прив’язки ігрового акаунта;
– email_verified – статус підтвердження електронної пошти;
– is_admin – ознака адміністратора системи.

Сутність Users є центральною у структурі бази даних та використовується у багатьох взаємозв’язках з іншими таблицями.

Сутність Donates містить інформацію про донат-пакети, доступні для придбання користувачами. Основні атрибути:

– id – унікальний ідентифікатор пакета;
– name – назва донат-пакета;
– price – ціна пакета;
– description – опис пакета;
– image – зображення пакета.

Сутність Purchases призначена для зберігання інформації про покупки донат-пакетів користувачами. Основні атрибути:

– id – ідентифікатор покупки;
– user_id – ідентифікатор користувача;
– donate_id – ідентифікатор придбаного донат-пакета;
– created_at – дата та час здійснення покупки.

Ця таблиця реалізує зв’язок між користувачами та донат-пакетами.

Сутність Promos зберігає інформацію про промокоди, які можуть використовуватися користувачами для отримання бонусів або винагород. Основні атрибути:

– id – ідентифікатор промокоду;
– code – текст промокоду;
– reward – розмір винагороди;
– created_at – дата створення промокоду.

Сутність Promo_Uses використовується для фіксації факту використання промокоду конкретним користувачем. Основні атрибути:

– id – ідентифікатор запису;
– promo_id – ідентифікатор використаного промокоду;
– user_id – ідентифікатор користувача;
– used_at – дата використання промокоду.

Ця таблиця реалізує зв’язок між користувачами та промокодами.

Сутність News містить інформацію про новини ігрового сервера, які відображаються користувачам на вебплатформі. Основні атрибути:

– id – ідентифікатор новини;
– title – заголовок новини;
– preview – короткий опис;
– content – повний текст новини;
– category – категорія новини;
– author_id – ідентифікатор адміністратора, який створив новину;
– published_at – дата публікації.

Між сутностями бази даних існують наступні зв’язки:

– users – purchases : зв’язок типу один-до-багатьох, оскільки один користувач може здійснити багато покупок;
– donates – purchases : зв’язок один-до-багатьох, оскільки один донат-пакет може бути придбаний багатьма користувачами;
– users – promo_uses : зв’язок один-до-багатьох, оскільки користувач може використати кілька промокодів;

– promos – promo_uses : зв’язок один-до-багатьох, оскільки один промокод може бути використаний багатьма користувачами;
– users – news : зв’язок один-до-багатьох, оскільки один адміністратор може створювати кілька новин.

Таким чином, розроблена ER-модель забезпечує структуроване зберігання даних та підтримує основний функціонал інформаційної системи, включаючи реєстрацію користувачів, придбання донат-пакетів, використання промокодів та публікацію новин.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ

3.1 Реалізація аутентифікації та авторизації користувачів

Одним із ключових модулів розробленої інформаційної системи є підсистема автентифікації та авторизації користувачів. Її основне призначення полягає у забезпеченні контрольованого доступу до функціоналу платформи, захисті персональних даних та підтримці механізму ідентифікації користувача під час роботи із захищеними ресурсами.

Серверна частина реалізована за допомогою середовища Node.js та фреймворку Express.js [11]. Для роботи з акаунтами використовується окремий маршрутизатор, який реалізує операції реєстрації, входу, перевірки сесії, зміни пароля, підтвердження електронної пошти та відновлення доступу.

Під час реєстрації виконується перевірка унікальності імені користувача. Якщо запис з таким ім’ям вже існує, сервер повертає повідомлення про помилку.

Після проходження перевірки пароль не зберігається у відкритому вигляді. Для цього використовується бібліотека bcryptjs, яка виконує хешування пароля перед записом у базу даних. Розглянемо реалізацію маршруту реєстрації (див. рис. 3.1).

```
router.post("/register", async (req, res) => {
  const { username, password } = req.body;

  const [users] = await db.query("SELECT * FROM users WHERE username = ?", [username]);
  if (users.length > 0) return res.status(400).json({ message: "Нік вже зайнятий" });

  const hash = await bcrypt.hash(password, 10);
  await db.query("INSERT INTO users (username, password_hash) VALUES (?, ?)", [username, hash]);

  res.json({ message: "Успішно зареєстровано" });
});
```

Рисунок 3.1 – Код маршруту реєстрації

Алгоритм роботи цього маршруту складається з таких етапів:

а) отримання username та password із тіла запиту;

б) перевірка наявності користувача в таблиці users;

в) генерація хешу за допомогою bcrypt.hash(password, 10);

г) запис нового користувача в базу даних.

У результаті до таблиці users зберігаються лише такі основні дані:

– ім’я користувача;

– хешований пароль;

– службова інформація акаунта.

Такий підхід підвищує безпеку системи навіть у випадку компрометації бази даних.

Після введення логіна та пароля виконується перевірка облікових даних. Сервер спочатку знаходить користувача за полем username, після чого порівнює введений пароль із збереженим хешем. Розглянемо реалізацію маршруту входу до системи (див. рис. 3.2).

```
router.post("/login", async (req, res) => {
  const { username, password } = req.body;

  const [users] = await db.query("SELECT * FROM users WHERE username = ?", [username]);
  if (users.length === 0) return res.status(400).json({ message: "Користувача не знайдено" });

  const user = users[0];
  const valid = await bcrypt.compare(password, user.password_hash);
  if (!valid) return res.status(401).json({ message: "Невірний пароль" });

  const token = jwt.sign({ id: user.id, username: user.username }, JWT_SECRET, { expiresIn: "7d" });

  res.json({ message: "Успішний вхід", token });
});
```

Рисунок 3.2 – Код маршруту авторизації

У випадку успішної перевірки формується JWT-токен. Створений токен містить:

– унікальний ідентифікатор користувача;

– ім’я користувача;

– строк дії токена.

Надалі цей токен використовується для доступу до всіх захищених маршрутів. Для перевірки валідності токена використовується проміжне програмне забезпечення authMiddleware.

Після проходження перевірки користувач отримує доступ до маршруту GET /auth/me. Розглянемо реалізацію отримання інформації про поточного користувача (див. рис. 3.3).

```
router.get("/me", authMiddleware, async (req, res) => {
  const [users] = await db.query(
    "SELECT id, username, balance, mc_username, mc_uuid, email, email_verified FROM users WHERE id = ?",
    [req.user.id]
  );
  if (users.length === 0) return res.status(404).json({ message: "Користувача не знайдено" });

  res.json(users[0]);
});

const fetch = (...args) => import('node-fetch').then(({ default: fetch }) => fetch(...args));
```

Рисунок 3.3 – Код маршруту для отримання даних користувача

У цьому маршруті виконується SQL-запит до таблиці users, після чого повертаються основні дані профілю: id; username; balance; mc_username; mc_uuid; email; email_verified. Саме цей маршрут використовується після авторизації для формування особистого кабінету користувача.

Для підвищення безпеки системи реалізовано механізм зміни пароля без повторної авторизації. Розглянемо реалізацію зміни пароля (див. рис. 3.4).

Принцип роботи модуля:

а) користувач передає поточний пароль і новий пароль;

б) сервер отримує з бази даних поточний password_hash;

в) виконується перевірка через bcrypt.compare();

г) якщо перевірка успішна – новий пароль хешується;

д) оновлений хеш записується у таблицю users.

Таким чином виключається можливість зміни пароля без знання поточного значення.

```
router.post('/change-password', authMiddleware, async (req, res) => {
  const userId = req.user.id;
  const { currentPassword, newPassword } = req.body;

  if (!currentPassword || !newPassword) {
    return res.status(400).json({ message: 'Обидва поля обов\'язкові' });
  }

  try {
    const [users] = await db.query('SELECT password_hash FROM users WHERE id = ?', [userId]);

    if (users.length === 0) {
      return res.status(404).json({ message: 'Користувача не знайдено' });
    }

    const isValid = await bcrypt.compare(currentPassword, users[0].password_hash);
    if (!isValid) {
      return res.status(401).json({ message: 'Невірний поточний пароль' });
    }

    const newHash = await bcrypt.hash(newPassword, 10);
    await db.query('UPDATE users SET password_hash = ? WHERE id = ?', [newHash, userId]);

    res.json({ message: 'Пароль успішно змінено' });
  } catch (err) {
    console.error('Помилка при зміні пароля:', err);
    res.status(500).json({ message: 'Помилка сервера' });
  }
});
```

Рисунок 3.4 – Код маршруту для зміни пароля

Для доступу до окремих функцій системи використовується механізм підтвердження електронної пошти. На першому етапі генерується код підтвердження та надсилається користувачу через Nodemailer. Розглянемо реалізацію надсилання коду підтвердження (див. рис. 3.5).

У базі даних оновлюються поля:

– email;

– email_verified;

– email_verification_code.

Після цього код відправляється на електронну адресу користувача.

Другий етап полягає у перевірці введеного коду. Розглянемо реалізацію підтвердження електронної пошти (див. рис. 3.6).

