МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему: **«РОЗРОБКА ГРИ НА БАЗІ TELEGRAM MINI-APPS»**

Виконав: студент 4 курсу, групи 6.1211-1пі
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми програмна інженерія
(назва освітньої програми)
Д.В. Беркман
(ініціали та прізвище)
Керівник доцент кафедри фундаментальної та прикладної математики, доцент, к.ф.-м.н. Панасенко Є.В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)
Рецензент доцент кафедри комп'ютерних наук, доцент, к.т.н. Матвіїшина Н.В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2025

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)

“ ____ ” _____ 2024 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Беркману Дмитру Валерійовичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка гри на базі Telegram Mini-Apps

керівник роботи Панасенко Євген Валерійович, к.ф.-м.н., доцент
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 20 » грудня 2024 року № 2001-с

2. Строк подання студентом роботи 02.06.2025 р.

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

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Основні теоретичні відомості.
2. Проєктування гри.
3. Програмна реалізація.

5. 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 26.12.2024 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.	20.01.2025	
2.	Збір вихідних даних.	20.02.2025	
3.	Обробка методичних та теоретичних	11.03.2025	
	джерел.		
4.	Розробка першого та другого розділу.	18.04.2025	
5.	Розробка третього розділу.	19.05.2025	
6.	Оформлення та нормоконтроль	26.05.2025	
	кваліфікаційної роботи бакалавра.		
7.	Захист кваліфікаційної роботи.	16.06.2025	

Студент _____ Д.В. Беркман
(підпис) (ініціали та прізвище)

Керівник роботи _____ Є.В. Панасенко
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

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

РЕФЕРАТ

Кваліфікаційна робота бакалавра «Розробка гри на базі Telegram Mini-Apps»: 75 с., 35 рис., 1 табл., 9 джерел, 1 додаток.

ВЕБЗАСТОСУНОК, ВЕБТЕХНОЛОГІЇ, ГРА, ІНТЕРФЕЙС КОРИСТУВАЧА, КЛІКЕР, ХМАРНІ СЕРВІСИ, FIREBASE, REACT, TELEGRAM MINI-APP, TELEGRAM WEBAPP API, UI.

Об’єкт дослідження – інтерактивні вебзастосунки у форматі Telegram Mini-Apps, зокрема ігри жанру «клікер».

Мета роботи – створення гри жанру «клікер» у форматі Telegram Mini-app з використанням сучасних вебтехнологій.

Методи дослідження – структурно-функціональний аналіз, методи проєктування інтерфейсів і ігрових механік, методи конструювання вебзастосунку з використанням React і Firebase, методи інтеграції з Telegram WebApp API, методи тестування функціоналу і користувацького інтерфейсу.

У роботі описано розробку інтерактивної гри жанру «клікер» у форматі Telegram Mini-App з використанням сучасних вебтехнологій. Основу фронтенд-частини реалізовано з використанням бібліотеки React, що забезпечує компонентний підхід до побудови інтерфейсу. Для зберігання та обробки даних застосовано хмарну платформу Firebase, яка забезпечує авторизацію користувачів, зберігання прогресу та синхронізацію даних у реальному часі. Розглянуто інтеграцію з Telegram WebApp API та особливості адаптації гри під середовище месенджера. Результатом є гра, яка демонструє ефективне використання вебтехнологій у створенні динамічного та зручного для користувача застосунку.

SUMMARY

Bachelor’s qualifying paper “Development of Game Based on the Telegram Mini-Apps”: 75 pages, 35 figures, 1 table, 9 references, 1 supplement.

CLICKER, CLOUD SERVICES, FIREBASE, GAME, REACT, TELEGRAM MINI-APP, TELEGRAM WEBAPP API, UI, USER INTERFACE, WEB APPLICATION, WEB TECHNOLOGIES

Object of research – interactive web applications in Telegram Mini-Apps format, in particular games of the “clicker” genre.

The purpose of the work is to create a game of the “clicker” genre in the Telegram Mini-app format using modern web technologies.

Research methods – structural and functional analysis, methods of designing interfaces and game mechanics, methods of constructing a web application using React and Firebase, methods of integration with the Telegram WebApp API, methods of testing functionality and user interface.

The work describes the development of an interactive game of the “clicker” genre in the Telegram Mini-App format using modern web technologies. The basis of the front-end part is implemented using the React library, which provides a component approach to building the interface. The Firebase cloud platform is used for data storage and processing, which provides user authorization, progress storage and data synchronization in real time. Integration with the Telegram WebApp API and features of adapting the game to the messenger environment are considered. The result is a game that demonstrates the effective use of web technologies in creating a dynamic and user-friendly application.

ЗМІСТ

Завдання на кваліфікаційну роботу……..2
Реферат 4
Summary 5
Вступ..... 8
1 Теоретичні основи розробки Telegram Mini-Apps та ігрових механік 10
1.1 Загальна інформація про Telegram Mini-Apps 10
1.1.1 Особливості Telegram Mini-Apps 10
1.1.2 Переваги використання Telegram Mini-Apps 11
1.2 Інструменти, що були використані для розробки проєкту 11
1.2.1 React..... 11
1.2.2 Firebase 12
1.2.3 Telegram WebApp API 13
1.2.4 GitHub Pages 13
1.3 Ігрові механіки гри..... 13
1.4 Висновки до розділу 1 14
2 Проєктування гри на базі telegram mini-apps 16
2.1 Архітектура застосунка 16
2.1.1 Структура застосунку 16
2.1.2 Маршрутизація застосунку 17
2.1.3 Інтеграція з Telegram WebApp API 18
2.1.4 Використання Firebase..... 18
2.2 Таблиця атрибутів компонентів 19
2.3 ER-діаграма сутностей 22
2.4 Висновки до розділу 1 24
3 Реалізація гри на базі Telegram Mini-Apps 25
3.1 Ініціалізація Firebase та Telegram WebApp API 25
3.2 Реалізація основний функцій гри 26

3.2.1 Компонент App 26
3.2.2 Головна сторінка гри (MainPage) 28
3.2.3 Головні показники гри (MainStats) 30
3.2.4 Тимчасові бустери (BoostsService) 33
3.2.5 Сторінка покращень (UpgradePage) 35
3.2.6 Компонент покращень (UpgradeService) 35
3.2.7 Сторінка магазину (ShopPage) 37
3.2.8 Компонент магазину скінів (SkinService) 37
3.2.9 Топ гравців (LeaderboardPage) 38
3.2.10 Компонент топу гравців (LeaderboardService) 39
3.2.11 Сторінка досягнень (AchievementsPage) 40
3.2.12 Компонент сторінки досягнень (AchievementsService) 40
3.2.13 Сторінка налаштувань (SettingsPage) 41
3.3 Взаємодія користувача з грою 42
3.4 Висновки до розділу 3 49
Висновки 51
Перелік посилань 53
Додаток А Код основних компонентів гри-клікер 54
A.1 MainStats.js 54
A.2 BoostsService.js 59
A.3 UpgradeService.js 64
A.4 SkinService.js 66
A.5 LeaderboardService.js 70
A.6 AchievementsService.js 72

ВСТУП

Сучасне програмне забезпечення все частіше орієнтується не лише на функціональність, а й на інтерактивність та залучення користувача. Одним із напрямів, що активно розвивається, є створення легких, доступних ігор та додатків, інтегрованих у вже звичні для користувача платформа, зокрема месенджери.

Популярність Telegram як платформи з відкритим API та підтримкою Mini-Apps зумовила стрімкий розвиток різноманітних сервісів без потреби інсталювати окремі застосунки. Ігрові проєкти у форматі Mini-Apps мають суттєві переваги: простота запуску, збереження прогресу в хмарі, підтримка мобільних та десктопних платформ без дублювання логіки. Це робить розробку ігор не лише актуальною, а й перспективною як з технічної, так і з комерційної точки зору [1].

Актуальність теми визначається зростанням попиту на ігровий контент у месенджерах, а також можливістю створення повноцінного інтерактивного ігрового досвіду без необхідності виходити за межі звичних комунікаційних інструментів. Зокрема, жанр клікерів ідеально підходить для міні-ігор у месенджерах завдяки своїй простоті, повторюваності та можливості легкого масштабування.

Метою даної кваліфікаційної роботи є створення гри жанру «клікер» у форматі Telegram Mini-app з використанням сучасних вебтехнологій.

Досягнення поставленої мети передбачає реалізацію наступних завдань:

– проаналізувати можливості створення інтерактивних застосунків у форматі Telegram Mini-Apps;
– розробити архітектуру гри типу «клікер» з підтримкою авторизації через Telegram WebApp API;
– спроєктувати функціональні модулі: система кліків, енергії, бустерів, досягнень, потращень та магазин;

– реалізувати збереження ігрових даних у Firebase [2];
– організувати розміщення сайту з використанням GitHub Pages [3].

Отримані результати мають практичне значення як приклад розробки інтерактивної гри у форматі Mini-App для Telegram. Реалізовані підходи можуть бути адаптовані для інших ігрових чи комерційних мінізастосунків. Кваліфікаційна робота складається зі вступу, трьох розділів, висновків, переліку використаних джерел та додатків.

1 ТЕОРЕТИЧНІ ОСНОВИ РОЗРОБКИ TELEGRAM MINI-APPS ТА ІГРОВИХ МЕХАНІК

1.1 Загальна інформація про Telegram Mini-Apps

Telegram Mini-App – це вебзастосунок, що запускається безпосередньо всередині інтерфейсу Telegram, без потреби встановлення окремих додатків. Такий тип застосунків відкривається у спеціальному вебконтейнері та взаємодіють через вбудований Telegram WebApp API.

Такий формат додатку відповідає сучасним тенденціям – користувачі цінують швидкий доступ, простоту використання і інтеграцію з профілем Telegram, що полегшує авторизацію і персоналізацію.

Також Telegram Mini-Apps підтримують роботу на різних платформах, включно з мобільними пристроями та десктопом, що значно розширює охоплення аудиторії.

1.1.1 Особливості Telegram Mini-Apps

Telegram Mini-App має деякі особливості, які роблять його зручним для інтеграції у Telegram.

Основною перевагою є автоматична авторизація, що при запуску застосунку передає базову інформацію про користувача, зокрема його унікальний ідентифікатор. Це дає можливість не створювати додаткову форму входу чи реєстрації. Також застосунок має можливість тісно інтегруватися з Telegram-ботом, а саме відкриватися з кнопки в меню бота, або після певного повідомлення у чаті.

Для безпечної передачі даних використовується формат у зашифрованому вигляді, забезпечуючи автентичність та захист інформації.

1.1.2 Переваги використання Telegram Mini-Apps

Розробка застосунків формату Telegram Mini-Apps має низку важливих переваг, а саме:

- **легке масштабування та оновлення** – Telegram Mini-Apps є вебзастосунками, тому усі оновлення відбуваються на строні сервера;
- **вбудованість у екосистему Telegram** – додаток вбудовується у месенджер Telegram, тому не потрібно нічого завантажувати окремо;
- **широке охоплення аудиторії** – Telegram має сотні мільйонів користувачів, що дає змогу працювати з великою базою потенційних клієнтів без додаткових маркетингових зусиль.

1.2 Інструменти, що були використані для розробки проєкту

Розробка сучасних вебзастосунків в наш час доволі часто базується на використанні готових технологічних рішень, які забезпечують швидкий старт, зручну організацію коду та гнучке масштабування функціоналу.

Для розробки гри було використано сучасний стек технологій, що поєднує у собі продуктивність React, хмарні можливості Firebase, зручність Telegram WebApp API та можливості розгортання через GitHub Pages.

Завдяки цим технологіям вдалося створити зручний та масштабований застосунок з високим рівнем інтерактивності, надійним збереженням даних та безшовною інтеграцією у екосистему месенджера.

1.2.1 React

React – це популярна бібліотека, створена компанією Meta, яка використовується для побудови зручних інтерфейсів для користувача. Бібліотека орієнтована на односторінкові застосунки (SPA), де змінюється

лише частина сторінки без повного її перезавантаження, що забезпечує швидкодію додатку та зручність у користувача [4].

React має деякі особливості, які зробили його однією з найпопулярніших бібліотек для розробки інтерфейсів. Головною перевагою є те, що він базується на компонентному підході, що дозволяє легко розбивати інтерфейс на незалежні багаторазові компоненти. Також, React використовує віртуальний DOM – копію дерева елементів сторінки, яка дозволяє оновлювати лише частини інтерфейсу, що змінилися [5].

Важливою особливістю є підтримка хуків, які дозволяють функціональним компонентам керувати станом, обробляти побічні ефекти на інші аспекти життєвого циклу. Окрім цього, React має доволі велику розвинену екосистему, що включає багато різних допоміжних бібліотек.

React Router – це бібліотека для маршрутизації у застосунках, побудованих на React. Вона дозволяє створювати багатосторінкові інтерфейси в середині односторінкового застосунку. Вона робить можливим створення зручної та динамічної навігації, не втрачаючи переваги односторінкової архітектури [6].

1.2.2 Firebase

Firebase – це платформа для розробки веб та мобільних застосунків, створена компанією Google, яка надає набір інструментів і сервісів для зручного та швидкого створення, розгортання та масштабування застосунків без необхідності створення власного серверного середовища [7].

Однією з головних переваг Firebase є хмарна база даних Firestore, яка дозволяє зберігати структуровані дані у вигляді документів та колекцій з підтримкою синхронізації в реальному часі між користувачами. Також Firebase автоматично масштабується під навантаження, тому застосунок може працювати стабільно незалежно від кількості користувачів.

1.2.3 Telegram WebApp API

Telegram WabApp API – це інструмент для розробки інтерактивних вебзастосунків, які запускаються прямо всередині Telegram у чатах або через ботів, зокрема для створення міні-ігор, онлайн-магазинів та інших сервісів.

Однією з основних переваг є можливість у реальному часі отримувати дані про користувача та взаємодіяти з ним у межах інтерфейсу, що забезпечує швидку та зручну інтеграцію вебзастосунків у середовище месенджера.

1.2.4 GitHub Pages

GitHub Pages – це сервіс для безкоштовного розгортання статичних вебзастосунків безпосередньо з репозиторіїв на платформі GitHub. Він дозволяє швидко публікувати сайти, документацію чи проєкти, не потребуючи окремого серверного хостингу.

Основною перевагою є простота налаштування та автоматичне оновлення сайту при кожному пуші змін у репозиторій, що забезпечує швидкий і зручний процес розгортання вебзастосунку.

1.3 Ігрові механіки гри

У рамках роботи було реалізовано захоплюючу гру-клікер з різними ігровими механіками, а саме:

- **основний механізм кліків** – гравець заробляє ігрові монети шляхом натискання на кнопку;
- **система енергії** – є певна кількість енергії, яка відіймається разом з додаванням монет;
- **покращення прогресу гри**:

  - **множник кліків** – за кожен рівень збільшується кількість монет за взаємодію з кнопкою;
  - **максимальна кількість енергії** – за кожен рівень збільшується максимальний вміст накопиченої з часом енергії;
  - **регенерація енергії** – за кожен рівень збільшується регенерація енергії за 1 секунду;
- **магазин скінів** – гравець може придбати модернізацію зовнішнього виду кнопки за монетки;
- **топ гравців** – відображення глобального топу гравців по заробітку монет;
- **тимчасові бусти**:
  - **тимчасовий множник** – множник, який тимчасово помножує заробіток за натискання на кнопку;
  - **відновлення енергії** – моментальна регенерація усієї енергії;
- **нікнейми** – кожен гравець має свій нікнейм.

1.4 Висновки до розділу 1

Отже, було розглянуто теоретичні основи розробки Telegram Mini-Apps та основні ігрові механіки клікерів. Описано особливості Telegram Mini-Apps як вебзастосунків, що запускаються безпосередньо в інтерфейсі месенджера, що забезпечує швидкий доступ, простоту використання та тісну інтеграцію з екосистемою Telegram. Виділено ключові переваги таких застосунків, зокрема автоматичну авторизацію, кросплатформеність та широке охоплення аудиторії.

Проаналізовано інструменти, які було використано для розробки проєкту, а саме React, Firebase, Telegram WebApp API та GitHub Pages. Пояснено вибір цих технологій з огляду на їх продуктивність, гнучкість, масштабованість та можливості інтеграції.

Окремо детально розглянуто ігрові механіки клікеру, що реалізовані у проєкті, включно з системою кліків, системою енергії, покращеннями прогресу, магазином скінів, топом гравців та тимчасовими бустами. Це створює основу для динамічної та цікавої ігрової взаємодії користувача з застосунком.

Таким чином, розділ закладає теоретичну та технологічну базу для подальшої розробки та реалізації інтерактивної гри у форматі Telegram Mini-App.

2 ПРОЄКТУВАННЯ ГРИ НА БАЗІ TELEGRAM MINI-APPS

У цьому розділі розглянуто архітектурні особливості програмного застосунку, реалізованого у форматі Telegram Mini-Apps. У підрозділах подано опис структури компонентів, маршрутизації між сторінками, реалізації роботи з Firebase та Telegram, а також таблицю з атрибутами компонентів та ER-діаграму, яка ілюструє основні сутності та зв’язки між ними.

2.1 Архітектура застосунка

Архітектура застосунка побудована за безсерверною моделлю з використанням Telegram Mini-Apps, React та хмарного сервісу Firebase. Уся логіка роботи гри реалізована на стороні клієнта, а зберігання та обробка даних здійснюється через Firebase без потреби у власному сервері. Інтеграція з Telegram WebApp API забезпечує безшовну взаємодію з користувачем безпосередньо у середовищі Telegram, що робить доступ до гри швидким і зручним. Такий підхід дозволяє забезпечити масштабованість та мінімальні затрати на інфраструктуру.

2.1.1 Структура застосунку

Гра побудована з компонентів, які здійснюють функції:

- **App** – основний компонент, що здійснює ініціалізацію Telegram WebApp API та Firestore і виконує маршрутизацією між сторінками;
- **MainPage** – компонент сторінки, що містить основні компоненти гри;
- **MainStats** – компонент з основною логікою гри;
- **BoostsService** – компонент з логікою тимчасових бустерів;
- **UpgradePage** – компонент сторінки для придбання покращень;

– **UpgradeService** – компонент з логікою покращень;
– **ShopPage** – компонент сторінки для придбання скінів;
– **SkinsService** – компонент з логікою вибору скінів;
– **LeaderboardPage** – компонент сторінки з відображенням топу гравців;
– **LeaderboardService** – компонент з логікою топу гравців;
– **AchievementsPage** – компонент сторінки з досягненнями гравця;
– **AchivementsService** – компонент з логікою досягнень;
– **SettingPage** – сторінка з налаштуваннями.

Структуру компонентів зображено на рисунку 2.1.

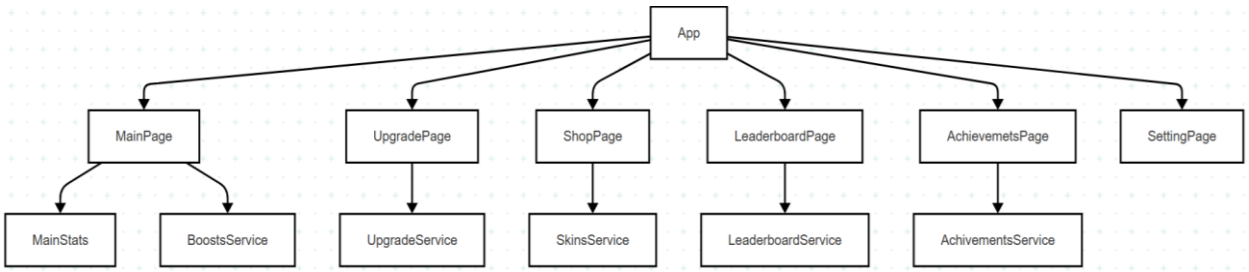


Рисунок 2.1 – Схема структури компонентів

2.1.2 Маршрутизація застосунку

Маршрутизація застосунку здійснена за допомогою бібліотеки React Router, яка дозволяє здійснювати навігацію між сторінками без перезавантаження сторінки. Основні маршрути:

– **/clicker/** – головна сторінка гри;
– **/clicker/upgrade** – сторінка з покращеннями;
– **/clicker/leaderboard** – сторінка з топом гравців;
– **/clicker/shop** – сторінка з магазином скінів;
– **/clicker/achievements** – сторінка з досягненнями;
– **/clicker/settings** – сторінка з налаштуваннями.

Схему маршрутизації зображено на рисунку 2.2.

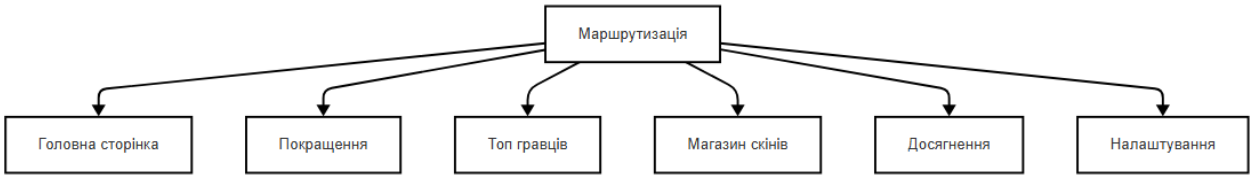


Рисунок 2.2 – Схема маршрутизації гри

2.1.3 Інтеграція з Telegram WebApp API

Для інтеграції у Telegram Mini-App було використано Telegram WebApp API, який дозволяє взаємодіяти з Telegram-клієнтом безпосередньо у вікні браузера користувача.

З метою спрощення доступу до функцій Telegram WebApp API було реалізовано кастомний React-хук useTelegram, який виконує ініціалізацію, перевіряє готовність до роботи та витягує унікальний userId користувача для подальшої персоналізації гри.

Завдяки цій інтеграції користувач може миттєво запускати гру без необхідності авторизації або встановлення сторонніх додатків, що є ключовою перевагою Telegram Mini-Apps.

2.1.4 Використання Firebase

Для збереження даних було використано хмарну платформу Firebase, а саме модуль Cloud Firestore.

Для взаємодії з Firestore було реалізовано кастомний хук useFirebase, який відповідає за ініціалізацію з’єднання з базою даних і забезпечує доступ до базових операцій (читання, запис, оновлення). Цей хук дозволяє швидко підключити Firebase до будь-якого компонента, зберігаючи при цьому гнучкість та повторне використання коду.

Схема запитів до бази даних зображена на рисунку 2.3.

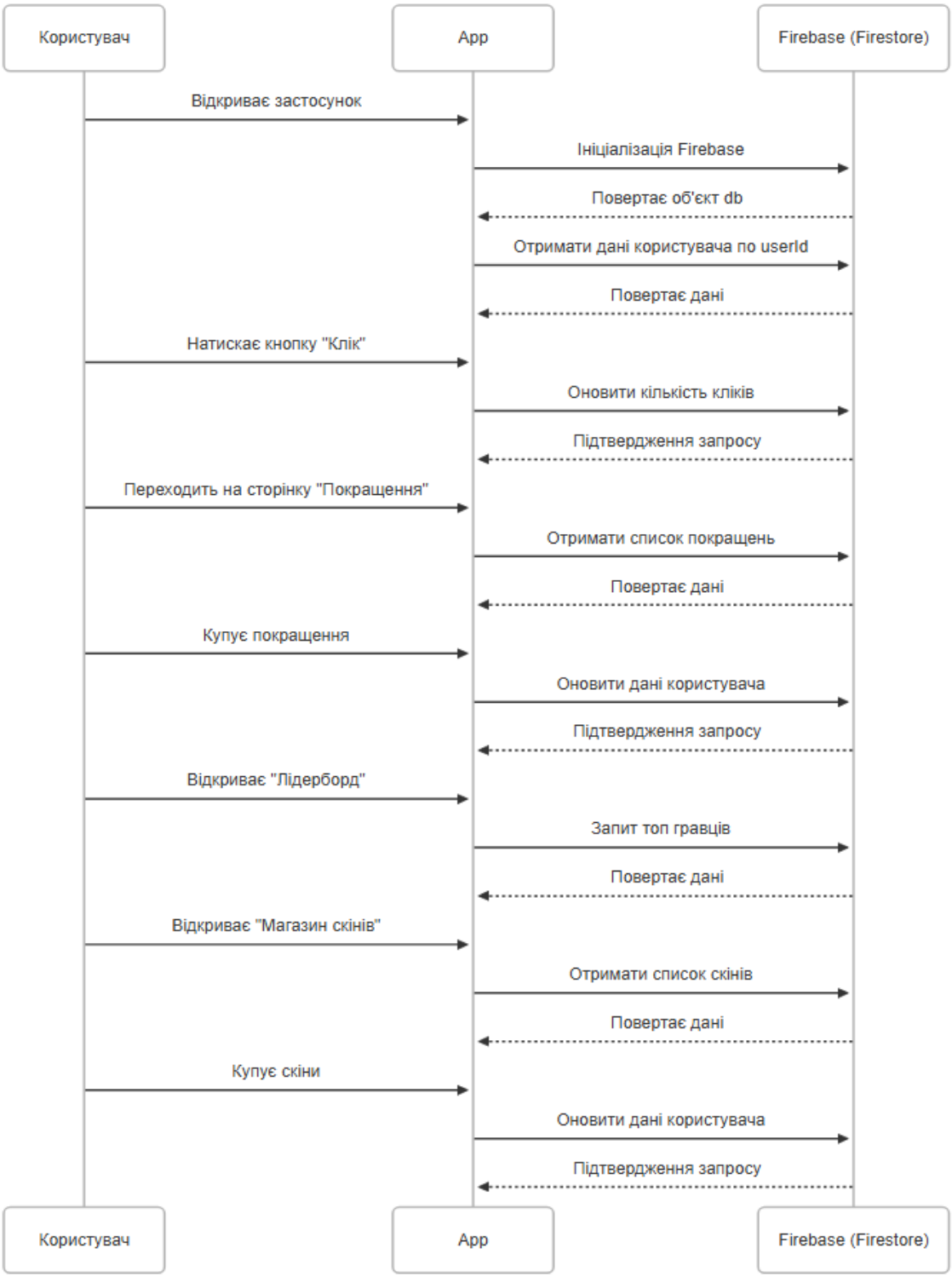


Рисунок 2.3 – Схема запитів до бази даних

2.2 Таблиця атрибутів компонентів

Компоненти реалізовані у вигляді окремих функціональних модулів у React, що дозволяє підтримувати високу модульність та масштабованість

застосунка. Для кращого розуміння структури гри та її логіки було створено таблицю, що описує основні компоненти, їх атрибути, типи даних та опис їх призначення (див. табл. 2.1).

Таблиця 2.1 – Опис атрибутів компонентів та їх типи

Компонент	**Атрибут**	**Тип**	**Опис**
App	userId	Число	Ініціалізація Telegram WebApp API
App	db	Число	Ініціалізація Firestore
MainPage	handleClick	Функція	Функція для обробки кліків по кнопці
MainPage	increment	Функція	Функція для збільшення монет з урахування множника та бустерів
MainPage	showFloatingNumber	Функція	Функція для створення анімації плаваючих чисел
MainStats	loading	Логічний	Ідентифікатор завантаження даних
MainStats	setLoading	Функція	Функція для оновлення стану loading
MainStats	count	Число	Поточний лічильник кліків
MainStats	setCount	Функція	Функція для оновлення лічильнику
MainStats	totalCount	Число	Загальний лічильник кліків
MainStats	setTotalCount	Функція	Функція для оновлення глобального лічильника
MainStats	multiplier	Число	Множник кліків
MainStats	setMultiplier	Функція	Функція для оновлення множника
MainStats	username	Рядок	Нікнейм користувача
MainStats	setUsername	Функція	Функція для оновлення нікнейму
MainStats	showUsernameForm	Логічний	Ідентифікатор відображення форми нікнейму
MainStats	setShowUsernameForn	Функція	Функція для оновлення показу форми нікнейму
MainStats	energy	Число	Поточний рівень енергії
MainStats	setEnergy	Функція	Функція для оновлення енергії
MainStats	maxEnergyLevel	Число	Поточний рівень максимальної енергії
MainStats	setMaxEnergyLevel	Функція	Функція оновлення максимальної кількості енергії
MainStats	regenSpeedLevel	Число	Поточний рівень швидкості відновлення енернії
MainStats	setRegenSpeedLevel	Функція	Функція для оновлення швидкості відновлення енергії
MainStats	pendingSave	Логічний	Ідентифікатор для відкладеного збереження даних
MainStats	getStatsRef	Функція	Функція для отримання посилання на документ статистики користувача Firestore

Продовження таблиці 2.1

Компонент	Атрибут	Тип	Опис
MainStats	setPendingSave	Функція	Функція для відкладеного збереження даних
MainStats	localkey	Рядок	Ключ для збереження локальних даних
MainStats	isReady	Логічний	Прапорець для готовності
MainStats	lastEnergySaveTime	Посилання	Посилання на час останнього збереження енергії
MainStats	getUpgradeCost	Функція	Функція, що повертає вартість покращення з Firestore
MainStats	saveUsername	Функція	Функція для збереження імені користувача в Firestore
BoostsService	energyRegenCooldown	Число	Час відновлення енергії
BoostsService	setEnergyRegenCooldown	Функція	Функція оновлення стану відновлення енергії
BoostsService	boostActive	Логічний	Ідентифікатор стану активності буста
BoostsService	setBoostActive	Функція	Функція оновлення ідентифікатора активності буста
BoostsService	boostRemainingTime	Число	Залишковий час дії буста
BoostsService	setBoostRemainingTime	Функція	Функція часу буста
BoostsService	boostCooldown	Число	Залишковий час кулдауну буста
BoostsService	lastRegenTime	Число	Час останнього відновлення енергії
BoostsService	activateBoost	Функція	Функція активації буста та синхронізації з Firestore
BoostsService	fetchRegenCooldown	Функція	Функція синхронізації часу буста відновлення енергії з Firestore
BoostsService	handleRegenEnergy	Функція	Функція синхронізації буста відновлення кулдауну енергії з Firestore
UpgradePage	none	–	Компонент, що відповідає за відображення інформації на сторінці покращень
UpgradeService	showErrorModal	Логічний	Ідентифікатор показу помилки при недостатньої кількості монет
UpgradeService	handleBuy	Функція	Функція для придбання покращень
UpgradeService	localCount	Число	Локальна кількість момент, що була синхронізована з Firestore

Продовження таблиці 2.1

Компонент	Атрибут	Тип	Опис
UpgradeService	localMultiplier	Число	Локальний множник, синхронізований з Firestore
UpgradeService	localMaxEnergy	Число	Локальний рівень максимальної енергії, синхронізований з Firestore
ShopPage	none	–	Компонент, що відповідає за відображення інформації на сторінці магазину скінів
SkinsService	showErrorModal	Логічний	Ідентифікатор показу помилки при недостатньої кількості монет
SkinsService	handleBuySkin	Функція	Функція для придбання скінів
SkinsService	selectedSkin	Рядок	Поточний вибраний скін, синхронізований з Firestore
leaderboardPage	none	–	Компонент, що відповідає з відображення інформації на сторінці топу гравців
leaderboardService	loading	Логічний	Ідентифікатор завантаження топу гравців
leaderboardService	desplayList	Об’єкт	Об’єкт, який містить інформацію про топ гравців, який синхронізований з Firestore
AchievementsPage	none	–	Компонент, що відповідає за відображення інформації на сторінці з досягненнями
AchievementsService	achievements	Масив	Список досягнень користувача, синхронізований з Firestore
SettingPage	none	–	Компонент, що відповідає за відображення інформації на сторінці за налаштуваннями

2.3 ER-діаграма сутностей

ER-діаграма ілюструє структуру компонентів гри. Вона відображає основні сутності системи та їх взаємозв’язки, архітектуру додатку, показуючи, як різні компоненти взаємодіють для забезпечення правильного функціонування гри. Також діаграма допомагає краще зрозуміти ролі кожного компонента та їх взаємодію.