Після успішної перевірки:

– поле email_verified набуває значення TRUE;

– код підтвердження видаляється.

```
router.post('/send-email-code', authMiddleware, async (req, res) => {
  const userId = req.user.id;
  const { email } = req.body;

  if (!email) return res.status(400).json({ message: 'Email обов\'язковий' });

  try {
    const code = crypto.randomBytes(3).toString('hex').toUpperCase();

    await db.query(
      'UPDATE users SET email = ?, email_verified = FALSE, email_verification_code = ? WHERE id = ?',
      [email, code, userId]
    );

await transporter.sendMail({
  from: process.env.EMAIL_FROM,
  to: email,
  subject: 'Код підтвердження email',
  text: `Ваш код підтвердження: ${code}`,
});

    res.json({ message: 'Код надіслано на пошту' });
  } catch (err) {
    console.error('Помилка відправки пошти:', err);
    res.status(500).json({ message: 'Помилка при відправці коду' });
  }
});
```

Рисунок 3.5 – Програмна реалізація надсилання листа з кодом підтвердження

```
router.post('/verify-email-code', authMiddleware, async (req, res) => {
  const userId = req.user.id;
  const { email, code } = req.body;

  if (!email || !code) return res.status(400).json({ message: 'Email та код обов\'язкові' });

  try {
    const [users] = await db.query(
      'SELECT * FROM users WHERE id = ? AND email = ? AND email_verification_code = ?',
      [userId, email, code]
    );

    if (users.length === 0) {
      return res.status(400).json({ message: 'Невірний код підтвердження' });
    }

    await db.query(
      'UPDATE users SET email_verified = TRUE, email_verification_code = NULL WHERE id = ?',
      [userId]
    );

    res.json({ message: 'Email успішно підтверджено' });
  } catch (err) {
    console.error('Помилка підтвердження email:', err);
    res.status(500).json({ message: 'Помилка сервера' });
  }
});
```

Рисунок 3.6 – Програмна реалізація маршруту для верифікації електронної пошти

Цей механізм використовується як додатковий рівень контролю доступу до функцій прив’язки ігрового акаунта.

Окремо реалізовано механізм відновлення доступу у випадку втрати пароля. На першому етапі користувач вводить адресу електронної пошти, після чого сервер:

а) перевіряє наявність користувача;

б) генерує тимчасовий JWT-токен;

в) записує токен і строк його дії в базу даних;

г) формує посилання для скидання пароля.

Розглянемо реалізацію надсилання посилання на відновлення пароля (див. рис. 3.7).

```
router.post('/forgot-password', async (req, res) => {
  const { email } = req.body;
  if (!email) return res.status(400).json({ message: 'Email обов\'язковий' });

  try {
    const [users] = await db.query('SELECT * FROM users WHERE email = ?', [email]);
    if (users.length === 0) return res.status(404).json({ message: 'Користувача не знайдено' });

    const user = users[0];
    const token = jwt.sign({ id: user.id }, JWT_SECRET, { expiresIn: '24h' });

    await db.query('UPDATE users SET reset_token = ?, reset_token_exp = DATE_ADD(NOW(), INTERVAL 24 HOUR) WHERE id = ?', [token, user.id]);

    const resetLink = `https://gentixcraft-online.onrender.com/reset-password?token=${token}`;
    await transporter.sendMail({
      from: process.env.EMAIL_FROM,
      to: email,
      subject: 'Скидання пароля',
      text: `Щоб скинути пароль, перейдіть за посиланням: ${resetLink}`
    });

    res.json({ message: 'Посилання для скидання пароля надіслано на пошту' });
  } catch (err) {
    console.error('Помилка forgot-password:', err);
    res.status(500).json({ message: 'Помилка сервера' });
  }
});
```

Рисунок 3.7 – Код маршруту для посилання на відновлення пароля

На другому етапі користувач передає новий пароль разом з токеном. Розглянемо реалізацію скидання пароля (див. рис. 3.8).

Після перевірки токена сервер:

– хешує новий пароль;

– оновлює password_hash;

– очищує поля reset_token та reset_token_exp.

```
router.post('/reset-password', async (req, res) => {
  const { token, newPassword } = req.body;

  if (!token || !newPassword) {
    return res.status(400).json({ message: 'Токен та новий пароль обов\'язкові' });
  }

  try {
    const decoded = jwt.verify(token, JWT_SECRET);

    const [users] = await db.query(
      'SELECT * FROM users WHERE id = ? AND reset_token = ? AND reset_token_exp > NOW()',
      [decoded.id, token]
    );

    if (users.length === 0) {
      return res.status(400).json({ message: 'Невірний або прострочений токен' });
    }

    const newHash = await bcrypt.hash(newPassword, 10);

    await db.query(
      'UPDATE users SET password_hash = ?, reset_token = NULL, reset_token_exp = NULL WHERE id = ?',
      [newHash, decoded.id]
    );

    res.json({ message: 'Пароль успішно скинуто' });
  } catch (err) {
    console.error('Помилка reset-password:', err);
    res.status(400).json({ message: 'Невірний або прострочений токен' });
  }
});
```

Рисунок 3.8 – Код маршруту для скидання пароля

Отже, реалізована підсистема автентифікації забезпечує:

– реєстрацію нових користувачів;

– захищений вхід до системи;

– перевірку валідності сесії;

– зміну пароля;

– підтвердження електронної пошти;

– відновлення доступу до облікового запису.

Використання JWT, bcrypt, middleware перевірки токенів та механізму поштового підтвердження забезпечує достатній рівень безпеки та відповідає сучасним технологіям розробки вебзастосунків.

3.2 Реалізація інтеграції вебсайту з ігровим сервером Minecraft

Однією з ключових функцій розробленої системи є інтеграція вебзастосунку з ігровим сервером Minecraft. Основним призначенням даного

механізму є автоматична прив’язка ігрового акаунта до облікового запису користувача на вебсайті. Реалізація інтеграції дозволяє синхронізувати дані між вебсистемою та ігровим сервером.

Для реалізації взаємодії було розроблено окремий плагін для сервера Minecraft мовою Java із використанням Bukkit API [12]. Плагін забезпечує обробку спеціальної команди «/bind», яку користувач виконує безпосередньо у грі.

Принцип роботи механізму прив’язки складається з декількох етапів:

а) авторизація користувача на вебсайті;

б) генерація унікального коду прив’язки через API;

в) виконання команди «/bind <код>» у грі;

г) оновлення інформації користувача у базі даних.

На серверній стороні вебзастосунку формується випадковий код прив’язки, який зберігається у таблиці «users» у полі «bind_code». Даний код використовується для підтвердження того, що власник акаунта на сайті дійсно володіє відповідним ігровим профілем Minecraft.

Після введення команди «/bind» плагін отримує:

– код прив’язки;

– ім’я гравця;

– UUID Minecraft-акаунта.

Далі виконується HTTP POST-запит до REST API вебзастосунку.

Після виконання команди виконується перевірка аргументів та формування асинхронного HTTP-запиту до API сервера (див. рис. 3.9).

У даному фрагменті використовується асинхронне виконання через runTaskAsynchronously(), що дозволяє уникнути блокування основного потоку Minecraft-сервера під час мережевого запиту.

Передача даних виконується через HttpURLConnection. Запит містить JSON-об’єкт із кодом прив’язки, UUID гравця та його нікнеймом (див. рис. 3.10). Передача даних у форматі JSON дозволяє легко інтегрувати Minecraft-сервер із REST API Node.js-застосунку.

```java
@Override
public boolean onCommand(CommandSender sender, Command command, String label, String[] args) {
    if (!(sender instanceof Player)) {
        sender.sendMessage("Тільки гравці можуть виконувати цю команду.");
        return true;
    }

    Player player = (Player) sender;

    if (args.length != 1) {
        player.sendMessage("Використання: /bind <код>");
        return true;
    }

    String bindCode = args[0];
    String playerName = player.getName();
    String playerUUID = player.getUniqueId().toString();

    player.sendMessage("Перевірка коду прив'язки...");

    Bukkit.getScheduler().runTaskAsynchronously(this, () -> {
        try {
            boolean success = sendBindRequest(bindCode, playerUUID, playerName);
            if (success) {
                player.sendMessage("Акаунт успішно доєднано до сайту!");
            } else {
                player.sendMessage("Некоректний код прив'язки або помилка на сайті.");
            }
        } catch (Exception e) {
            player.sendMessage("Виникла помилка при перевірці коду.");
            getLogger().severe("Помилка при відправці запиту прив'язки: " + e.getMessage());
        }
    });

    return true;
}
```

Рисунок 3.9 – Реалізація команди прив’язки у плагіні

```java
String jsonInputString = String.format(
    "{\"bindCode\":\"%s\", \"mcUuid\":\"%s\", \"playerName\":\"%s\"}",
    bindCode, uuid, playerName
);

byte[] input = jsonInputString.getBytes(StandardCharsets.UTF_8);

conn.setRequestProperty("Content-Type", "application/json");
conn.setRequestProperty("Content-Length", String.valueOf(input.length));
conn.setRequestProperty("Accept", "application/json");
conn.setDoOutput(true);

try (OutputStream os = conn.getOutputStream()) {
    os.write(input, 0, input.length);
}
```

Рисунок 3.10 – Реалізація формування HTTP-запиту до вебзастосунку

Після отримання запиту сервер Node.js виконує перевірку коду прив’язки та оновлює інформацію користувача у базі даних (див. рис. 3.11).

```
router.post("/confirm-bind", async (req, res) => {
  console.log("req.body:", req.body);
  const { bindCode, mcUuid, playerName } = req.body;

  if (!bindCode || !mcUuid || !playerName) {
    return res.status(400).json({ message: "Відсутні необхідні параметри" });
  }

  try {
    const [users] = await db.query(
      "SELECT * FROM users WHERE bind_code = ? AND bind_confirmed = FALSE",
      [bindCode]
    );

    if (users.length === 0) {
      return res.status(400).json({ message: "Невірний або вже використаний код прив'язки" });
    }

    const user = users[0];

    await db.query(
      "UPDATE users SET mc_uuid = ?, mc_username = ?, bind_confirmed = TRUE WHERE id = ?",
      [mcUuid, playerName, user.id]
    );

    res.status(200).json({ message: "Прив'язка успішна" });

  } catch (err) {
    console.error("Помилка в confirm-bind:", err);
    res.status(500).json({ message: "Помилка сервера при прив'язці" });
  }
});
```

Рисунок 3.11 – Реалізація обробки запиту на стороні вебдодатка

У разі успішної перевірки:

– у таблицю users записується UUID Minecraft-акаунта;

– зберігається нікнейм гравця;

– поле bind_confirmed змінюється на значення TRUE.

Після цього обліковий запис користувача вважається синхронізованим із Minecraft-сервером.

Розроблений механізм інтеграції має декілька важливих особливостей:

– використання UUID Minecraft для унікальної ідентифікації гравця;

– асинхронна обробка мережевих запитів;

– автоматичне оновлення інформації у базі даних;

– використання REST API для взаємодії між компонентами системи.

Завдяки реалізованій архітектурі вебзастосунок та ігровий сервер функціонують як єдина система, що забезпечує централізоване управління акаунтами користувачів та їх ігровими даними.

3.3 Реалізація панелі користувача

Панель користувача є центральною частиною вебзастосунку, через яку виконується взаємодія між користувачем, серверною частиною системи, базою даних та зовнішніми сервісами. Саме в межах даного модуля реалізовано керування акаунтом, отримання ігрової статистики, систему безпеки, новинний модуль, магазин привілеїв та інтеграцію ШІ-помічника.

Архітектурно інтерфейс реалізований як багатокомпонентний React-модуль із використанням функціональних компонентів, хуків useState та useEffect, а також маршрутизації через react-router-dom.

Основний компонент Dashboard виконує роль контейнера для всіх вкладок системи (див. рис. 3.12).

```
export default function Dashboard() {
  const [user, setUser] = useState(null);
  const [bindCode, setBindCode] = useState(null);
  const [history, setHistory] = useState([]);
  const [cardNumber, setCardNumber] = useState('');
  const [expiry, setExpiry] = useState('');
  const [cvc, setCvc] = useState('');
  const [isTopUpModalOpen, setIsTopUpModalOpen] = useState(false);
  const [topUpAmount, setTopUpAmount] = useState(0);
  const [mcUUID, setMcUUID] = useState(null);
  const [bwStats, setBwStats] = useState(null);
  const location = useLocation();
  const [isSidebarOpen, setIsSidebarOpen] = useState(false);
  const [activeTab, setActiveTab] = useState(
  localStorage.getItem("activeTab") || "home"
  );
  const navigate = useNavigate();

  useEffect(() => {
  localStorage.setItem("activeTab", activeTab);
}, [activeTab]);
```

Рисунок 3.12 – Реалізація модуля панелі користувача

У даному блоці ініціалізуються основні стани застосунку:

– user – містить інформацію про авторизованого користувача;

– bindCode – код прив’язки Minecraft-акаунта;

– history – історія покупок;

– activeTab – активна вкладка панелі користувача.

Для збереження активної вкладки після перезавантаження сторінки використовується localStorage.

Одним із ключових механізмів роботи панелі є автоматичне отримання інформації про користувача після авторизації. Для цього реалізовано окрему функцію fetchUser (див. рис. 3.13).

```
const fetchUser = async () => {
    try {
      const token = localStorage.getItem("token");
const res = await API.get('/auth/me', {
  headers: { Authorization: `Bearer ${token}` }
});
      setUser(res.data);

      if (res.data.email_verified) {
        if (!res.data.mc_uuid && !res.data.bind_code) {
          const codeRes = await API.post('/auth/generate-bind-code', {}, {
            headers: { Authorization: `Bearer ${localStorage.getItem("token")}` }
          });
          setBindCode(codeRes.data.bindCode);
        } else {
          setBindCode(res.data.bind_code);
        }
      } else {
        setBindCode(null);
      }
    } catch {
      localStorage.removeItem("token");
      navigate('/login');
    }
  };

  useEffect(() => {
    fetchUser();
  }, []);
```

Рисунок 3.13 – Реалізація отримання інформації про користувача

Даний механізм реалізує одразу декілька критично важливих функцій:

а) перевірку JWT-токена;

б) отримання інформації про користувача;

в) автоматичне створення коду прив’язки Minecraft-акаунта;

г) контроль підтвердження електронної пошти;

д) автоматичний вихід із системи при помилці авторизації.

Таким чином забезпечується цілісність сесії користувача та контроль доступу до функціоналу системи.

Після успішного завантаження профілю виконується відображення основної інформації про акаунт (див. рис. 3.14).

```
<div className="rightbar">
  <h2>Ваш профіль</h2>
  <div style={{ display: 'grid', gridTemplateColumns: 'repeat(auto-fit, minmax(250px, 1fr))', gap: '20px' }}>

    <div className="cardgroup1">
    <div className="card">
      <h3>Дані акаунта</h3>
      <p><strong>Нікнейм:</strong> {user.mc_username}</p>
      <p><strong>UUID:</strong> {user.mc_uuid || mcUUID || 'Не прив'язано'}</p>
      {!user.mc_uuid && !mcUUID && (
user.email_verified ? (
  <p>Для прив'язки використайте команду: <code>/bind {bindCode}</code></p>
) : (
  <p>Щоб прив'язати акаунт Minecraft, спершу підтвердіть вашу пошту</p>
)
)}
    </div>
```

Рисунок 3.14 – Відображення інформації про акаунт

Даний компонент реалізує рендеринг елементів інтерфейсу. Якщо акаунт Minecraft ще не прив'язаний – система відображає код прив'язки. Якщо пошта не підтверджена – користувач отримує про це повідомлення.

Окремим важливим модулем є система поповнення внутрішнього балансу. Для реалізації механізму оплати використовується модальне вікно, що відкривається після вибору суми поповнення (див. рис. 3.15).

Реалізація даного механізму дозволяє:

– зменшити кількість некоректних запитів до сервера;

– виконувати первинну валідацію на стороні клієнта;

– покращити користувацький досвід;

– зменшити навантаження на API.

Після проходження перевірки виконується запит на сервер (див. рис. 3.16).

```
  const openTopUpModal = (amount) => {
  setTopUpAmount(amount);
  setIsTopUpModalOpen(true);
};

const confirmTopUp = async () => {
  const token = localStorage.getItem("token");

  const safeCardNumber = (cardNumber || '').replace(/\s+/g, '');
  const safeExpiry = expiry || '';
  const safeCvc = cvc || '';

  if (!safeCardNumber || safeCardNumber.length !== 16) {
    toast.error("Некоректний номер картки");
    return;
  }
  if (!safeExpiry || !/^\d{2}\/\d{2}$/.test(safeExpiry)) {
    toast.error("Некоректний термін дії картки");
    return;
  }
  if (!safeCvc || safeCvc.length !== 3) {
    toast.error("Некоректний CVC");
    return;
  }
```

Рисунок 3.15 – Реалізація системи поповнення балансу

```
try {
  const res = await API.post("/store/topup", {
    amount: topUpAmount,
    cardNumber: safeCardNumber,
    expiry: safeExpiry,
    cvc: safeCvc
  }, {
    headers: { Authorization: `Bearer ${token}` },
  });

  toast.success(res.data.message || `Баланс поповнено на ${topUpAmount} GTC`);
  setIsTopUpModalOpen(false);
  fetchUser();
} catch (err) {
  console.error(err);
  toast.error(err.response?.data?.message || "Помилка поповнення");
  setIsTopUpModalOpen(false);
}
};
```

Рисунок 3.16 – Відправка запиту на сервер при поповненні балансу користувачем

У випадку успішного поповнення система автоматично оновлює баланс користувача через повторне виконання fetchUser().

Також у панелі користувача реалізовано систему прив’язки електронної пошти (див. рис. 3.17).

```
const sendCode = async () => {
  try {
    await API.post('/auth/send-email-code', { email }, {
      headers: { Authorization: `Bearer ${token}` }
    });
    toast.success('Код відправлено на пошту');
    setStep(2);
  } catch {
    toast.error('Помилка при відправці коду');
  }
};

const verifyCode = async () => {
  try {
    await API.post('/auth/verify-email-code', { email, code }, {
      headers: { Authorization: `Bearer ${token}` }
    });
    toast.success('Пошту підтверджено');
    setStep(3);
    refreshUser();
  } catch {
    toast.error('Некоректний код підтвердження');
  }
};
```

Рисунок 3.17 – Реалізація системи прив’язки електронної пошти

Даний механізм використовується як додатковий рівень захисту акаунта та необхідний для прив’язки внутрішньоігрового профілю.