На діаграмі відображаються ключові сторінки інтерфейсу та сервіси, що відповідають за логіку гри. Це дає змогу побачити загальну архітектуру системи та її модульність, що полегшує розробку, тестування, та подальше розширення функціоналу (див. рис. 2.4).

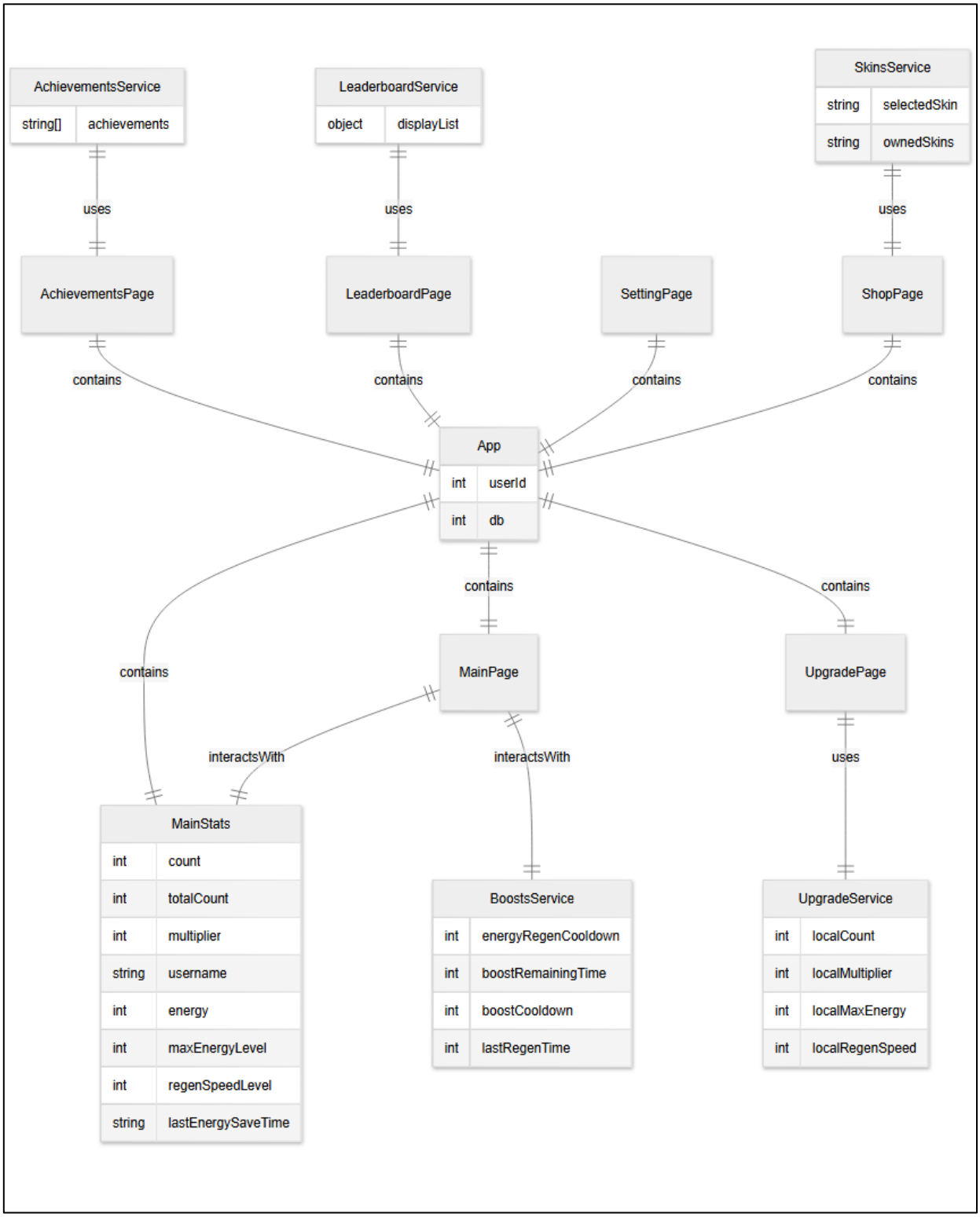


Рисунок 2.4 – ER-діаграма сутностей

2.4 Висновки до розділу 1

Отже, було детально розглянуто архітектурні особливості програмного застосунку, реалізованого у форматі Telegram Mini-Apps із використанням React та хмарного сервісу Firebase. Безсерверна модель архітектури забезпечує високу масштабованість і зручність розробки, оскільки вся логіка гри виконується на клієнтській стороні, а зберігання і обробка даних – у Firebase.

Інтеграція з Telegram WebApp API через кастомний React-хук useTelegram дозволяє забезпечити безшовний доступ до гри без додаткової авторизації, що значно покращує користувацький досвід. Використання React Router для маршрутизації створює плавну та зручну навігацію між різними сторінками застосунку без перезавантаження. Реалізація кастомного хука useFirebase дозволяє централізовано керувати операціями з базою даних, забезпечуючи повторне використання коду та гнучкість.

Таблиця атрибутів компонентів та ER-діаграма сутностей надають чітке уявлення про структуру застосунку, функціональні модулі і взаємозв’язки між ними, що сприяє підтримці, масштабуванню та подальшому розвитку проєкту.

Загалом, архітектура застосунку є ефективною та адаптованою для роботи в середовищі Telegram Mini-Apps, що забезпечує якісний користувацький досвід і простоту розгортання.

3 РЕАЛІЗАЦІЯ ГРИ НА БАЗІ TELEGRAM MINI-APPS

У цьому розділі описано процес створення інтерактивної клікер-гри, реалізованої як Telegram Mini-App за допомогою технології React. Основну увагу приділено технічним аспектам проєкту, таким як ініціалізація Firebase для зберігання даних, використання Telegram WebApp API для взаємодії з користувачем у середовищі месенджера, реалізація основної ігрової логіки та забезпечення динамічного користувацького досвіду.

3.1 Ініціалізація Firebase та Telegram WebApp API

Для коректної роботи гри в Telegram та синхронізації даних із хмарною базою даних було реалізовано два окремих хуки: useTelegramInit та useFirebase. Вони дозволяють отримати дані користувача з Telegram та забезпечити зберігання цих даних у Firebase.

Хук useTelegramInit забезпечує взаємодію з Telegram WebApp API (див. рис. 3.1). Він активує методи ready() та expand(), які повідомляють Telegram-клієнту про готовність вебдодатку до взаємодії та розгортають інтерфейс на повний екран.

```
export function useTelegramInit() {
  const [userId, setUserId] = useState(null);

  useEffect(() => {
    const tg = window.Telegram?.WebApp;
    if (!tg) return;

    tg.ready();
    tg.expand();

    const unsafe = tg.initDataUnsafe;
    if (unsafe?.user?.id) {
      setUserId(unsafe.user.id);
    }
  }, []);

  return userId;
}
```

Рисунок 3.1 – Хук useTelegramInit

Основним етапом є отримання об’єкта initDataUnsafe, з якого витягується унікальний ідентифікатор користувача user.id. Він використовується як ключ збереження персональних даних користувача в базі Firebase.

Хук useFirebase відповідає за підключення до хмарної NoSQL бази даних Firestore (див. рис. 3.2). Ініціалізація відбувається лише один раз, після цього створюється екземпляр бази даних db, доступний для зчитування та запису даних. Після успішного підключення змінна стану initialized встановлюється в true.

```
export function useFirebase() {
  const [initialized, setInitialized] = useState(false);

  useEffect(() => {
    if (!firebase.apps.length) {
      firebase.initializeApp(firebaseConfig);
      db = firebase.firestore();
    }
    setInitialized(true);
  }, []);

  return { db, initialized };
}
```

Рисунок 3.2 – Хук useFirebase

3.2 Реалізація основний функцій гри

3.2.1 Компонент App

Компонент App є головним кореневим компонентом, який відповідає за ініціалізацію додатку, маршрутизацію сторінок та умовний рендер основного інтерфейсу, після завершення ініціалізації (див. рис. 3.3).

Основні функції компоненту:

– **useTelegramInit**: ініціалізує Telegram WebApp API та повертає userId (використовується для отримання ідентифікатора користувача Telegram, щоб пов’язати його з даними у Firebase);

– **useFirebase**: ініціалізує Firebase (повертає об’єкт із двома властивостями, а саме initialized (показує, чи завершилась ініціалізація) та db (посилання на базу даних Firebase));
– **isReady**: перевіряє, чи всі необхідні дані (userId, initialized, db) готові для відображення додатку (якщо хоча б одне з них відсутнє – показується екран завантаження).

```
function App() {

  const userId = useTelegramInit();
  const { initialized, db } = useFirebase();

  const isReady = userId && initialized && db;

  if (!isReady) {
    return <div className="wrapper"><p>Завантаження…</p></div>;
  }

  return (
    <Router basename="/clicker">
      <Routes>
        <Route path="/" element={<MainPage userId={userId} db={db} initialized={initialized} />} />
        <Route path="/upgrade" element={<UpgradePage userId={userId} db={db} initialized={initialized} />} />
        <Route path="/leaderboard" element={<LeaderboardPage userId={userId} db={db} initialized={initialized} />} />
        <Route path="/shop" element={<ShopPage userId={userId} db={db} initialized={initialized} />} />
        <Route path="/settings" element={<SettingsPage db={db} userId={userId} initialized={initialized} />} />
        <Route path="/achievements" element={<AchivementsPage db={db} userId={userId} initialized={initialized} />} />
      </Routes>
    </Router>
  );
}

export default App;
```

Рисунок 3.3 – Компонент App

Маршрутизатор налаштовується з параметром `basename="/clicker"`, тому усі маршрути будуть відкриватися з префіксом `/clicker`.

Маршрутизація:

– **/clicker/** – головна сторінка гри (MainPage);
– **/clicker/upgrade** – сторінка з покращеннями (UpgradePage);
– **/clicker/leaderboard** – сторінка з топом гравців (LeaderboardPage);
– **/clicker/shop** – сторінка з магазином скінів (ShopPage);
– **/clicker/achievements** – сторінка з досягненнями (SettingsPage);
– **/clicker/settings** – сторінка з налаштуваннями (AchivcementsPage).

Для кожного шляху задається окремий маршрут, і кожна сторінка отримує необхідні пропси: `userId`, `db` та `initialized`.

3.2.2 Головна сторінка гри (MainPage)

На головній сторінці об’єднано основні функції гри та елементи інтерфейсу. Це головна сторінка застосунку, що показує дані користувача, кількість монет, рівень енергії, а також забезпечує взаємодію з ігровим процесом – виконання кліків, активацію бустів, відновлення енергії та навігацію до інших розділів. Нижче наведено основні функції, які забезпечують працездатність сторінки.

Кешування скінів. Функція cachedSkin визначає кешований скіна користувача з localStorage, який використовується до завантаження даних з Firebase (див. рис. 3.4).

```
const cachedSkin = useMemo(() => {
  try {
    const cached = localStorage.getItem(`mainStats_${userId}`);
    if (cached) {
      const parsed = JSON.parse(cached);
      return parsed.selectedSkin || 'default';
    }
  } catch {}
  return 'default';
}, [userId]);
```

Рисунок 3.4 – Кешування скінів

Предзавантаження зображення. Функція preloadImageWithCache завантажує зображення скіна з кешу.

Збереження стандартного скіна. Функція storeDefaultImageInCache один раз при монтуванні зберігає стандартне зображення кнопки у localStorage як dataURL (див. рис. 3.5).

Лічильник монет. Функція increment обчислює кількість монет, що додається за клік, з урахуванням множника та активного бусту і оновлює лічильники (див. рис. 3.6).

Обробник кліків. Функція handleClick обробляє подію кліку по кнопці кліку. Зменшує енергію, оновлює статистики та викликає анімацію (див. рис. 3.7).

```
useEffect(() => {
  const storeDefaultImageInCache = async () => {
    const existing = localStorage.getItem('clickImage_default');
    if (!existing) {
      const response = await fetch(defaultClickImg);
      const blob = await response.blob();
      const reader = new FileReader();
      reader.onloadend = () => {
        localStorage.setItem('clickImage_default', reader.result);
      };
      reader.readAsDataURL(blob);
    }
  };

  storeDefaultImageInCache();
}, []);
```

Рисунок 3.5 – Збереження стандартного скіна

```
const increment = () => {
  if (energy <= 0) return;

  const effectiveMultiplier = boostActive ? multiplier * 10 : multiplier;

  setCount(prev => prev + effectiveMultiplier);
  setTotalCount(prev => prev + effectiveMultiplier);

  setPendingSave(true);
};
```

Рисунок 3.6 – Лічильник монет

```
const handleClick = (e) => {
  const energyToConsume = multiplier ?? 1;

  if (energy >= energyToConsume || boostActive) {
    if (!boostActive) {
      setEnergy((prev) => Math.max(0, prev - energyToConsume));
    }
    increment();

    const rect = e.currentTarget.getBoundingClientRect();
    const x = e.clientX - rect.left;
    const y = e.clientY - rect.top;

    const finalAmount = boostActive ? Math.floor(energyToConsume * 10) : Math.floor(energyToConsume);

    showFloatingNumber(`+${finalAmount}`, x, y);
  }
};
```

Рисунок 3.7 – Обробник кліків

Лівітуючі цифри. Функція showFloatingNumber створює та додає на сторінку анімований елемент з числом монет, що з’являється при кліку.