Таким чином панель користувача виконує роль єдиного центру взаємодії між усіма підсистемами проєкту. Реалізована архітектура забезпечує модульність та інтеграцію із зовнішніми сервісами та зручне розширення функціоналу в майбутньому.

3.4 Реалізація модуля новин та магазину

Модуль новин використовується для інформування користувачів про оновлення сервера, події, патчі та нові функції системи. Реалізація даного компонента побудована на основі REST API та взаємодії клієнтської частини із базою даних. Архітектурно модуль складається з:

– серверного API для роботи з новинами;
– таблиці news у базі даних;
– React-компонентів для відображення контенту;
– системи фільтрації та пагінації.

Основною функцією модуля є отримання списку новин (див. рис. 3.18).

```
const express = require('express');
const router = express.Router();
const db = require('../db');

router.get('/', async (req, res) => {
  const { category } = req.query;

  try {
    const [rows] = await db.execute(
      category
        ? 'SELECT * FROM news WHERE category = ? ORDER BY published_at DESC'
        : 'SELECT * FROM news ORDER BY published_at DESC',
      category ? [category] : []
    );
    res.json(rows);
  } catch (err) {
    res.status(500).json({ error: 'Помилка при отриманні новин' });
  }
});
```

Рисунок 3.18 – Реалізація отримання списку новин

У даному блоці реалізовано:

– отримання новин із бази даних;

– сортування за датою публікації;

– фільтрацію за категоріями;

– обробку помилок сервера.

Особливістю реалізації є динамічне формування SQL-запиту. Якщо користувач передає категорію – система автоматично виконує вибірку лише відповідних новин.

Для отримання конкретної статті використовується окремий API-маршрут (див. рис. 3.19).

```
router.get('/:id', async (req, res) => {
  const { id } = req.params;

  try {
    const [rows] = await db.execute('SELECT * FROM news WHERE id = ?', [id]);
    if (rows.length === 0) return res.status(404).json({ error: 'Статтю не знайдено' });
    res.json(rows[0]);
  } catch {
    res.status(500).json({ error: 'Помилка при отриманні статті' });
  }
});
```

Рисунок 3.19 – Реалізація отримання конкретної статті

Даний механізм використовується при переході користувача до сторінки повної новини. Пошук статті виконується за унікальним ідентифікатором id, що мінімізує навантаження на базу даних та забезпечує швидке отримання контенту.

Для створення нових новин реалізовано окремий POST-маршрут (див. рис. 3.20).

```
router.post('/', async (req, res) => {
  const { title, content, preview, category } = req.body;

  if (!title || !content || !preview || !category)
    return res.status(400).json({ error: 'Заповніть усі поля' });

  try {
    await db.execute(
      'INSERT INTO news (title, content, preview, category) VALUES (?, ?, ?, ?)',
      [title, content, preview, category]
    );
    res.json({ message: 'Статтю додано' });
  } catch {
    res.status(500).json({ error: 'Помилка при додаванні статті' });
  }
});
```

Рисунок 3.20 – Реалізація додавання нової статті

Перед записом новини виконується перевірка всіх обов’язкових полів. Це дозволяє уникнути появи некоректних записів у базі даних. Для адміністрування новин реалізовано механізм видалення статей (див. рис. 3.21).

```
router.delete('/:id', async (req, res) => {
  try {
    await db.execute('DELETE FROM news WHERE id = ?', [req.params.id]);
    res.json({ message: 'Статтю видалено' });
  } catch {
    res.status(500).json({ error: 'Помилка при видаленні статті' });
  }
});
```

Рисунок 3.21 – Реалізація видалення статті

Такий підхід дозволяє реалізувати повноцінну систему керування контентом усередині вебзастосунку.

Для навігації між категоріями використовується механізм фільтрації (див. рис. 3.22).

```
useEffect(() => {

  let news = articles;
  if (selectedCategory !== "Всі") {
    news = articles.filter((a) => a.category === selectedCategory);
  }
  setFiltered(news);
}, [selectedCategory, articles]);
```

Рисунок 3.22 – Реалізація фільтрації новин

Дана реалізація дозволяє уникнути повторних запитів до сервера та прискорює роботу інтерфейсу.

Також реалізовано систему пагінації (див. рис. 3.23).

```
const totalPages = Math.ceil(filtered.length / perPage);
const startIndex = (page - 1) * perPage;
const currentNews = filtered.slice(startIndex, startIndex + perPage);
```

Рисунок 3.23 – Реалізація пагінації новин

Пагінація використовується для оптимізації відображення великої кількості новин та зменшення навантаження на браузер.

Таким чином модуль новин забезпечує централізоване керування інформаційним контентом проєкту та підтримує масштабовану архітектуру взаємодії між клієнтом і сервером.

Внутрішній магазин реалізований як окремий серверний модуль, що забезпечує:

– відображення доступних привілеїв;
– придбання донатів;
– видачу прав на Minecraft-сервері;
– систему поповнення балансу;
– підтримку промокодів;
– історію покупок користувача.

Основою роботи магазину є взаємодія між:

– React-клієнтом;
– Node.js API;

– базою даних MySQL;

– Minecraft-сервером через RCON.

Для отримання списку доступних привілеїв використовується GET-маршрут (див. рис. 3.24).

```
router.get("/", async (req, res) => {
  const [donates] = await db.query("SELECT * FROM donates");
  res.json(donates);
});
```

Рисунок 3.24 – Реалізація отримання списку привілеїв

У таблиці donates зберігаються:

– назва привілею;

– вартість;

– опис;

– зображення.

Отримані дані використовуються для автоматичного формування каталогу магазину на клієнтській стороні.

Найбільш складним механізмом є система покупки донату (див. рис. 3.25).

```
router.post("/buy", authMiddleware, async (req, res) => {
  const { donateId } = req.body;

  try {
    const [[donate]] = await db.query("SELECT * FROM donates WHERE id = ?", [donateId]);
    if (!donate) return res.status(404).json({ message: "Донат не знайдено" });
```

Рисунок 3.25 – Реалізація маршруту для придбання привілеїв

На початковому етапі виконується:

– перевірка авторизації;

– отримання інформації про товар;

– перевірка існування привілею.

Після цього система отримує інформацію про баланс користувача (див. рис. 3.26).

```
const [[user]] = await db.query("SELECT balance, mc_username FROM users WHERE id = ?", [req.user.id]);
const userBalance = Number(user.balance);
const donatePrice = Number(donate.price);

if (userBalance < donatePrice) {
  return res.status(400).json({ message: "Недостатньо коштів" });
}
```

Рисунок 3.26 – Реалізація перевірки балансу користувача

Перед покупкою виконується перевірка достатності коштів, що дозволяє уникнути негативного балансу.

Окремо реалізовано захист від повторної покупки привілею (див. рис. 3.27).

```
const [existing] = await db.query(
  "SELECT * FROM purchases WHERE user_id = ? AND donate_id = ?",
  [req.user.id, donateId]
);
if (existing.length > 0) {
  return res.status(400).json({ message: "Вы вже придбали цей донат" });
}
```

Рисунок 3.27 – Реалізація захисту від повторного придбання

Перед видачею привілею система перевіряє наявність прив’язаного Minecraft-акаунта (див. рис. 3.28).

```
if (!user.mc_username) {
  return res.status(400).json({ message: "Minecraft-акаунт не прив'язано" });
}
```

Рисунок 3.28 – Код перевірки прив’язки акаунту перед списанням коштів

Ключовою частиною роботи магазину є інтеграція з Minecraft-сервером через RCON (див. рис. 3.29).

```
const command = `lp user ${user.mc_username} parent add ${donate.name.toLowerCase()}`;
try {
  await sendCommand(command);
} catch (err) {
  console.error("Ошибка RCON:", err);
  return res.status(500).json({ message: "Не вдалося видати донат на сервері, кошти не списані" });
}
```

Рисунок 3.29 – Реалізація видачі привілею на ігровому сервері

Лише після успішного виконання RCON-команди система списує кошти з балансу користувача.

Окремою складовою магазину є система промокодів (див. рис. 3.30). Після отримання коду система виконує пошук у таблиці promos та перевіряється факт попереднього використання промокоду (див. рис. 3.31). Після успішної перевірки користувач отримує бонусний баланс (див. рис. 3.32).

```
router.post("/activate-promo", authMiddleware, async (req, res) => {
  const userId = req.user.id;
  const { code } = req.body;
```

Рисунок 3.30 – Реалізація маршруту активації промокодів

```
if (!code) return res.status(400).json({ message: "Введіть промокод" });

try {
  const [[promo]] = await db.query(
    "SELECT * FROM promos WHERE code = ?",
    [code]
  );

  if (!promo) {
    return res.status(400).json({ message: "Некоректний промокод" });
  }

  const [[used]] = await db.query(
    "SELECT * FROM promo_uses WHERE promo_id = ? AND user_id = ?",
    [promo.id, userId]
  );

  if (used) {
    return res.status(400).json({ message: "Промокод вже використано" });
  }
```

Рисунок 3.31 – Реалізація перевірки актуальності прокомоду

```
    await db.query("UPDATE users SET balance = balance + ? WHERE id = ?", [
      promo.reward,
      userId,
    ]);

    await db.query(
      "INSERT INTO promo_uses (promo_id, user_id) VALUES (?, ?)",
      [promo.id, userId]
    );

    res.json({ message: "Промокод активовано", reward: promo.reward });
  } catch (err) {
    console.error(err);
    res.status(500).json({ message: "Помилка сервера" });
  }
});
```

Рисунок 3.32 – Реалізація збільшення балансу користувача

Таким чином внутрішній магазин реалізує повноцінну систему монетизації ігрового сервера із підтримкою автоматичної видачі привілеїв, системи промокодів та інтеграції з Minecraft-сервером у реальному часі.

3.5 Інтеграція AI-асистента

Однією з функціональних особливостей вебплатформи є інтеграція AI-асистента, який забезпечує автоматизовану взаємодію з користувачами. Основною задачею модуля є обробка текстових запитів та генерація відповідей за допомогою великої мовної моделі Google Gemini. Реалізація асистента виконана як окремий серверний API-маршрут, що дозволяє централізовано обробляти всі звернення клієнтської частини.

Нижче наведено основний код серверної реалізації ШІ-модуля (див. рис. 3.33).

```
const express = require("express");
const router = express.Router();
const { GoogleGenerativeAI } = require("@google/generative-ai");

const genAI = new GoogleGenerativeAI(process.env.GEMINI_API_KEY);

router.post("/", async (req, res) => {
  try {
    const { question } = req.body;

    const model = genAI.getGenerativeModel({ model: "gemini-2.5-flash" });

    const chat = model.startChat({
      history: [
        {
          role: "user",
          parts: [{ text: "Відповідай лише українською мовою. Ти є спеціалістом з Minecraft серверів та міні-ігор." }],
        },
      ],
    });

    const result = await chat.sendMessage(question);

    res.json({
      reply: result.response.text(),
    });
  } catch (err) {
    console.error(err);
    res.status(500).json({ message: "Помилка виклику Асистента" });
  }
});

module.exports = router;
```

Рисунок 3.33 – Реалізація модуля ШІ асистента

На початковому етапі виконується підключення необхідних модулів та ініціалізація AI-клієнта:

– модуль express використовується для створення HTTP-маршруту, через який клієнтська частина надсилає повідомлення до AI-асистента – бібліотека @google/generative-ai забезпечує взаємодію із сервісом Gemini API;
– API-ключ зберігається у змінних середовища (process.env.GEMINI_API_KEY), що дозволяє уникнути його прямого розміщення у вихідному коді – такий підхід підвищує безпеку системи та спрощує конфігурацію застосунку при розгортанні на сервері.

Основна логіка взаємодії із моделлю реалізована всередині POST-маршруту.

Після надходження POST-запиту сервер отримує текст повідомлення із поля question.

Далі виконується створення екземпляра моделі Gemini. Для реалізації обрано модель gemini-2.5-flash, оскільки вона оптимізована для швидкої генерації відповідей та забезпечує низьку затримку при роботі у режимі реального часу. Це особливо важливо для чат-інтерфейсів, де користувач очікує швидку реакцію системи.

Після ініціалізації моделі створюється AI-чат із попередньо заданим контекстом. Масив history використовується для передачі системного контексту моделі. У даному випадку реалізовано обмеження, яке змушує AI-асистента формувати відповіді виключно українською мовою. Такий механізм дозволяє контролювати поведінку моделі та підтримувати єдиний стиль взаємодії із користувачами.

Після створення чату виконується надсилання запиту користувача до AI-моделі. Метод sendMessage() передає текст запитання до моделі Gemini та очікує завершення генерації відповіді. Отриманий результат містить сформований текст, який далі повертається клієнтській частині у форматі JSON.

Таким чином сервер виступає проміжним шаром між фронтендом та AI-сервісом, виконуючи:

– прийом запитів користувача;

– взаємодію із моделлю Gemini;

– обробку відповіді;

– передачу результату у вебінтерфейс.

Окрему увагу приділено механізму обробки помилок. Використання конструкції try/catch дозволяє уникнути аварійного завершення роботи сервера у випадку:

– недоступності Gemini API;

– перевищення лімітів запитів;

– помилок мережевого з’єднання;

– некоректних відповідей AI-сервісу.

У випадку виникнення помилки сервер повертає HTTP-статус 500 та відповідне повідомлення клієнтській частині.

На стороні клієнта AI-асистент інтегрований у вигляді окремої вкладки панелі користувача. Після введення повідомлення фронтенд надсилає POST-запит до маршруту /api/assistant, після чого отримана відповідь відображається у чат-інтерфейсі (див. рис. 3.34).

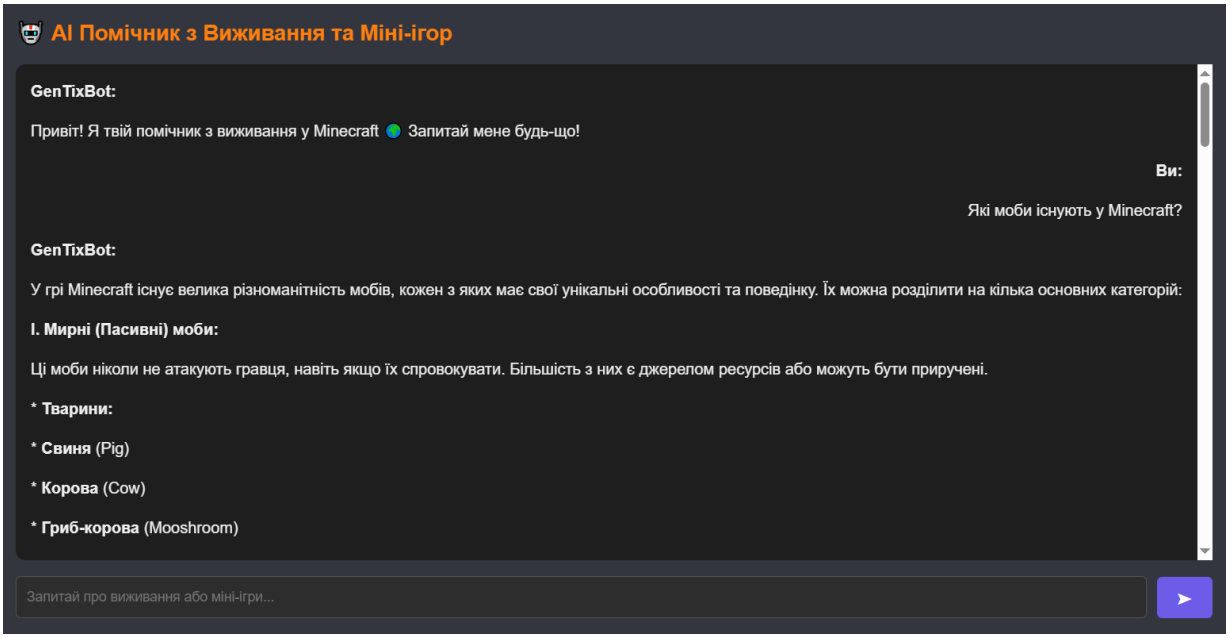


Рисунок 3.34 – Робота з реалізованим асистентом

Інтеграція AI-модуля дозволила реалізувати:

– автоматичну інформаційну підтримку користувачів;

– швидкі відповіді на типові запитання;

– зменшення навантаження на адміністрацію проєкту;

– загальну взаємодію із сервісами штучного інтелекту.

Завдяки серверній архітектурі реалізації модуль також може бути розширений у майбутньому шляхом:

– інтеграції внутрішньої бази знань;

– додавання контекстної пам’яті діалогу;

– фільтрації або модерації AI-відповідей;

– персоналізації відповідей на основі профілю користувача;

– аналізу статистики використання AI-асистента.

Таким чином, AI-асистент виступає окремим інтелектуальним компонентом вебплатформи, який автоматизує інформаційну підтримку користувачів та покращує загальний рівень взаємодії із системою.

3.6 Реалізація панелі адміністратора для керування контентом

Для забезпечення керування контентом вебплатформи реалізовано окрему адміністративну панель. Основним призначенням модуля є надання адміністраторам можливості керувати внутрішнім магазином, новинами та промокодами без необхідності прямого втручання у базу даних.

Архітектурно панель реалізована у вигляді окремого серверного маршруту Express.js із використанням механізму авторизації та перевірки прав доступу.

Усі адміністративні маршрути доступні лише для користувачів із прапорцем is_admin у базі даних.

Ключовим елементом системи безпеки є middleware перевірки адміністратора (див. рис. 3.35).

```
const adminOnly = async (req, res, next) => {
  const [[user]] = await db.query("SELECT is_admin FROM users WHERE id = ?", [req.user.id]);
  if (!user || !user.is_admin) {
    return res.status(403).json({ message: "Немає доступу" });
  }
  next();
};
```

Рисунок 3.35 – Реалізація перевірки на наявність ролі адміністратора

Після проходження авторизації система отримує ідентифікатор користувача із JWT-токена та виконує перевірку поля is_admin у таблиці users. Якщо користувач не має адміністративних прав, сервер повертає HTTP-статус 403 Forbidden.