Компоненти, необхідні для функціювання сторінки:

– **MainStats**: компонент, що повертає значення статистик та функції для їх оновлення і збереження;
– **SkinsService**: компонент, що завантажує та повертає обраний користувачем скін (selectedSkin), який визначає вигляд кнопки кліку;
– **BoostsService**: компонент, що керує логікою тимчасового бусту та відновлення енергії (повертає статус активного бусту, кулдауни, форматовані таймери та функції activateBoost і handleRegenEnergy).

3.2.3 Головні показники гри (MainStats)

Компонент MainStats відповідає за управління основними статистиками користувача, а саме кількість кліків, загальним рахунком, енергією, множником, іменем користувача та рівнем покращень. Синхронізує дані з Firebase і локальним кешем. Нижче наведено основні функції.

Отримання посилання на документ. Функції getStatsRef та getUserRef: отримують посилання на документ користувача та його статистики в Firebase.

Збереження нікнейму. Функція saveUsername зберігає введене ім’я користувача до Firebase і закриває форму вводу.

Максимальна кількість енергії. Функція getMaxEnergy(level) обчислює максимально доступний рівень енергії на основі рівня покращення.

Швидкість регенерації енергії. Функція getRegenRate(level) визначає швидкість відновлення енергії в залежності від відповідного рівня.

Синхронізація даних. Функція fetchData виконується після ініціалізації компонента (див. рис. 3.8 – 3.10). Завантажує дані користувача з Firebase та локального кешу (localStorage), синхронізує стани, обраховує актуальний рівень енергії з урахуванням часу останнього оновлення. Також кешує актуальні значення в localStorage [8].

```
const fetchData = async () => {
  try {
    const cached = localStorage.getItem(localKey);
    if (cached) {
      const parsed = JSON.parse(cached);
      setCount(parsed.count ?? 0);
      setTotalCount(parsed.totalCount ?? 0);
      setMultiplier(parsed.multiplier ?? 1);
      setUsername(parsed.username ?? '');
      setEnergy(parsed.energy ?? BASE_ENERGY);
      setMaxEnergyLevel(parsed.maxEnergyLevel ?? 1);
      setRegenSpeedLevel(parsed.regenSpeedLevel ?? 1);
    }

    const [
      countDoc, multDoc, totalDoc, usernameDoc, energyDoc,
      maxLevelDoc, regenLevelDoc, userDoc
    ] = await Promise.all([
      getStatsRef('count').get(),
      getStatsRef('multiplier').get(),
      getStatsRef('totalCount').get(),
      getStatsRef('username').get(),
      getStatsRef('energy').get(),
      getStatsRef('maxEnergyLevel').get(),
      getStatsRef('regenSpeedLevel').get(),
      getUserRef().get(),
    ]);
```

Рисунок 3.8

```
const countVal = countDoc.exists ? countDoc.data().value : 0;
const multVal = multDoc.exists ? multDoc.data().value : 1;
const totalVal = totalDoc.exists ? totalDoc.data().value : 0;
const usernameVal = usernameDoc.exists ? usernameDoc.data().value : '';
const maxLevelVal = maxLevelDoc.exists ? maxLevelDoc.data().value : 1;
const regenLevelVal = regenLevelDoc.exists ? regenLevelDoc.data().value : 1;

let energyVal = getMaxEnergy(maxLevelVal);
const regenRate = getRegenRate(regenLevelVal);
if (energyDoc.exists) {
  const energyData = energyDoc.data();
  const storedEnergy = energyData?.value ?? energyVal;

  if (energyData?.lastUpdated instanceof firebase.firestore.Timestamp) {
    const last = energyData.lastUpdated.toMillis();
    const now = Date.now();
    const diffSeconds = Math.floor((now - last) / 1000);
    energyVal = Math.min(getMaxEnergy(maxLevelVal), storedEnergy + diffSeconds * regenRate);
  } else {
    energyVal = storedEnergy;
  }
}

setCount(countVal);
setTotalCount(totalVal);
setMultiplier(multVal);
setEnergy(energyVal);
setMaxEnergyLevel(maxLevelVal);
setRegenSpeedLevel(regenLevelVal);
if (usernameVal) setUsername(usernameVal);
else setShowUsernameForm(true);

const serverTime = firebase.firestore.FieldValue.serverTimestamp();
```

Рисунок 3.9

```
  const serverTime = firebase.firestore.FieldValue.serverTimestamp();

  if (!userDoc.exists)
    await getUserRef().set({ createdAt: serverTime }, { merge: true });
  if (!countDoc.exists)
    await getStatsRef('count').set({ value: 0, createdAt: serverTime });
  if (!multDoc.exists)
    await getStatsRef('multiplier').set({ value: 1, createdAt: serverTime });
  if (!totalDoc.exists)
    await getStatsRef('totalCount').set({ value: 0, createdAt: serverTime });
  if (!energyDoc.exists)
    await getStatsRef('energy').set({ value: energyVal, lastUpdated: serverTime, createdAt: serverTime });
  if (!maxLevelDoc.exists)
    await getStatsRef('maxEnergyLevel').set({ value: 1, createdAt: serverTime });
  if (!regenLevelDoc.exists)
    await getStatsRef('regenSpeedLevel').set({ value: 1, createdAt: serverTime });

  localStorage.setItem(localKey, JSON.stringify({
    count: countVal,
    totalCount: totalVal,
    multiplier: multVal,
    username: usernameVal,
    energy: energyVal,
    maxEnergyLevel: maxLevelVal,
    regenSpeedLevel: regenLevelVal,
  }));

} catch (err) {
  console.error('[STATS] Завантаження помилка:', err);
} finally {
  setLoading(false);
}
```

Рисунок 3.10

Збереження з затримкою. Функція timeout виконує збереження статистик (count, totalCount, energy) до Firebase з невеликою затримкою після змін, щоб уникнути надмірного запису [9].

Відновлення енергії. Функція setEnergy виконує автоматичне відновлення енергії через задані інтервали часу, зберігаючи зміни до Firebase та локального сховища (див. послідовно рис. 3.11 – 3.12).

```
const interval = setInterval(() => {
  setEnergy(prevEnergy => {
    const maxEnergy = getMaxEnergy(maxEnergyLevel);
    const regenRate = getRegenRate(regenSpeedLevel);

    if (prevEnergy >= maxEnergy) return prevEnergy;

    const newEnergy = Math.min(maxEnergy, prevEnergy + regenRate);
    const now = Date.now();

    if (now - lastEnergySaveTime.current >= SAVE_ENERGY_INTERVAL_MS) {
      const serverTime = firebase.firestore.FieldValue.serverTimestamp();
      getStatsRef('energy').set({
```

Рисунок 3.11

```
      getStatsRef('energy').set({
        value: newEnergy,
        updatedAt: serverTime,
        lastUpdated: serverTime
      }, { merge: true });
      lastEnergySaveTime.current = now;

      localStorage.setItem(localKey, JSON.stringify({
        count,
        totalCount,
        multiplier,
        username,
        energy: newEnergy,
        maxEnergyLevel,
        regenSpeedLevel,
      }));
    }

    return newEnergy;
  });
}, ENERGY_REGEN_INTERVAL_MS);
```

Рисунок 3.12

3.2.4 Тимчасові бустери (BoostsService)

Компонент BoostsService відповідає за управління механікою бонусів (бустів) та відновлення енергії у грі. Він контролює активацію бустів, відлік часу їх дії та кулдаунів, а також керує миттєвим відновленням енергії гравця з урахуванням обмежень часу.

BoostsService взаємодіє з базою даних Firebase для збереження і отримання актуального стану бустів та енергії, синхронізує ці дані з локальним сховищем і забезпечує оновлення інтерфейсу користувача в реальному часі. Основні функції наведено нижче.

Активація бустеру. Функція activateBoost активує буст, якщо він ще неактивний і кулдаун завершився (див. рис. 3.13). Встановлює тривалість буста і кулдаун, зберігаючи відповідні часові позначки у Firebase.

Синхронізація даних. Функція fetchData завантажує дані про активні бусти та стан відновлення енергії з Firebase. Вона визначає, чи активний буст, скільки часу залишилось до завершення його дії, а також залишок часу до повторного використання буста. Викликає допоміжну функцію fetchRegenCooldown для завантаження часу кулдауну на миттєве відновлення енергії.

```
const activateBoost = () => {
  const now = Date.now();
  if (boostCooldown > 0 || boostActive) return;

  const boostDuration = 30 * 1000;
  const cooldownDuration = 3600 * 1000;

  const endsAt = now + boostDuration;
  const cooldownEndsAt = endsAt + cooldownDuration;

  setBoostActive(true);
  setBoostRemainingTime(30);
  setBoostCooldown(3600);

  getStatsRef('boost').set({
    active: true,
    boostEndsAt: new Date(endsAt),
    boostCooldownEndsAt: new Date(cooldownEndsAt),
    updatedAt: firebase.firestore.FieldValue.serverTimestamp(),
  }, { merge: true });
};
```

Рисунок 3.13 – Активація бустеру

Кулдаун повного відновлення енергії. Функція fetchRegenCooldown отримує останній час використання функції миттєвого відновлення енергії з Firebase та обчислює залишок часу до можливості повторного використання (див. рис. 3.14). Оновлює стан energyRegenCooldown у компоненті.

```
const fetchRegenCooldown = async () => {
  try {
    const doc = await getStatsRef('energyRegenCooldown').get();
    if (doc.exists) {
      const data = doc.data();
      if (data?.lastRegenTime) {
        const lastTime = data.lastRegenTime;
        const timePassed = Math.floor((Date.now() - lastTime) / 1000);
        const cooldownLeft = Math.max(REGEN_ENERGY_COOLDOWN - timePassed, 0);
        const formattedCooldown = formatTime(cooldownLeft);

        setLastRegenTime(lastTime);
        setEnergyRegenCooldown(cooldownLeft);
      }
    }
  } catch (error) {
    console.error('Failed to fetch regen cooldown:', error);
  }
};
```

Рисунок 3.14 – Кулдаун повного відновлення енергії

Повне відновлення енергії. Функція handleRegenEnergy викликається при натисканні на кнопку миттєвого відновлення енергії (див. рис. 3.15). Якщо

кулдаун завершився та енергія не повна, встановлює енергію на максимальне значення, зберігає нові значення у Firebase та оновлює кулдаун.

```
const handleRegenEnergy = async () => {
  if (energy >= getMaxEnergy(maxEnergyLevel)) return;
  const now = Date.now();
  const cooldownEnded = !lastRegenTime || (now - lastRegenTime >= REGEN_ENERGY_COOLDOWN * 1000);
  if (!cooldownEnded) return;

  const serverTime = firebase.firestore.FieldValue.serverTimestamp();
  const maxEnergy = getMaxEnergy(maxEnergyLevel);

  setEnergy(maxEnergy);
  setLastRegenTime(now);
  setEnergyRegenCooldown(REGEN_ENERGY_COOLDOWN);

  await getStatsRef('energy').set({
    value: maxEnergy,
    updatedAt: serverTime,
    lastUpdated: serverTime,
  }, { merge: true });

  await getStatsRef('energyRegenCooldown').set({
    lastRegenTime: now
  }, { merge: true });
};
```

Рисунок 3.15 – Повне відновлення енергії

3.2.5 Сторінка покращень (UpgradePage)

Сторінка покращень дозволяє гравцю переглядати доступні покращення, їх рівень та вартість наступного рівня. Також є можливість придбати покращення, які вливають на різні характеристики гри, за монети.

Для функціювання сторінки використано сервіс UpgradeService. Він містить усю логіку сторінки покращень. Також для відображення кількості монет використано MainStats.

3.2.6 Компонент покращень (UpgradeService)

Для прискорення завантаження сторінки було створено локальні стани для рахунку, множника, максимального вмісту та швидкості регенерації енергії, а саме localCount, LocalMultiplier, setLocalMaxEnergy та setLocalRegenSpeed відповідно.

До кожного стану було створено функцію useEffect, що автоматично оновлюють локальні значення, якщо відповідні значення змінюються у Firebase. Також було створено функцію getUpgradeCost, яка обчислює вартість покращення за рівнем характеристик. Основні функції наведено нижче.

Придбання покращень. Функція handleBuy, яка відповідає за купівлю апгрейду (див. рис. 3.16). Якщо в користувача достатньо монет, придбається покращення, оновлює Firebase та локальний стан. Якщо монет недостатньо, показує модальне вікно з помилкою.

```
const handleBuy = async (type) => {
  const upgrades = {
    multiplier: {
      level: localMultiplier,
      cost: multiplierCost,
      onUpgrade: async (newLevel) => {
        await db.collection('users').doc(String(userId)).collection('stats').doc('multiplier')
          .set({ value: newLevel, updatedAt: new Date() }, { merge: true });
        setMultiplier(newLevel);
        setLocalMultiplier(newLevel);
      }
    },
    maxEnergy: {
      level: localMaxEnergy,
      cost: maxEnergyCost,
      onUpgrade: async (newLevel) => {
        await db.collection('users').doc(String(userId)).collection('stats').doc('maxEnergyLevel')
          .set({ value: newLevel, updatedAt: new Date() }, { merge: true });
        setMaxEnergyLevel(newLevel);
        setLocalMaxEnergy(newLevel);
      }
    },
    regenSpeed: {
      level: localRegenSpeed,
      cost: regenSpeedCost,
      onUpgrade: async (newLevel) => {
        await db.collection('users').doc(String(userId)).collection('stats').doc('regenSpeedLevel')
          .set({ value: newLevel, updatedAt: new Date() }, { merge: true });
        setRegenSpeedLevel(newLevel);
        setLocalRegenSpeed(newLevel);
      }
    }
  };

  const upgrade = upgrades[type];
  const cost = upgrade.cost;

  if (localCount >= cost) {
    const newLevel = upgrade.level + 1;
    await Promise.all([
      db.collection('users').doc(String(userId)).collection('stats').doc('count')
        .set({ value: localCount - cost, updatedAt: new Date() }, { merge: true }),
      upgrade.onUpgrade(newLevel)
    ]);
    setCount(localCount - cost);
    setLocalCount(localCount - cost);
  } else {
    setShowErrorModal(true);
  }
};
```

Рисунок 3.16 – Придбання покращень

3.2.7 Сторінка магазину (ShopPage)

Сторінка магазину скінів дозволяє гравцю переглядати доступні скіни, їх вартість, статус придбання, а також обирати або купувати нові скіни за монети. Деякі скіни є нагородою за досягнення, і їх можна отримати лише при виконанні певних умов.

Для функціювання сторінки використано компонент MainStats для відображення кількості монет, та компонент SkinService, який відповідає за логіку магазину скінів.

3.2.8 Компонент магазину скінів (SkinService)

Компонент SkinsService відповідає за логіку роботи магазину скінів, а саме зберігання, вибір та купівлю скінів. Він обробляє дані користувача з бази даних Firebase, зберігає поточний стан у локальне сховище браузера та оновлює інтерфейс відповідно до дій гравця. Нижче наведено функції.

Завантаження локальних даних. Функція fetchData відповідає за початкове завантаження локальних даних користувача з localStorage. Якщо дані існують, встановлює selectedSkin. Додатково виконує запит до Firebase для перевірки наявності користувача. Після завантаження – зберігає дані назад у локальне сховище та завершує ініціалізацію, знімаючи стан завантаження.