Такий підхід дозволяє:

– ізолювати адміністративний функціонал;

– запобігти несанкціонованому доступу;

– централізовано контролювати права користувачів;

– використовувати один механізм перевірки для всіх адміністративних маршрутів.

Однією з функцій адміністративної панелі є керування внутрішнім магазином донатів. Адміністратор може:

– переглядати список товарів;

– додавати нові донати;

– редагувати існуючі позиції;

– видаляти товари із системи.

Для отримання списку товарів використовується наступний маршрут (див. рис. 3.36).

```
router.get("/donates", authMiddleware, adminOnly, async (req, res) => {
  const [donates] = await db.query("SELECT * FROM donates");
  res.json(donates);
});
```

Рисунок 3.36 – Реалізація отримання списку товарів

Після проходження авторизації та перевірки прав система виконує SQL-запит до таблиці donates і повертає список усіх товарів у форматі JSON.

Додавання нового товару реалізовано через POST-маршрут (див. рис. 3.37).

```
router.post(
  "/donates",
  authMiddleware,
  adminOnly,
  upload.single("image"),
  async (req, res) => {
    const { name, price, description } = req.body;
    const image = req.file ? `/uploads/${req.file.filename}` : null;

    await db.query(
      "INSERT INTO donates (name, price, description, image) VALUES (?, ?, ?, ?)",
      [name, price, description, image]
    );

    res.json({ message: "Донат додано" });
  }
);
```

Рисунок 3.37 – Реалізація додавання нового товару

У процесі створення товару система:

а) отримує текстові дані товару із req.body;

б) обробляє завантажене зображення через multer;

в) формує шлях до зображення;

г) додає запис у таблицю donates.

Middleware upload.single("image") автоматично обробляє multipart/form-data запити та зберігає файл на сервері.

Редагування товару реалізоване через PUT-запит (див. рис. 3.38). Особливістю реалізації є динамічне формування SQL-запиту. Якщо адміністратор завантажує нове зображення, система оновлює також поле image. В іншому випадку змінюються лише текстові параметри товару.

Видалення товарів реалізовано окремим DELETE-маршрутом (див. рис. 3.39).

```
router.put(
  "/donates/:id",
  authMiddleware,
  adminOnly,
  upload.single("image"),
  async (req, res) => {
    const { name, price, description } = req.body;
    const image = req.file ? `/uploads/${req.file.filename}` : null;

    const sql = image
      ? "UPDATE donates SET name = ?, price = ?, description = ?, image = ? WHERE id = ?"
      : "UPDATE donates SET name = ?, price = ?, description = ? WHERE id = ?";

    const params = image
      ? [name, price, description, image, req.params.id]
      : [name, price, description, req.params.id];

    await db.query(sql, params);
    res.json({ message: "Донат оновлено" });
  }
);
```

Рисунок 3.38 – Реалізація редагування товару

```
router.delete("/donates/:id", authMiddleware, adminOnly, async (req, res) => {
  await db.query("DELETE FROM donates WHERE id = ?", [req.params.id]);
  res.json({ message: "Донат видалено" });
});
```

Рисунок 3.39 – Реалізація видалення товару

Адміністративна панель також містить систему керування новинами проєкту. Вона дозволяє створювати, редагувати та видаляти інформаційні публікації. Для отримання списку новин використовується маршрут (див. рис. 3.40).

```
router.get("/news", authMiddleware, adminOnly, async (req, res) => {
  const [rows] = await db.query("SELECT * FROM news ORDER BY published_at DESC");
  res.json(rows);
});
```

Рисунок 3.40 – Реалізація отримання списку новин

Сортування за published_at DESC дозволяє відображати найновіші публікації першими. Створення новини реалізовано через POST-запит (див. рис. 3.41).

```
router.post(
  "/news",
  authMiddleware,
  adminOnly,
  upload.single("image"),
  async (req, res) => {
    const { title, content, preview, category } = req.body;
    const image = req.file ? `/uploads/${req.file.filename}` : null;

    await db.query(
      "INSERT INTO news (title, content, preview, category, image) VALUES (?, ?, ?, ?, ?)",
      [title, content, preview, category, image]
    );

    res.json({ message: "Новину додано" });
  }
);
```

Рисунок 3.41 – Реалізація додавання новини

У процесі створення новини система підтримує:

– заголовок статті;

– короткий опис;

– повний текст;

– категорію;

– зображення публікації.

Редагування новин реалізовано аналогічно до механізму редагування донатів (див. рис. 3.42).

```
router.put(
  "/news/:id",
  authMiddleware,
  adminOnly,
  upload.single("image"),
  async (req, res) => {
    const { title, content, preview, category } = req.body;
    const image = req.file ? `/uploads/${req.file.filename}` : null;

    const sql = image
      ? "UPDATE news SET title = ?, content = ?, preview = ?, category = ?, image = ? WHERE id = ?"
      : "UPDATE news SET title = ?, content = ?, preview = ?, category = ? WHERE id = ?";

    const params = image
      ? [title, content, preview, category, image, req.params.id]
      : [title, content, preview, category, req.params.id];

    await db.query(sql, params);
    res.json({ message: "Новину оновлено" });
  }
);
```

Рисунок 3.42 – Реалізація редагування новини

Окремою частиною адміністративної панелі є керування промокодами. Для отримання списку активних промокодів використовується маршрут (див. рис. 3.43).

```
router.get("/promos", authMiddleware, adminOnly, async (req, res) => {
  const [promos] = await db.query("SELECT * FROM promos");
  res.json(promos);
});
```

Рисунок 3.43 – Реалізація отримання списку промокодів

Додавання нового промокоду реалізоване через POST-запит (див. рис. 3.44).

```
router.post("/promos", authMiddleware, adminOnly, async (req, res) => {
  const { code, reward } = req.body;

  if (!code || !reward) {
    return res.status(400).json({ message: "Код і нагорода обов'язкові" });
  }

  await db.query("INSERT INTO promos (code, reward) VALUES (?, ?)", [code, reward]);
  res.json({ message: "Промокод додано" });
});
```

Рисунок 3.44 – Реалізація додавання промокоду

У процесі створення промокоду виконується базова перевірка коректності введених даних. Після цього код та значення нагороди зберігаються у таблиці promos.

Видалення промокодів реалізовано окремим DELETE-маршрутом (див. рис. 3.45).

```
router.delete("/promos/:id", authMiddleware, adminOnly, async (req, res) => {
  await db.query("DELETE FROM promos WHERE id = ?", [req.params.id]);
  res.json({ message: "Промокод видалено" });
});
```

Рисунок 3.45 – Реалізація видалення промокоду

На рисунку нижче представлено інтерфейс панелі адміністратора (див. рис. 3.46).

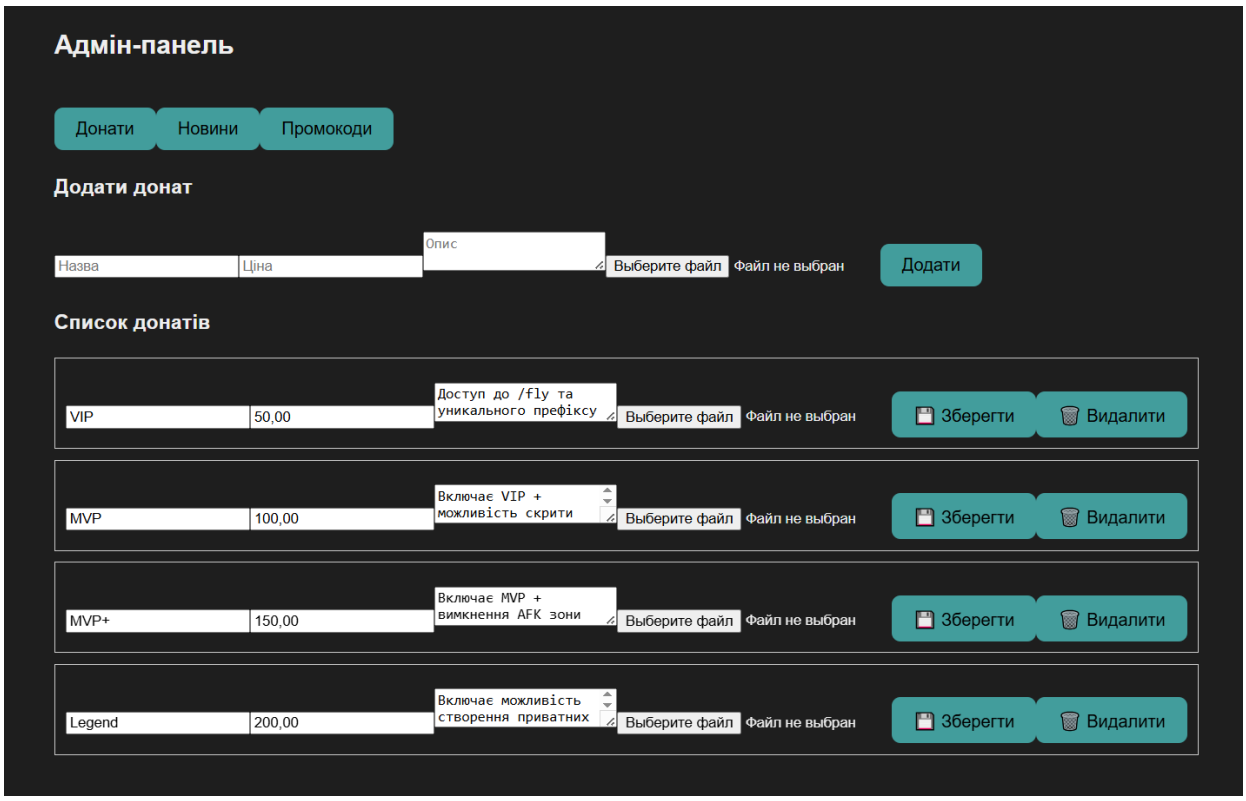


Рисунок 3.46 – Інтерфейс панелі адміністратора

Реалізована панель дозволила:

– централізувати керування контентом;

– автоматизувати редагування інформаційних матеріалів;

– спростити керування магазином та промокодами;

– забезпечити контроль доступу до функціоналу адміністратора;

– реалізувати роботу із мультимедійними файлами.

Таким чином, адміністративний модуль виступає окремою системою керування контентом, яка забезпечує оперативне адміністрування вебплатформи без необхідності прямої роботи із базою даних або серверною інфраструктурою.

3.7 Огляд готового додатка та тестування

Після завершення реалізації основних функціональних модулів необхідно провести тестування вебзастосунку. Основною метою перевірки є

визначення стабільності роботи системи, коректності взаємодії серверної та клієнтської частини, а також перевірка захищеності API від типових атак.

У процесі тестування використовувалися інструменти Apache JMeter [13] для навантажувального тестування, а також Postman [14] для тестування API та перевірки захисту від SQL-ін'єкцій.

Перед проведенням тестування було виконано розгортання серверної частини, бази даних та клієнтського застосунку у тестовому середовищі.

Готовий застосунок являє собою багатофункціональну вебплатформу для Minecraft-сервера, яка поєднує:

– особистий кабінет гравця;
– AI-асистента;
– внутрішній магазин;
– модуль новин;
– панель адміністратора;
– інтеграцію з Minecraft-сервером.

Реалізований додаток можна побачити нижче (див. рис. 3.47).

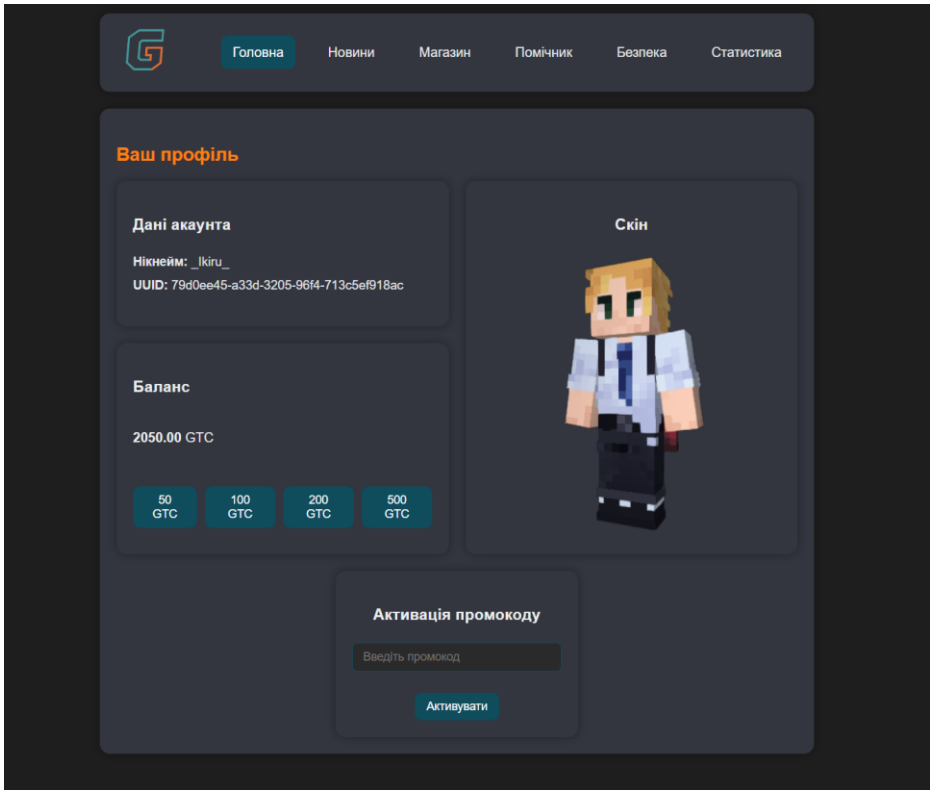


Рисунок 3.47 – Інтерфейс панелі користувача

Основна навігація виконується через меню, яке дозволяє швидко переходити між:

– профілем користувача;
– новинами;
– внутрішнім магазином;
– AI-асистентом;
– налаштуваннями безпеки;
– статистикою гравця.

Для оцінки продуктивності серверної частини було проведено навантажувальне тестування із використанням Apache JMeter. Основною задачею тестування стало:

– визначення стабільності API при одночасних запитах;
– перевірка часу відповіді сервера;
– аналіз обробки великої кількості HTTP-запитів;
– виявлення потенційних вузьких місць системи.

У процесі тестування створено Thread Group у розмірі однієї тисячі користувачів, які одночасно надсилали GET-запити до застосунку. Основне навантаження виконувалося на головну сторінку сайту. У процесі тестування аналізувалися:

– середній час відповіді сервера;
– максимальний час обробки запиту;
– кількість помилок;
– пропускна здатність;
– стабільність роботи сервера під навантаженням.

Результат тестування можна побачити на зображенні нижче (див. рис. 3.48).

Під час тестування сервер коректно обробляв паралельні запити без аварійного завершення роботи.

Для перевірки безпечності серверної частини використовувався інструмент Postman. Основною задачею стало тестування API на стійкість до SQL-ін’єкцій та перевірка коректності обробки некоректних даних.

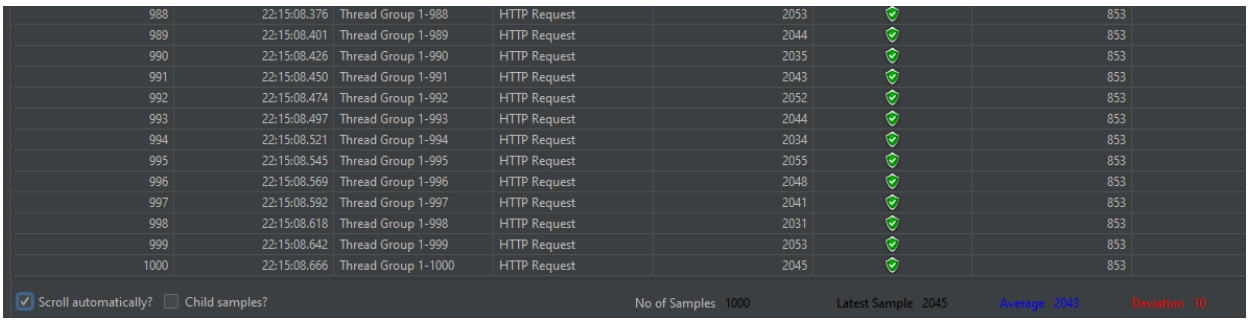


Рисунок 3.48 – Тестування навантаження за допомогою Apache JMeter

Тестування виконувалося шляхом надсилання спеціально сформованих запитів до API-маршрутів авторизації та новин.

Основна увага приділялася маршрутам:

– /api/auth/login;

– /api/auth/register.

У процесі аналізу було встановлено, що серверна частина захищена від SQL-ін’єкцій завдяки використанню параметризованих SQL-запитів.

Розглянемо спробу підміни пароля за допомогою SQL – ін’єкції (див. рис. 3.49).

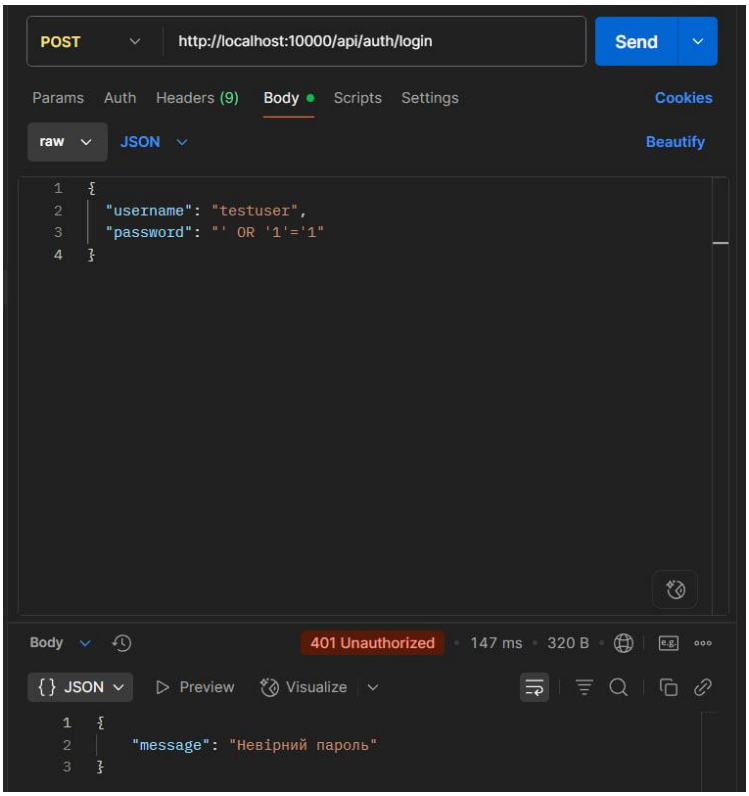


Рисунок 3.49 – Спроба SQL-ін’єкції для обходу пароля

Під час тестування сервер:

– блокував некоректні SQL-конструкції;

– повертав повідомлення про помилку;

– не допускав витоку даних;

– не виконував сторонній SQL-код.