Завантаження даних про скіни. Функція fetchSkins відповідає за завантаження списку придбаних користувачем скінів (ownedSkins) та активного скіна (selectedSkin) з Firebase. Якщо записи відсутні, створює їх зі значенням за замовчуванням (default). Отримані дані зберігаються у стані компонента та використовуються для визначення доступності та вигляду елементів у магазині.

Придбання скінів. Функція handleBuySkin відповідає за придбання або вибір скіна (див. рис. 3.17). Якщо скін уже придбано, встановлюється як

активний. Якщо користувач має достатньо монет, скін купується, оновлюються дані у Firebase. Якщо монет недостатньо, показується модальне вікно з помилкою.

```
const handleBuySkin = async (skin) => {

  if (ownedSkins.includes(skin.id)) {
    await db.collection('users').doc(String(userId)).collection('stats').doc('selectedSkin')
      .set({ value: skin.id, updatedAt: new Date() }, { merge: true });
    setSelectedSkin(skin.id);
    return;
  }

  if (achievementSkins.includes(skin.id)) {
    return;
  }

  if (count >= skin.cost) {
    const newBalance = count - skin.cost;
    const newOwned = [...ownedSkins, skin.id];

    await Promise.all([
      db.collection('users').doc(String(userId)).collection('stats').doc('count')
        .set({ value: newBalance, updatedAt: new Date() }, { merge: true }),
      db.collection('users').doc(String(userId)).collection('stats').doc('ownedSkins')
        .set({ value: newOwned, updatedAt: new Date() }, { merge: true }),
      db.collection('users').doc(String(userId)).collection('stats').doc('selectedSkin')
        .set({ value: skin.id, updatedAt: new Date() }, { merge: true })
    ]);

    setCount(newBalance);
    setOwnedSkins(newOwned);
    setSelectedSkin(skin.id);
  } else {
    setShowErrorModal(true);
  }
};
```

Рисунок 3.17 – Придбання скінів

3.2.9 Топ гравців (LeaderboardPage)

Сторінка дозволяє користувачу переглядати глобальний рейтинг гравців за кількістю зароблених кліків. Вона відображає топ-10 найкращих гравців, а також поточну позицію користувача, якщо він не входить до цього списку.

Основна логіка сторінки реалізована у компоненті LeaderboardService, де отримуються та форматуються рейтингові дані, а саме інформація про топ гравців та поточного користувача.

3.2.10 Компонент топу гравців (LeaderboardService)

Компонент LeaderboardService відповідає за логіку роботи рейтингової таблиці користувачів. Він отримує дані про гравців із бази даних Firebase, обробляє інформацію про їхні імена та зароблені монети, сортує гравців за результатами та формує список для відображення топ-10 користувачів та поточного гравця. Компонент також обробляє стан завантаження даних. Нижче наведено функції.

Формування списку для відображення. Логіка, що формує displayList, створює список для відображення топ-10 гравців та інформацію про поточного користувача, якщо він не входить у топ-10 (див. рис. 3.18 та 3.19). Якщо дані ще завантажуються, відображає заглушки.

```
if (loading) {
    displayList = {
        top10: Array.from({ length: 10 }, (_, i) => ({
            userId: `loading-${i}`,
            username: '...',
            totalCount: '...',
            place: i + 1,
            isCurrentUser: false,
        })),
        currentUser: null,
    };
} else {
    const fullList = leaders.map((user, index) => ({
        ...user,
        place: index + 1,
        isCurrentUser: user.userId == userId,
    }));
```

Рисунок 3.18 – Формування списку для відображення

```
    const top10 = fullList.slice(0, 10);

    const currentUserInTop = top10.some(u => u.userId == userId);
    const currentUser = currentUserInTop
        ? null
        : fullList.find(user => user.userId == userId);

    displayList = {
        top10,
        currentUser,
    };
}
```

Рисунок 3.19 – Формування списку для відображення

Завантаження даних для топу гравців. Функція fetchLeaderboard відповідає за отримання списку всіх користувачів та їх статистику з Firebase. Потім збирає, сортує та зберігає в локальний стан список гравців, відсортований за кількістю монет.

3.2.11 Сторінка досягнень (AchievementsPage)

На сторінці досягнень відображається перелік ігрових цілей, які користувач може виконати. На ній реалізовано інтерфейс, де гравець бачить список досягнень, що дозволяє мотивувати його до подальшої активності у грі.

Основна логіка сторінки реалізована у компоненті AchievementsService, де зчитуються виконані досягнення конкретного користувача, та повертається масив досягнень, що вже виконані.

3.2.12 Компонент сторінки досягнень (AchievementsService)

Компонент AchievementsService відповідає за логіку роботи досягнень користувача у грі. Він визначає, які досягнення були виконані на основі статистики гравця (кліки, рівні покращень), зберігає цей прогрес у Firebase та автоматично відкриває відповідні нагороди. Нижче наведено основні функції компоненту.

Завантаження досягнень. Функція fetchData відповідає за отримання списку досягнень При ініціалізації компоненту отримує поточні досягнення з Firebase і зберігає їх у локальний стан.

Оновлення досягнень. Функція updateAchievements відповідає за автоматичне оновлення досягнень на основі прогресу користувача (див. рис. 3.20). Якщо користувач виконав нові умови, то досягнення додаються в Firebase, і відкриваються нові скіни (при виконанні деяких досягнень).

```
const updateAchievements = async () => {
  try {
    const serverTime = firebase.firestore.FieldValue.serverTimestamp();

    const achievementsRef = getStatsRef('achievements');
    const achievementsDoc = await achievementsRef.get();
    const existingAchievements = achievementsDoc.exists ? achievementsDoc.data().value || [] : [];

    const unlocked = ACHIEVEMENTS
      .filter(a => a.condition({ totalCount, energy, maxEnergyLevel, regenSpeedLevel, multiplier }))
      .map(a => a.id);

    const newlyUnlocked = unlocked.filter(id => !existingAchievements.includes(id));

    if (newlyUnlocked.length > 0) {
      await achievementsRef.set({
        value: firebase.firestore.FieldValue.arrayUnion(...newlyUnlocked),
        updatedAt: serverTime
      }, { merge: true });

      setAchievements([...existingAchievements, ...newlyUnlocked]);
    } else {
      setAchievements(existingAchievements);
    }

    const unlockedSkins = newlyUnlocked
      .map(id => ACHIEVEMENT_SKIN_REWARDS[id])
      .filter(Boolean)
      .filter(skinId => !ownedSkins.includes(skinId));

    if (unlockedSkins.length > 0) {
      const ownedSkinsRef = getStatsRef('ownedSkins');
      await ownedSkinsRef.set({
        value: firebase.firestore.FieldValue.arrayUnion(...unlockedSkins),
        updatedAt: serverTime
      }, { merge: true });

      setOwnedSkins([...ownedSkins, ...unlockedSkins]);
    }

  } catch (err) {
    console.error('[ACHIEVEMENTS] Помилка при оновленні:', err);
  }
};
```

Рисунок 3.20 – Оновлення досягнень

3.2.13 Сторінка налаштувань (SettingsPage)

Компонент сторінки SettingsPage відповідає за інтерфейс налаштувань користувача, зокрема відображення ідентифікатора користувача та зміну нікнейму.

Дані завантажуються через хук MainStats, з якого отримуються поточний нікнейм, стан модального вікна, а також функції для відкриття та закриття модального вікна і збереження нового нікнейму.

3.3 Взаємодія користувача з грою

Після першого запуску гри користувачу пропонується ввести його нікнейм (див. рис. 3.21).

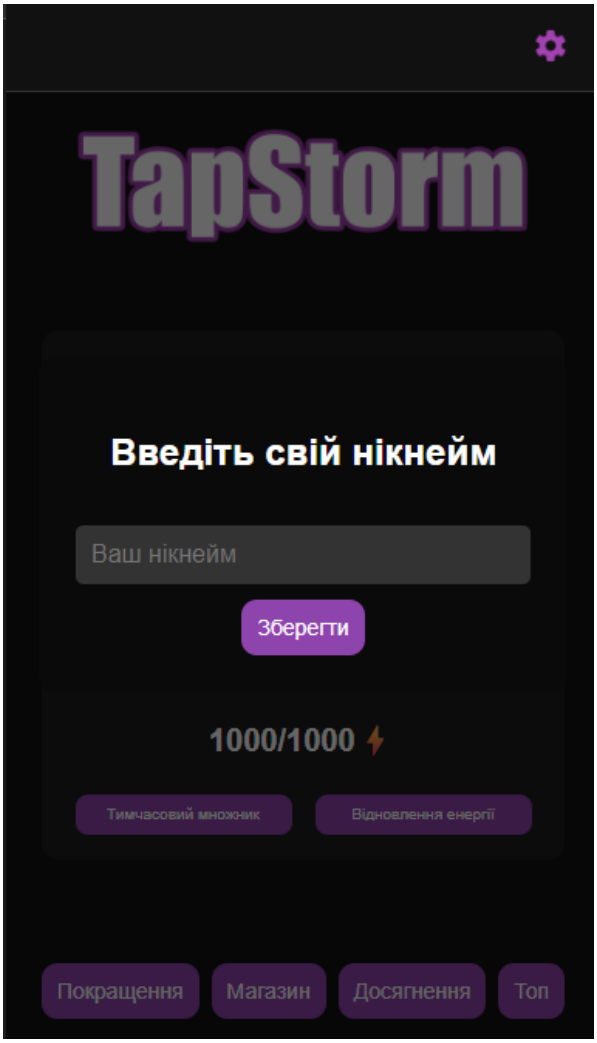


Рисунок 3.21 – Меню введення нікнейму

Коли користувач ввів свій нікнейм і натиснув кнопку «Зберегти», він потрапляє на головну сторінку гри. Вона містить нікнейм гравця, кнопку налаштувань, логотип, кількість монет, кнопка-зображення, кількість енергії, бустери та кнопки для навігації між сторінками (див. рис. 3.22).

При натисканні кнопки-зображення до балансу додаються монет, відображається анімація плаваючих чисел, та віднімається доступна енергія (див. рис. 3.23).

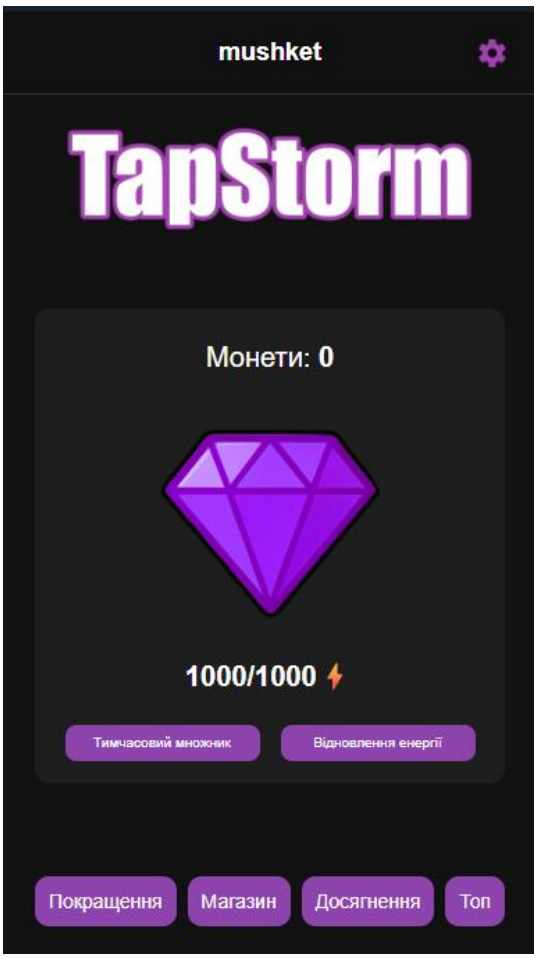


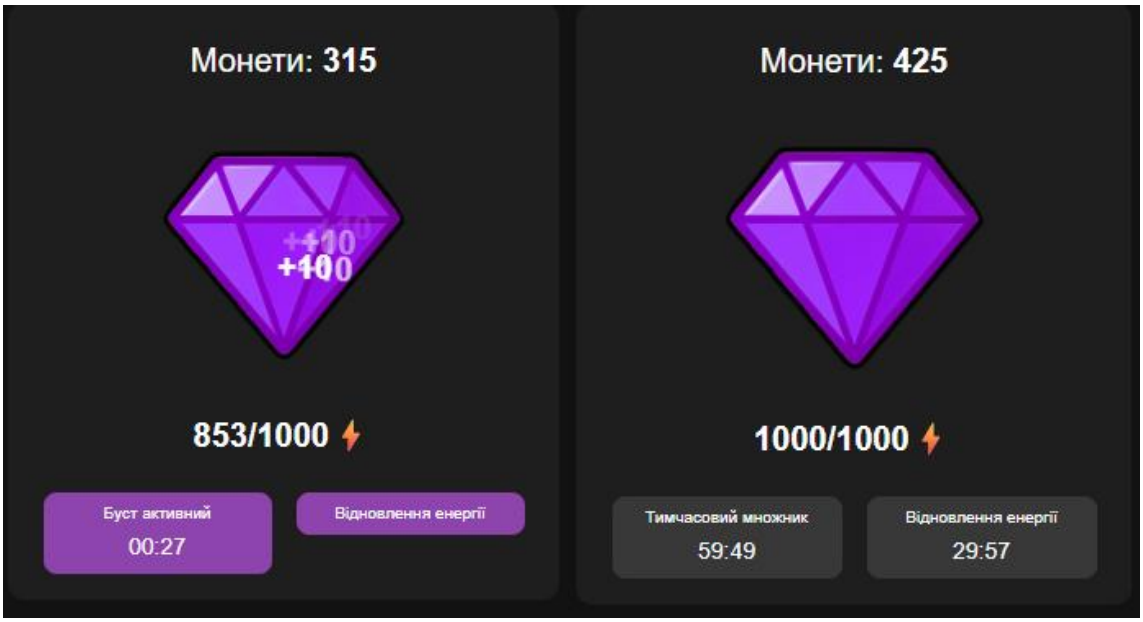
Рисунок 3.22 – Головна сторінка гри



Рисунок 3.23 – Анімація плаваючих чисел

При натисканні на кнопку «Тимчасовий множник», кількість монет при одному натисканні помножується на 10 на протязі 30 секунд. Коли час спливає, кнопка стає неактивною протягом години. При натисканні на кнопку

«Відновлення енергії», енергія відновлюється до максимального значення, та кнопка стає неактивною протягом 30 хвилин (див. рис. 3.24).



а) Активний тимчасовий множник б) Неактивні кнопки бустерів

Рисунок 3.24 – Тимчасові бустери

При натисканні на копку налаштувань, відкривається сторінка, на якій відображено ID аккаунту, та є можливість змінити нікнейм. (див. рис. 3.25).

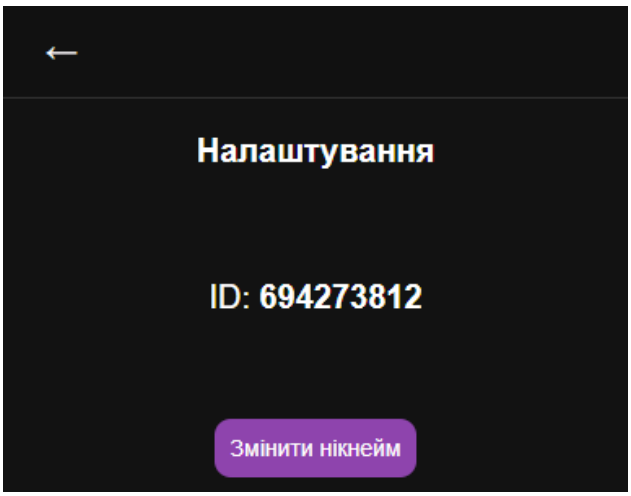


Рисунок 3.25 – Сторінка налаштувань

Якщо натиснути кнопку «Змінити нікнейм», то відкриється вікно, у якому є можливість відредагувати нікнейм. При натисканні кнопки

«Зберегти», новий нікнейм буде збережений, а якщо натиснути кнопку «Скасувати», то зміни не будуть внесені (див. рис. 3.26).

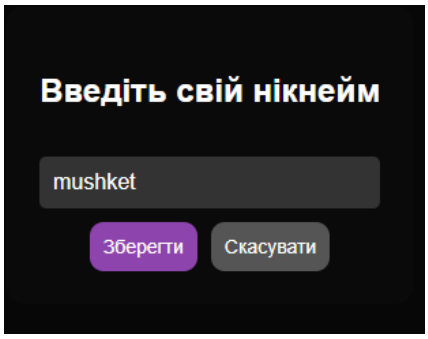


Рисунок 3.26 – Вікно зміни нікнейму

Якщо в головному меню натиснути кнопку «Покращення», то відкриється сторінка з покращеннями ігрових характеристик, таких як множник кліку, максимальна кількість енергії та регенерація енергії та їх рівень. Також на сторінці відображену поточний баланс монет. При натисканні на зображення покращень, їх рівень буде збільшуватись (див. рис. 3.27).



Рисунок 3.27 – Сторінка покращень

Якщо буде недостатньо монет для придбання покращень, на екрані з’явиться відповідне повідомлення (див. рис. 3.28).

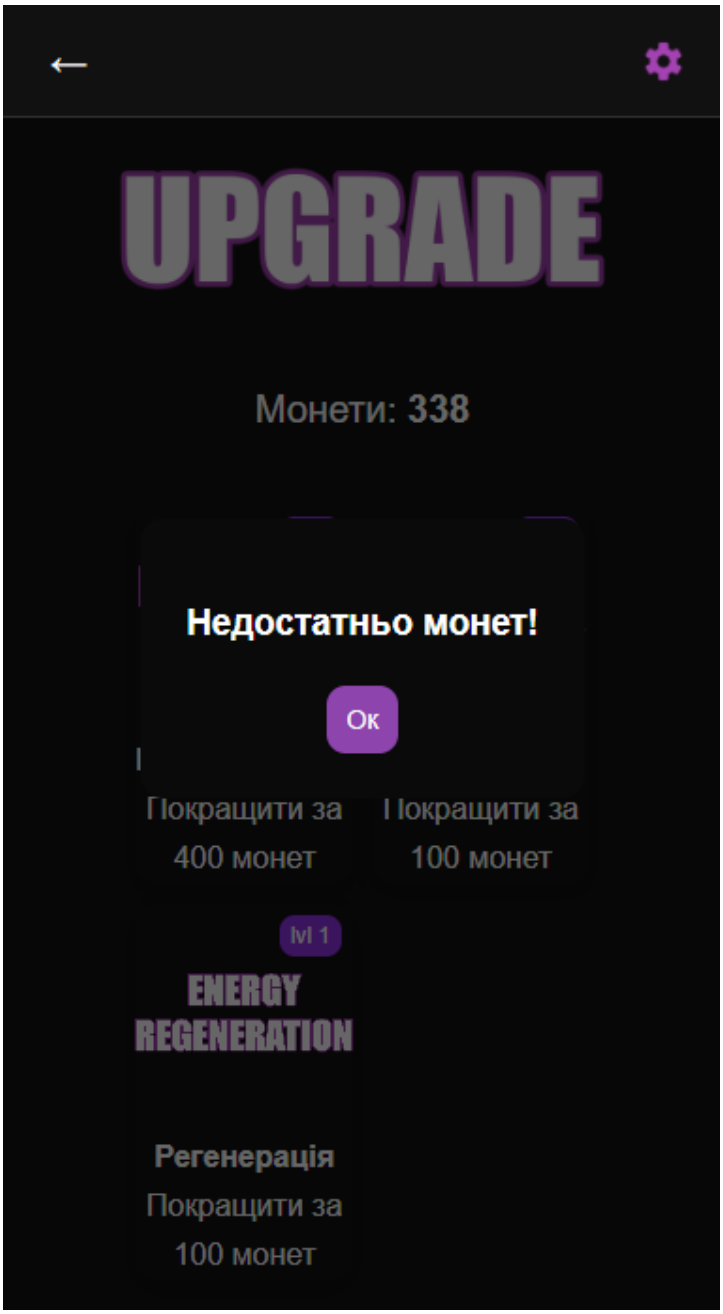


Рисунок 3.28 – Повідомлення про недостатню кількість монет

Якщо користувач натисне в головному меню кнопку «Магазин», то відкриється сторінка з магазином скінів (див. рис. 3.29). На ній можливо придбати за монети, або вибрати вже куплений скін.

Аналогічно до сторінки з покращеннями, якщо на балансі не буде достатньої кількості монет, то на екрані з’явиться відповідне повідомлення.



Рисунок 3.29 – Сторінка з магазином скінів

Якщо користувач натисне кнопку «Досягнення», то відкриється сторінка зі списком усіх виконаних на не виконаних досягнень (див. рис. 3.30). При отримані деяких досягнень розблоковуються скіни на кнопку.

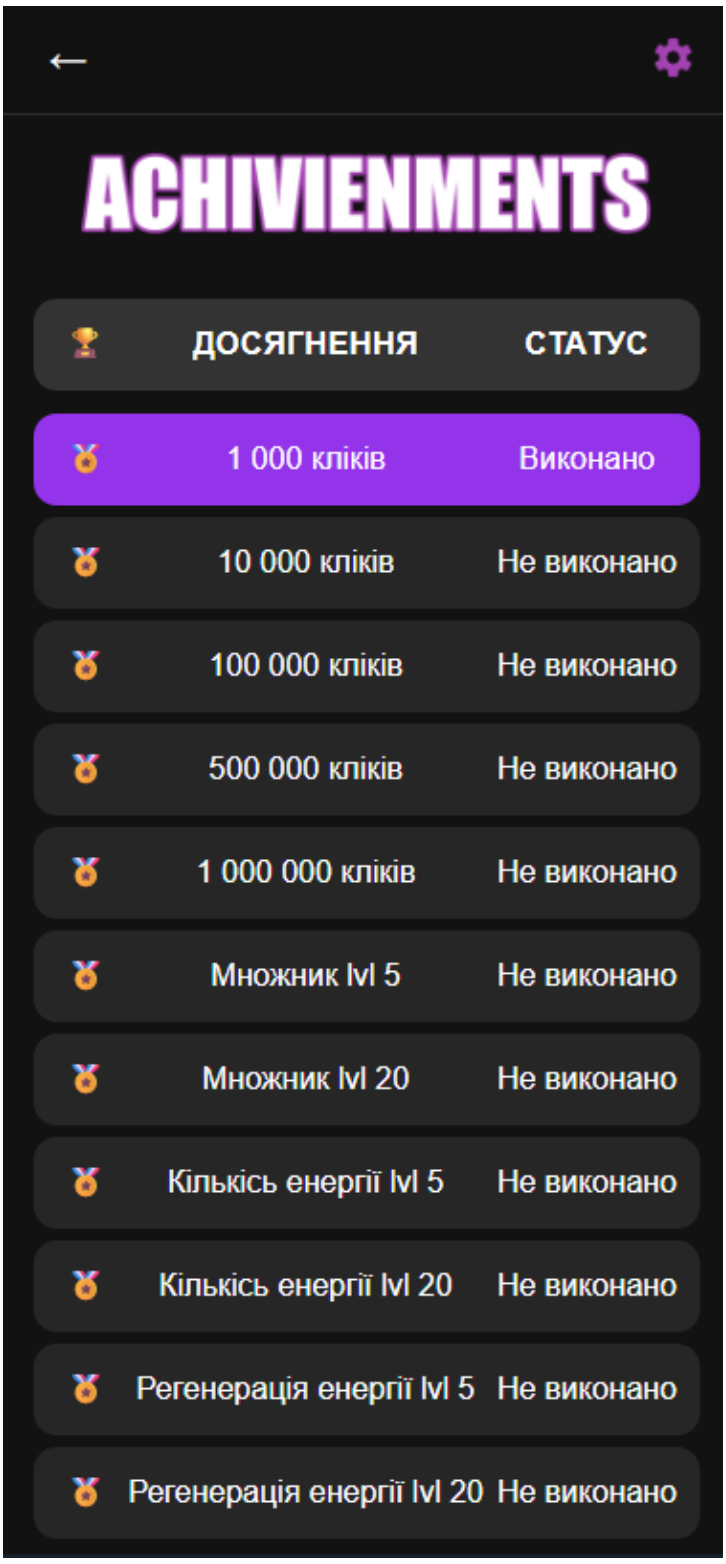


Рисунок 3.30 – Сторінка досягнень

Якщо на головній сторінці користувач натисне кнопку «Топ», то відкриється сторінка з топом гравців (див. рис. 3.31). Список відсортований за загальною кількістю зароблених монет. Поточне місце гравця відображається фіолетовим кольором.



Рисунок 3.31 – Сторінка з топом гравців

3.4 Висновки до розділу 3

Отже, було детально розглянуто технічну реалізацію клікер-гри як Telegram Mini-App із використанням React та Firebase. Проєкт демонструє сучасний підхід до розробки вебзастосунків, орієнтованих на інтеграцію з популярними платформами, такими як Telegram. Основними досягненнями розділу є:

– **ініціалізація середовища**: реалізовано хуки useTelegramInit та useFirebase, які забезпечують безпечне отримання даних користувача

з Telegram та з’єднання з Firebase для подальшої роботи з хмарною базою даних;

- **модульна структура**: побудовано набір взаємопов’язаних компонентів (App, MainPage, UpgradePage, ShopPage, тощо), кожен з яких відповідає за окрему функціональність інтерфейсу;
- **ігрова логіка**: впроваджено основну механіку гри, включаючи кліки, монети, енергію, бусти, покращення, досягнення, кешування даних, анімації та взаємодію з інтерфейсом у реальному часі;
- **синхронізація даних**: забезпечено стабільну синхронізацію між локальним станом і Firebase, що дозволяє зберігати прогрес гравця та уникати втрати даних;
- **оптимізація продуктивності**: застосовано локальні стани, кешування зображень, відкладене збереження даних до Firebase для зниження навантаження на мережу та покращення швидкодії застосунку.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було реалізовано інтерактивну гру жанру «клікер» у форматі Telegram Mini-App, що підтвердило можливість ефективного застосування сучасних вебтехнологій для створення динамічного, зручного та персоналізованого ігрового застосунку безпосередньо в екосистемі месенджера.

У ході дослідження було досягнуто поставленої мети та виконано всі поставлені завдання:

- проведено аналіз можливостей створення застосунків у форматі Telegram Mini-Apps та вивчено специфіку використання Telegram WebApp API для автентифікації користувачів і доступу до персоналізованих даних;
- розроблено архітектуру клікер-гри з модульною побудовою, що включає компоненти для управління статистикою, кліками, енергією, бустерами, магазином скінів, досягненнями та покращеннями;
- реалізовано інтерфейс на основі бібліотеки React, що забезпечило компонентну структуру, повторне використання логіки через кастомні хуки та гнучке управління станом програми;
- для зберігання ігрових даних, синхронізації прогресу та обробки користувацької інформації застосовано хмарну платформу Firebase – вона забезпечує обробку у реальному часі, кешування локальних змін, оптимізовані механізми читання/запису даних і можливість розширення проєкту;
- реалізовано механізм збереження ігрової сесії, автоматичної синхронізації прогресу, таймери відновлення ресурсів та обрахунок ефектів від бустерів із підтримкою кулдаунів;
- здійснено розгортання застосунку на платформі GitHub Pages, що дозволяє користувачам миттєво запускати гру в браузері Telegram без інсталяції.

У результаті виконаної роботи отримано повнофункціональну гру, що інтегрується в месенджер, демонструючи приклад сучасної веброзробки із застосуванням React, Firebase і Telegram WebApp API.

Під час написання роботи було вдосконалено комплекс практичних навичок, що охоплюють розробку сучасних вебзастосунків, інтеграцію з зовнішніми сервісами та API, проєктування ігрової логіки та забезпечення її стабільної роботи в інтерактивному середовищі.

Отже, результати виконаної кваліфікаційної роботи підтверджують доцільність і ефективність використання сучасних вебтехнологій для створення інтерактивних ігрових застосунків у форматі Telegram Mini-App.

ПЕРЕЛІК ПОСИЛАНЬ

1. Telegram Mini-Apps – офіційна документація. URL : https://core.telegram.org/bots/webapps (дата звернення: 18.04.2025).

2. Firebase – офіційна документація. URL : https://firebase.google.com/docs (дата звернення: 18.04.2025).

3. GitHub Pages – публікація вебзастосунків. URL : https://pages.github.com/ (дата звернення: 18.04.2025).

4. React – офіційна документація. URL : https://reactjs.org (дата звернення: 18.04.2025).

5. Salunke M. Mastering React JS From Development to Deployment: Become React JS Expert. 2024. 387 p. URL : https://a.co/d/eCqkGVi (дата звернення: 18.04.2025).

6. React Router – офіційна документація. URL : https://reactrouter.com/en/main (дата звернення: 18.04.2025).

7. StudioD21 Smart Tech Content, Rodrigues D. LEARN FIREBASE: Integrate Real-Time Backend for Web and Mobile Applications : 8th ed. *Quick Learn Series*. 2025. 232 p. URL : https://a.co/d/7uMpwlh (дата звернення: 18.04.2025).

8. LocalStorage API – документація по збереженню даних у локальний стан. URL : https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage (дата звернення: 18.04.2025).

9. JavaScript Timer Functions – документація щодо функцій таймерів. URL : https://developer.mozilla.org/en-US/docs/Web/API/Window/setTimeout (дата звернення: 18.04.2025).