Також розглянемо спробу видалення таблиці користувачів через маршрут реєстрації (див. рис. 3.50).

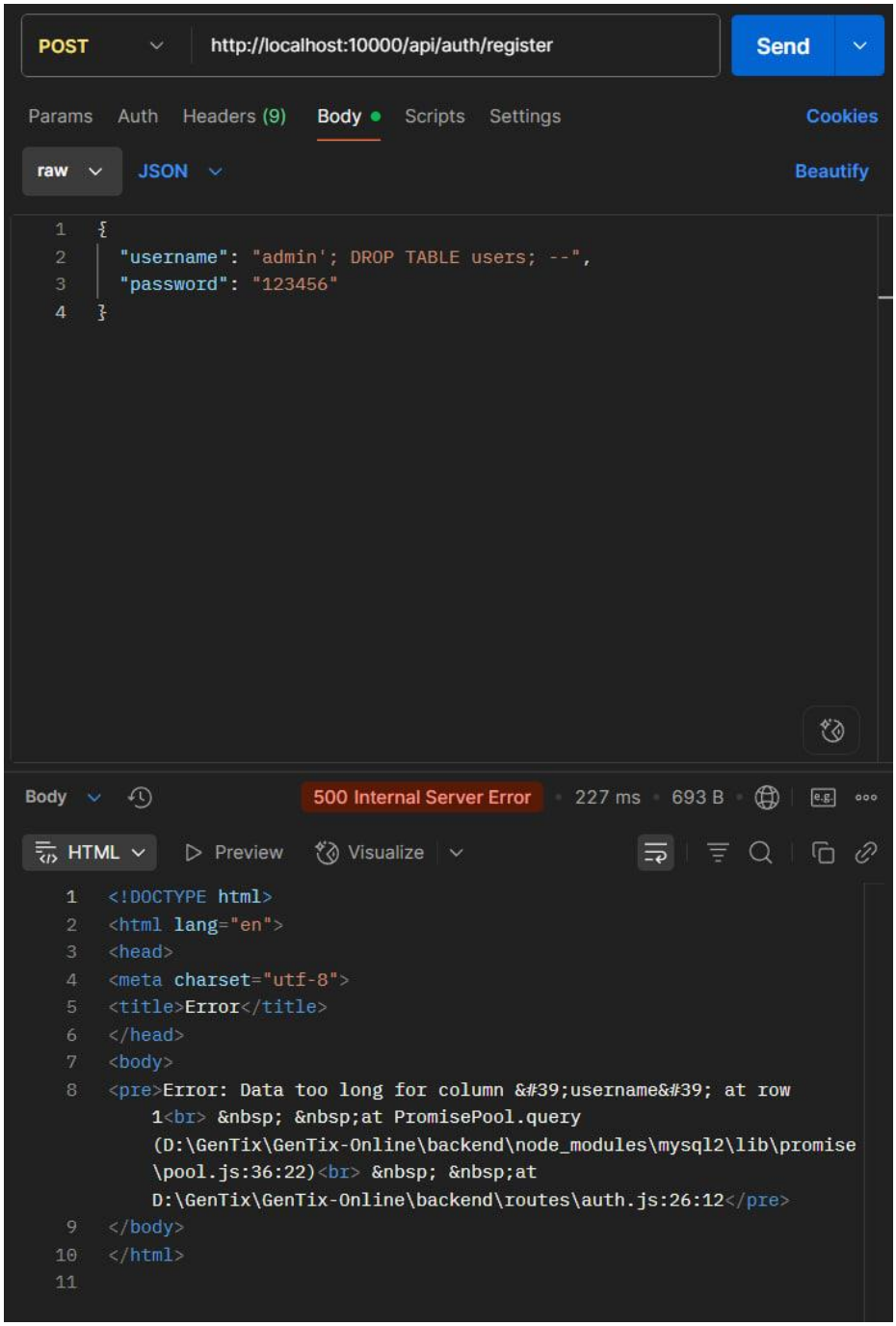


Рисунок 3.50 – Спроба SQL-ін’єкції для видалення таблиці користувачів

У результаті проведеного тестування було встановлено:

– серверна частина стабільно працює під навантаженням;

– API коректно обробляє одночасні запити;

– система використовує безпечні SQL-запити;

– основні модулі функціонують без критичних помилок;

– механізми авторизації та перевірки доступу працюють коректно.

Таким чином, проведене тестування підтвердило працездатність, стабільність та базову захищеність реалізованого вебзастосунку.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи було розроблено вебзастосунок для Minecraft-проєкту, який забезпечує взаємодію між вебплатформою, базою даних, ігровим сервером та AI-модулем. Метою роботи стала розробка ігрової інформаційної платформи з використанням React.js на стороні клієнта та Node.js на стороні сервера для забезпечення зручного доступу до новин, сервісів, адміністрування та AI-асистента.

На серверній частині реалізовано REST API на базі Node.js та Express. Розроблено систему аутентифікації та авторизації з використанням JWT-токенів, bcrypt-хешування паролів, механізмів зміни й відновлення пароля, а також підтвердження електронної пошти. Додатково реалізовано прив’язку Minecraft-акаунта через взаємодію з ігровим сервером.

На клієнтській частині із використанням React створено багатофункціональну панель користувача. Реалізовано систему керування безпекою акаунта, роботу з промокодами, перегляд новин, поповнення балансу, перегляд статистики та інтеграцію AI-асистента.

Також реалізовано внутрішній магазин із можливістю покупки привілеїв, перевіркою балансу користувача та автоматичною видачею через RCON-з’єднання. Для адміністраторів створено окрему панель керування контентом із можливістю додавання новин, донатів та промокодів.

Окремо інтегровано AI-асистента на базі Google Gemini API для генерації відповідей українською мовою та підтримки користувачів.

На завершальному етапі проведено тестування системи за допомогою JMeter та Postman. Результати тестування підтвердили стабільність роботи API та відсутність критичних SQL-вразливостей.

Таким чином, у межах роботи було створено комплексний вебзастосунок, який об’єднує сучасні вебтехнології, інтеграцію з ігровим сервером та інструменти штучного інтелекту в єдину інформаційну систему.

ПЕРЕЛІК ПОСИЛАНЬ

1. Hypixel. URL: https://hypixel.net/ (дата звернення: 12.01.2026).

2. CurseForge. URL: https://www.curseforge.com/ (дата звернення: 12.01.2026).

3. Minecraft Forum. URL: https://www.minecraftforum.net/ (дата звернення: 12.01.2026).

4. React Documentation. URL: https://react.dev/ (дата звернення: 02.02.2026).

5. The starting point for learning TypeScript. URL: https://www.typescriptlang.org/docs/ (дата звернення: 06.02.2026).

6. Node.js Documentation. URL: https://nodejs.org/en/docs/ (дата звернення: 10.02.2026).

7. GitHub REST API documentation. URL: https://docs.github.com/en/rest?apiVersion=2026-03-10 (дата звернення: 15.03.2026).

8. MySQL 8.0 Reference Manual. URL: https://dev.mysql.com/doc/ (дата звернення: 10.03.2026).

9. JSON Web Tokens Introduction. URL: https://jwt.io/introduction (дата звернення: 22.03.2026).

10. Google AI for Developers Documentation. URL: https://ai.google.dev/ (дата звернення: 14.04.2026).

11. Express.js Documentation. URL: https://expressjs.com/ (дата звернення: 04.03.2026).

12. Bukkit API Documentation. URL: https://hub.spigotmc.org/javadocs/bukkit/ (дата звернення: 15.02.2026).

13. Apache JMeter User Manual. URL: https://jmeter.apache.org/usermanual/ (дата звернення: 05.05.2026).

14. Postman Learning Center. URL: https://learning.postman.com/ (дата звернення: 06.05.2026).

15. Mozilla Developer Network Web Docs. URL: https://developer.mozilla.org/en-US/ (дата звернення: 22.01.2026).

16. Axios Documentation. URL: https://axios-http.com/docs/intro (дата звернення: 29.03.2026).

17. GitHub – D3cryptHAX/Bakalavrska_robota_Matviichuk. *GitHub*. URL: https://github.com/D3cryptHAX/Bakalavrska_robota_Matviichuk (дата звернення: 25.05.2026).