ДОДАТОК А

Код основних компонентів гри-клікер

A.1 MainStats.js

```
import { useState, useEffect, useRef } from 'react';
import firebase from 'firebase/compat/app';
import 'firebase/compat/firestore';

const BASE_ENERGY = 1000;
const BASE_REGEN = 1;
const ENERGY_REGEN_INTERVAL_MS = 1000;
const SAVE_ENERGY_INTERVAL_MS = 10000;
export function MainStats({ db, userId, initialized }) {
  const [loading, setLoading] = useState(true);
  const [count, setCount] = useState(0);
  const [totalCount, setTotalCount] = useState(0);
  const [multiplier, setMultiplier] = useState(1);
  const [username, setUsername] = useState('');
  const [energy, setEnergy] = useState(BASE_ENERGY);
  const [maxEnergyLevel, setMaxEnergyLevel] = useState(1);
  const [regenSpeedLevel, setRegenSpeedLevel] = useState(1);
  const [showUsernameForm, setShowUsernameForm] = useState(false);
  const [pendingSave, setPendingSave] = useState(false);
  const localKey = `mainStats_${userId}`;
  const isReady = userId && db && initialized;
  const lastEnergySaveTime = useRef(Date.now());
  const getStatsRef = (field) =>
db.collection('users').doc(String(userId)).collection('stats').doc(field);
  const getUserRef = () => db.collection('users').doc(String(userId));
  const getMaxEnergy = (level) => BASE_ENERGY * level;
  const getRegenRate = (level) => BASE_REGEN * level;

  useEffect(() => {
    if (!isReady) return;
    const fetchData = async () => {
      try {
        const cached = localStorage.getItem(localKey);
        if (cached) {
          const parsed = JSON.parse(cached);
          setCount(parsed.count ?? 0);
          setTotalCount(parsed.totalCount ?? 0);
          setMultiplier(parsed.multiplier ?? 1);
```

```
        setUsername(parsed.username ?? '');
        setEnergy(parsed.energy ?? BASE_ENERGY);
        setMaxEnergyLevel(parsed.maxEnergyLevel ?? 1);
        setRegenSpeedLevel(parsed.regenSpeedLevel ?? 1);
      }
      const [
        countDoc, multDoc, totalDoc, usernameDoc, energyDoc,
        maxLevelDoc, regenLevelDoc, userDoc
      ] = await Promise.all([
        getStatsRef('count').get(),
        getStatsRef('multiplier').get(),
        getStatsRef('totalCount').get(),
        getStatsRef('username').get(),
        getStatsRef('energy').get(),
        getStatsRef('maxEnergyLevel').get(),
        getStatsRef('regenSpeedLevel').get(),
        getUserRef().get(),
      ]);
      const countVal = countDoc.exists ? countDoc.data().value : 0;
      const multVal = multDoc.exists ? multDoc.data().value : 1;
      const totalVal = totalDoc.exists ? totalDoc.data().value : 0;
      const usernameVal = usernameDoc.exists ? usernameDoc.data().value : '';
      const maxLevelVal = maxLevelDoc.exists ? maxLevelDoc.data().value : 1;
      const regenLevelVal = regenLevelDoc.exists ? regenLevelDoc.data().value : 1;

      let energyVal = getMaxEnergy(maxLevelVal);
      const regenRate = getRegenRate(regenLevelVal);
      if (energyDoc.exists) {
        const energyData = energyDoc.data();
        const storedEnergy = energyData?.value ?? energyVal;

        if (energyData?.lastUpdated instanceof firebase.firestore.Timestamp) {
          const last = energyData.lastUpdated.toMillis();
          const now = Date.now();
          const diffSeconds = Math.floor((now - last) / 1000);
          energyVal = Math.min(getMaxEnergy(maxLevelVal), storedEnergy +
diffSeconds * regenRate);
        } else {
          energyVal = storedEnergy;
        }
      }

      setCount(countVal);
      setTotalCount(totalVal);
      setMultiplier(multVal);
      setEnergy(energyVal);
      setMaxEnergyLevel(maxLevelVal);
      setRegenSpeedLevel(regenLevelVal);
```

```
            if (usernameVal) setUsername(usernameVal);
            else setShowUsernameForm(true);

            const serverTime = firebase.firestore.FieldValue.serverTimestamp();
            if (!userDoc.exists)
              await getUserRef().set({ createdAt: serverTime }, { merge: true });
            if (!countDoc.exists)
              await getStatsRef('count').set({ value: 0, createdAt: serverTime });
            if (!multDoc.exists)
              await getStatsRef('multiplier').set({ value: 1, createdAt: serverTime });
            if (!totalDoc.exists)
              await getStatsRef('totalCount').set({ value: 0, createdAt: serverTime });
            if (!energyDoc.exists)
              await getStatsRef('energy').set({ value: energyVal, lastUpdated: serverTime,
createdAt: serverTime });
            if (!maxLevelDoc.exists)
              await getStatsRef('maxEnergyLevel').set({ value: 1, createdAt: serverTime });
            if (!regenLevelDoc.exists)
              await getStatsRef('regenSpeedLevel').set({ value: 1, createdAt: serverTime });

            localStorage.setItem(localKey, JSON.stringify({
              count: countVal,
              totalCount: totalVal,
              multiplier: multVal,
              username: usernameVal,
              energy: energyVal,
              maxEnergyLevel: maxLevelVal,
              regenSpeedLevel: regenLevelVal,
            }));

          } catch (err) {
            console.error('[STATS] Завантаження помилка:', err);
          } finally {
            setLoading(false);
          }
        };

        fetchData();
      }, [isReady]);

      useEffect(() => {
        if (!isReady) return;

        const interval = setInterval(() => {
          setEnergy(prevEnergy => {
            const maxEnergy = getMaxEnergy(maxEnergyLevel);
            const regenRate = getRegenRate(regenSpeedLevel);
```

```
      if (prevEnergy >= maxEnergy) return prevEnergy;

      const newEnergy = Math.min(maxEnergy, prevEnergy + regenRate);
      const now = Date.now();

      if (now - lastEnergySaveTime.current >= SAVE_ENERGY_INTERVAL_MS) {
        const serverTime = firebase.firestore.FieldValue.serverTimestamp();
        getStatsRef('energy').set({
          value: newEnergy,
          updatedAt: serverTime,
          lastUpdated: serverTime
        }, { merge: true });
        lastEnergySaveTime.current = now;

        localStorage.setItem(localKey, JSON.stringify({
          count,
          totalCount,
          multiplier,
          username,
          energy: newEnergy,
          maxEnergyLevel,
          regenSpeedLevel,
        }));
      }

      return newEnergy;
    });
  }, ENERGY_REGEN_INTERVAL_MS);

  return () => clearInterval(interval);
}, [
  isReady,
  count,
  totalCount,
  multiplier,
  username,
  maxEnergyLevel,
  regenSpeedLevel,
]);

useEffect(() => {
  if (!pendingSave || !isReady) return;

  const timeout = setTimeout(() => {
    const serverTime = firebase.firestore.FieldValue.serverTimestamp();
    const nowTimestamp = firebase.firestore.Timestamp.now();

    Promise.all([
```

```
            getStatsRef('count').set({ value: count, updatedAt: serverTime }, { merge: true }),
            getStatsRef('totalCount').set({ value: totalCount, updatedAt: serverTime }, { merge:
true }),
            getStatsRef('energy').set({
              value: energy,
              updatedAt: serverTime,
              lastUpdated: nowTimestamp,
            }, { merge: true }),
          ]).then(() => setPendingSave(false));

          localStorage.setItem(localKey, JSON.stringify({
            count,
            totalCount,
            multiplier,
            username,
            energy,
            maxEnergyLevel,
            regenSpeedLevel,
          }));
        }, 200);

        return () => clearTimeout(timeout);
      }, [
        pendingSave,
        count,
        totalCount,
        energy,
        isReady,
      ]);

      const saveUsername = async (usernameInput) => {
        if (!usernameInput.trim()) return;
        const serverTime = firebase.firestore.FieldValue.serverTimestamp();
        await getStatsRef('username').set({
          value: usernameInput,
          updatedAt: serverTime,
          createdAt: serverTime
        });
        setUsername(usernameInput);
        setShowUsernameForm(false);
      };

      return {
        loading,
        setLoading,
        count,
```

```
    totalCount,
    multiplier,
    username,
    showUsernameForm,
    setShowUsernameForm,
    saveUsername,
    energy,
    setEnergy,
    maxEnergyLevel,
    setTotalCount,
    regenSpeedLevel,
    setMultiplier,
    setMaxEnergyLevel,
    setRegenSpeedLevel,
    setCount,
    setPendingSave,
    getMaxEnergy,
    maxEnergyLevel
  };
}
```

A.2 BoostsService.js

```
import { useState, useEffect, useRef } from 'react';
import firebase from 'firebase/compat/app';
import { MainStats } from '../hooks/MainStats';
import 'firebase/compat/firestore';

const REGEN_ENERGY_COOLDOWN = 1800;

export function BoostsService({ db, userId, initialized }) {
  const isReady = userId && db && initialized;
  const [energyRegenCooldown, setEnergyRegenCooldown] = useState(0);
  const [boostActive, setBoostActive] = useState(false);
  const [boostRemainingTime, setBoostRemainingTime] = useState(0);
  const [boostCooldown, setBoostCooldown] = useState(0);
  const [lastRegenTime, setLastRegenTime] = useState(null);
  const { loading, setLoading, energy, getMaxEnergy, maxEnergyLevel, setEnergy } =
MainStats({ db, userId, initialized });
  const localKey = `mainStats_${userId}`;
  const getStatsRef = (field) =>
    db.collection('users').doc(String(userId)).collection('stats').doc(field);
  const getUserRef = () => db.collection('users').doc(String(userId));
```

```
useEffect(() => {
  if (!isReady) return;

  const fetchData = async () => {
    try {

      const [
        boostDoc
      ] = await Promise.all([
        getStatsRef('boost').get(),

      ]);
      const [
      ] = await Promise.all([
        getUserRef().get(),
      ]);

      const boostData = boostDoc.exists ? boostDoc.data() : {};
      const now = Date.now();
      let boostStillActive = false;
      let newBoostRemaining = 0;
      let newBoostCooldown = 0;

      if (boostData.boostEndsAt?.toMillis) {
        const endsAt = boostData.boostEndsAt.toMillis();
        const remaining = Math.floor((endsAt - now) / 1000);
        boostStillActive = remaining > 0;
        newBoostRemaining = Math.max(remaining, 0);
      }

      if (boostData.boostCooldownEndsAt?.toMillis) {
        const cooldownEndsAt = boostData.boostCooldownEndsAt.toMillis();
        const remainingCooldown = Math.floor((cooldownEndsAt - now) / 1000);
        newBoostCooldown = Math.max(remainingCooldown, 0);
      }

      setBoostActive(boostStillActive);
      setBoostRemainingTime(newBoostRemaining);
      setBoostCooldown(newBoostCooldown);
      await fetchRegenCooldown();

    } catch (err) {
      console.error('[STATS] Завантаження помилка:', err);
    } finally {
      setLoading(false);
    }
  };
```

```
  fetchData();
}, [isReady]);

useEffect(() => {
  if (!isReady || !boostActive) return;

  const updateBoost = async () => {
    const serverTime = firebase.firestore.FieldValue.serverTimestamp();

    await getStatsRef('boost').set({
      active: boostActive,
      remainingTime: boostRemainingTime,
      cooldown: boostCooldown,
      updatedAt: serverTime,
    }, { merge: true });

    localStorage.setItem(localKey, JSON.stringify({
      boostActive,
      boostRemainingTime,
      boostCooldown
    }));
  };

  updateBoost();
}, [boostActive, boostRemainingTime, boostCooldown, isReady]);

const activateBoost = () => {
  const now = Date.now();
  if (boostCooldown > 0 || boostActive) return;

  const boostDuration = 30 * 1000;
  const cooldownDuration = 3600 * 1000;

  const endsAt = now + boostDuration;
  const cooldownEndsAt = endsAt + cooldownDuration;

  setBoostActive(true);
  setBoostRemainingTime(30);
  setBoostCooldown(3600);

  getStatsRef('boost').set({
    active: true,
    boostEndsAt: new Date(endsAt),
    boostCooldownEndsAt: new Date(cooldownEndsAt),
    updatedAt: firebase.firestore.FieldValue.serverTimestamp(),
  }, { merge: true });
};
```

```
  useEffect(() => {
    const interval = setInterval(() => {
      if (boostActive) {
        setBoostRemainingTime(prev => Math.max(prev - 1, 0));
        if (boostRemainingTime <= 1) {
          setBoostActive(false);
        }
      } else if (boostCooldown > 0) {
        setBoostCooldown(prev => Math.max(prev - 1, 0));
      }
    }, 1000);

    return () => clearInterval(interval);
  }, [boostActive, boostRemainingTime, boostCooldown]);

  const formatTime = (seconds) => {
    const m = Math.floor(seconds / 60).toString().padStart(2, '0');
    const s = (seconds % 60).toString().padStart(2, '0');
    return `${m}:${s}`;
  };

  const formattedBoostTime = formatTime(boostRemainingTime);
  const formattedCooldown = formatTime(boostCooldown);

  const fetchRegenCooldown = async () => {
    try {
      const doc = await getStatsRef('energyRegenCooldown').get();
      if (doc.exists) {
        const data = doc.data();
        if (data?.lastRegenTime) {
          const lastTime = data.lastRegenTime;
          const timePassed = Math.floor((Date.now() - lastTime) / 1000);
          const cooldownLeft = Math.max(REGEN_ENERGY_COOLDOWN - timePassed,
0);
          const formattedCooldown = formatTime(cooldownLeft);

          setLastRegenTime(lastTime);
          setEnergyRegenCooldown(cooldownLeft);
        }
      }
    } catch (error) {
      console.error('Failed to fetch regen cooldown:', error);
    }
  };

  const formattedenergyCooldown = formatTime(energyRegenCooldown);

  const handleRegenEnergy = async () => {
```

```
        if (energy >= getMaxEnergy(maxEnergyLevel)) return;

        const now = Date.now();
        const cooldownEnded = !lastRegenTime || (now - lastRegenTime >=
REGEN_ENERGY_COOLDOWN * 1000);
        if (!cooldownEnded) return;

        const serverTime = firebase.firestore.FieldValue.serverTimestamp();
        const maxEnergy = getMaxEnergy(maxEnergyLevel);

        setEnergy(maxEnergy);
        setLastRegenTime(now);
        setEnergyRegenCooldown(REGEN_ENERGY_COOLDOWN);

        await getStatsRef('energy').set({
          value: maxEnergy,
          updatedAt: serverTime,
          lastUpdated: serverTime,
        }, { merge: true });

        await getStatsRef('energyRegenCooldown').set({
          lastRegenTime: now
        }, { merge: true });

      };

      useEffect(() => {
        const interval = setInterval(() => {
          setEnergyRegenCooldown(prev => Math.max(prev - 1, 0));
        }, 1000);
        return () => clearInterval(interval);
      }, []);

      return {
        boostActive,
        boostRemainingTime,
        boostCooldown,
        activateBoost,
        formattedBoostTime,
        formattedCooldown,
        energyRegenCooldown,
        handleRegenEnergy,
        lastRegenTime,
        formatTime,
        formattedenergyCooldown,
        setEnergyRegenCooldown
      };
    }
```

A.3 UpgradeService.js

```
import { useState, useEffect } from 'react';
import { MainStats } from '../hooks/MainStats';
import multiplierImg from '../assets/multiplierclicks.png';
import maxEnergyImg from '../assets/maxenergy.png';
import regenSpeedImg from '../assets/energyregeneration.png';

export function UpgradeService({ userId, db, initialized }) {
  const {
    count,
    multiplier,
    username,
    setCount,
    setMultiplier,
    maxEnergyLevel,
    regenSpeedLevel,
    setMaxEnergyLevel,
    setRegenSpeedLevel
  } = MainStats({ db, userId, initialized });

  const [showErrorModal, setShowErrorModal] = useState(false);

  const [localCount, setLocalCount] = useState(count);
  const [localMultiplier, setLocalMultiplier] = useState(multiplier);
  const [localMaxEnergy, setLocalMaxEnergy] = useState(maxEnergyLevel);
  const [localRegenSpeed, setLocalRegenSpeed] = useState(regenSpeedLevel);

  useEffect(() => setLocalCount(count), [count]);
  useEffect(() => setLocalMultiplier(multiplier), [multiplier]);
  useEffect(() => setLocalMaxEnergy(maxEnergyLevel), [maxEnergyLevel]);
  useEffect(() => setLocalRegenSpeed(regenSpeedLevel), [regenSpeedLevel]);

  const getUpgradeCost = (level, base) => Math.floor(base * Math.pow(level, 2));
  const multiplierCost = getUpgradeCost(localMultiplier || 1, 100);
  const maxEnergyCost = getUpgradeCost(localMaxEnergy || 1, 100);
  const regenSpeedCost = getUpgradeCost(localRegenSpeed || 2, 100);

  const handleBuy = async (type) => {
    const upgrades = {
      multiplier: {
        level: localMultiplier,
        cost: multiplierCost,
        onUpgrade: async (newLevel) => {
          await db.collection('users').doc(String(userId)).collection('stats').doc('multiplier')
            .set({ value: newLevel, updatedAt: new Date() }, { merge: true });
```

```
            setMultiplier(newLevel);
            setLocalMultiplier(newLevel);
           }
          },
          maxEnergy: {
           level: localMaxEnergy,
           cost: maxEnergyCost,
           onUpgrade: async (newLevel) => {
            await
db.collection('users').doc(String(userId)).collection('stats').doc('maxEnergyLevel')
              .set({ value: newLevel, updatedAt: new Date() }, { merge: true });
            setMaxEnergyLevel(newLevel);
            setLocalMaxEnergy(newLevel);
           }
          },
          regenSpeed: {
           level: localRegenSpeed,
           cost: regenSpeedCost,
           onUpgrade: async (newLevel) => {
            await
db.collection('users').doc(String(userId)).collection('stats').doc('regenSpeedLevel')
              .set({ value: newLevel, updatedAt: new Date() }, { merge: true });
            setRegenSpeedLevel(newLevel);
            setLocalRegenSpeed(newLevel);
           }
          }
         };

         const upgrade = upgrades[type];
         const cost = upgrade.cost;

         if (localCount >= cost) {
          const newLevel = upgrade.level + 1;
          await Promise.all([
           db.collection('users').doc(String(userId)).collection('stats').doc('count')
            .set({ value: localCount - cost, updatedAt: new Date() }, { merge: true }),
           upgrade.onUpgrade(newLevel)
          ]);
          setCount(localCount - cost);
          setLocalCount(localCount - cost);
         } else {
          setShowErrorModal(true);
         }
        };

        const upgrades = [
         {
          type: 'multiplier',
```

```
    title: 'Множник кліку',
    level: localMultiplier,
    cost: multiplierCost,
    bg: multiplierImg,
    desc: `Покращити за\n ${multiplierCost} монет`
   },
   {
    type: 'maxEnergy',
    title: 'Макс. енергія',
    level: localMaxEnergy,
    cost: maxEnergyCost,
    bg: maxEnergyImg,
    desc: `Покращити за\n ${maxEnergyCost} монет`
   },
   {
    type: 'regenSpeed',
    title: 'Регенерація',
    level: localRegenSpeed,
    cost: regenSpeedCost,
    bg: regenSpeedImg,
    desc: `Покращити за\n ${regenSpeedCost} монет`
   },
  ];

   return {
   upgrades,
   showErrorModal,
   handleBuy,
   localCount,
   localMultiplier,
   localMaxEnergy,
   localRegenSpeed,
   setShowErrorModal,
  };
}
```

A.4 SkinService.js

```
import { useState, useEffect, useRef } from 'react';
import firebase from 'firebase/compat/app';
import { MainStats } from '../hooks/MainStats';
import 'firebase/compat/firestore';
import defaultClickImg from '../assets/diamondpurple.png';
import blueSkin from '../assets/bluediamond.png';
```

```
import pinkSkin from '../assets/pinkdiamond.png';
import rubySkin from '../assets/rubydiamond.png';
import greenSkin from '../assets/greendiamond.png';
import orangeSkin from '../assets/orangediamond.png';
import gradient1Skin from '../assets/diamondgradiend1.png';
import gradient2Skin from '../assets/diamondgradiend2.png';
import gradient3Skin from '../assets/diamondgradiend3.png';

export function SkinsService({ db, userId, initialized }) {
  const isReady = userId && db && initialized;
  const { count, setCount, setLoading } = MainStats({ db, userId, initialized });
  const [ownedSkins, setOwnedSkins] = useState([]);
  const [selectedSkin, setSelectedSkin] = useState(null);
  const localKey = `mainStats_${userId}`;

  const getStatsRef = (field) =>
    db.collection('users').doc(String(userId)).collection('stats').doc(field);
  const getUserRef = () => db.collection('users').doc(String(userId));

  useEffect(() => {
    if (!isReady) return;

    const fetchData = async () => {
      try {
        const cached = localStorage.getItem(localKey);
        if (cached) {
          const parsed = JSON.parse(cached);
          setSelectedSkin(parsed.selectedSkin ?? null);
        }

        const [
        ] = await Promise.all([
          getUserRef().get(),
        ]);

        localStorage.setItem(localKey, JSON.stringify({
          selectedSkin
        }));

      } catch (err) {
        console.error('[STATS] Завантаження помилка:', err);
      } finally {
        setLoading(false);
      }
    };
```

```
  fetchData();
}, [isReady]);

useEffect(() => {
  if (!isReady) return;

  const cached = localStorage.getItem(localKey);
  if (cached) {
    const parsed = JSON.parse(cached);
    localStorage.setItem(localKey, JSON.stringify({
      ...parsed,
      selectedSkin
    }));
  }
}, [selectedSkin, isReady]);

useEffect(() => {
  if (!isReady) return;

  const fetchSkins = async () => {
    try {
      const ownedSkinsRef = getStatsRef('ownedSkins');
      const selectedSkinRef = getStatsRef('selectedSkin');
      const [ownedSkinsDoc, selectedSkinDoc] = await Promise.all([
        ownedSkinsRef.get(),
        selectedSkinRef.get()
      ]);

      const serverTime = firebase.firestore.FieldValue.serverTimestamp();

      if (!ownedSkinsDoc.exists) {
        await ownedSkinsRef.set({
          value: ['default'],
          createdAt: serverTime
        });
        setOwnedSkins(['default']);
      } else {
        setOwnedSkins(ownedSkinsDoc.data().value || []);
      }

      if (!selectedSkinDoc.exists) {
        await selectedSkinRef.set({
          value: 'default',
          createdAt: serverTime
        });
        setSelectedSkin('default');
      } else {
        setSelectedSkin(selectedSkinDoc.data().value || null);
```

```
      }
    } catch (err) {
      console.error('[STORE] Помилка при завантаженні скінів:', err);
    }
  };

  fetchSkins();
}, [isReady]);

const [showErrorModal, setShowErrorModal] = useState(false);

const skins = [
  { id: 'default', title: 'Фіолетовий', image: defaultClickImg, cost: 0 },
  { id: 'blue', title: 'Блакитний', image: blueSkin, cost: 10000 },
  { id: 'pink', title: 'Рожевий', image: pinkSkin, cost: 10000 },
  { id: 'ruby', title: 'Рубін', image: rubySkin, cost: 10000 },
  { id: 'green', title: 'Зелений', image: greenSkin, cost: 10000 },
  { id: 'orange', title: 'Помаранчевий', image: orangeSkin, cost: 10000 },
  { id: 'gradient1', title: 'Градієнт', image: gradient1Skin },
  { id: 'gradient2', title: 'Градієнт', image: gradient2Skin },
  { id: 'gradient3', title: 'Градієнт', image: gradient3Skin },
];

const achievementSkins = ['gradient1', 'gradient2', 'gradient3'];

const handleBuySkin = async (skin) => {

  if (ownedSkins.includes(skin.id)) {
    await db.collection('users').doc(String(userId)).collection('stats').doc('selectedSkin')
      .set({ value: skin.id, updatedAt: new Date() }, { merge: true });
    setSelectedSkin(skin.id);
    return;
  }

  if (achievementSkins.includes(skin.id)) {
    return;
  }

  if (count >= skin.cost) {
    const newBalance = count - skin.cost;
    const newOwned = [...ownedSkins, skin.id];

    await Promise.all([
      db.collection('users').doc(String(userId)).collection('stats').doc('count')
        .set({ value: newBalance, updatedAt: new Date() }, { merge: true }),
      db.collection('users').doc(String(userId)).collection('stats').doc('ownedSkins')
        .set({ value: newOwned, updatedAt: new Date() }, { merge: true }),
      db.collection('users').doc(String(userId)).collection('stats').doc('selectedSkin')
```

```
        .set({ value: skin.id, updatedAt: new Date() }, { merge: true })
    ]);

    setCount(newBalance);
    setOwnedSkins(newOwned);
    setSelectedSkin(skin.id);
   } else {
    setShowErrorModal(true);
   }
  };
  return {
   skins,
   showErrorModal,
   handleBuySkin,
   ownedSkins,
   selectedSkin
  };
}
```

A.5 LeaderboardService.js

```
import { useState, useEffect, useRef } from 'react';
import firebase from 'firebase/compat/app';
import 'firebase/compat/firestore';
import { MainStats } from '../hooks/MainStats';

export function LeaderboardService({ db, userId, initialized }) {
   const [leaders, setLeaders] = useState([]);
   const [loading, setLoading] = useState(true);
   const isReady = db && initialized;

   const {
      username,
      totalCount,
      multiplier,
      loading: statsLoading,
   } = MainStats({ db, userId, initialized });

   useEffect(() => {
      if (!isReady) return;

      const fetchLeaderboard = async () => {
         try {
            const usersSnap = await db.collection('users').get();
```

```
            const results = [];

            for (const userDoc of usersSnap.docs) {
               const statsRef = db.collection('users').doc(userDoc.id).collection('stats');
               const statsSnap = await statsRef.get();
               const data = {};
               statsSnap.forEach(doc => {
                  data[doc.id] = doc.data().value;
               });
               const updates = [];
               if (data.username === undefined) {
                  const generatedName = `Гравець-${userDoc.id.slice(0, 4)}`;
                  updates.push(statsRef.doc('username').set({ value: generatedName }));
                  data.username = generatedName;
               }
               if (data.totalCount === undefined) {
                  updates.push(statsRef.doc('totalCount').set({ value: 0 }));
                  data.totalCount = 0;
               }
               if (updates.length > 0) {
                  await Promise.all(updates);
               }
               results.push({
                  userId: userDoc.id,
                  username: data.username,
                  totalCount: data.totalCount,
               });
            }
            results.sort((a, b) => b.totalCount - a.totalCount);
            setLeaders(results);
         } catch (err) {
            console.error('[ЛІДЕРБОРД] Помилка завантаження:', err);
         } finally {
            setLoading(false);
         }
      };
      fetchLeaderboard();
   }, [isReady]);

   let displayList = {};

   if (loading) {
      displayList = {
         top10: Array.from({ length: 10 }, (_, i) => ({
            userId: `loading-${i}`,
            username: '...',
            totalCount: '...',
            place: i + 1,
```

```
                isCurrentUser: false,
            })),
            currentUser: null,
          };
        } else {
          const fullList = leaders.map((user, index) => ({
            ...user,
            place: index + 1,
            isCurrentUser: user.userId == userId,
          }));

          const top10 = fullList.slice(0, 10);

          const currentUserInTop = top10.some(u => u.userId == userId);
          const currentUser = currentUserInTop
            ? null
            : fullList.find(user => user.userId == userId);

          displayList = {
            top10,
            currentUser,
          };
        }

        return {
          loading,
          displayList
        };
      }
```

A.6 AchievementsService.js

```
      import { useState, useEffect, useRef } from 'react';
      import firebase from 'firebase/compat/app';
      import { MainStats } from '../hooks/MainStats';
      import { SkinsService } from '../hooks/SkinsService';
      import 'firebase/compat/firestore';

      const ACHIEVEMENTS = [
        { id: 'click_1000', label: '1 000 кліків', condition: ({ totalCount }) => totalCount >=
1000 },
        { id: 'click_10000', label: '10 000 кліків', condition: ({ totalCount }) => totalCount >=
10000 },
        { id: 'click_100000', label: '100 000 кліків', condition: ({ totalCount }) => totalCount
```

```
>= 100000 },
        { id: 'click_500000', label: '500 000 кліків', condition: ({ totalCount }) => totalCount
>= 500000 },
        { id: 'click_1000000', label: '1 000 000 кліків', condition: ({ totalCount }) => totalCount
>= 1000000 },
        { id: 'upgrade_multiplier_lvl5', label: 'Множник lvl 5', condition: ({ multiplier }) =>
multiplier >= 5 },
        { id: 'upgrade_multiplier_lvl120', label: 'Множник lvl 20', condition: ({ multiplier }) =>
multiplier >= 15 },
        { id: 'upgrade_energy_lvl5', label: 'Енергія рівень 5', condition: ({ maxEnergyLevel })
=> maxEnergyLevel >= 5 },
        { id: 'upgrade_energy_lvl20', label: 'Енергія рівень 20', condition: ({ maxEnergyLevel
}) => maxEnergyLevel >= 20 },
        { id: 'upgrade_regen_lvl5', label: 'Регенерація рівень 20', condition: ({
regenSpeedLevel }) => regenSpeedLevel >= 5 },
        { id: 'upgrade_regen_lvl20', label: 'Регенерація рівень 20', condition: ({
regenSpeedLevel }) => regenSpeedLevel >= 20 },
      ];

      const ACHIEVEMENT_SKIN_REWARDS = {
        'click_1000000': 'gradient1',
        'upgrade_multiplier_lvl120': 'gradient2',
        'upgrade_energy_lvl20': 'gradient3',
      };

      export function AchievementsService({ db, userId, initialized }) {
        const { totalCount, multiplier, maxEnergyLevel, regenSpeedLevel, energy, setLoading }
= MainStats({ db, userId, initialized });
        const { ownedSkins, setOwnedSkins } = SkinsService({ db, userId, initialized });
        const [achievements, setAchievements] = useState([]);
        const isReady = userId && db && initialized;

        const getStatsRef = (field) =>
          db.collection('users').doc(String(userId)).collection('stats').doc(field);
        const getUserRef = () => db.collection('users').doc(String(userId));
        useEffect(() => {
          if (!isReady) return;

          const fetchData = async () => {
            try {

              const [
                achievementsDoc
              ] = await Promise.all([
                getStatsRef('achievements').get(),
                getUserRef().get(),

              ]);
```

```
            const achievementsVal = achievementsDoc.exists ? achievementsDoc.data().value ||
[] : [];

            setAchievements(achievementsVal);

          } catch (err) {
            console.error('[STATS] Завантаження помилка:', err);
          } finally {
            setLoading(false);
          }
        };

        fetchData();
      }, [isReady]);

      useEffect(() => {
        if (!isReady || !totalCount) return;

        const updateAchievements = async () => {
          try {
            const serverTime = firebase.firestore.FieldValue.serverTimestamp();

            const achievementsRef = getStatsRef('achievements');
            const achievementsDoc = await achievementsRef.get();
            const existingAchievements = achievementsDoc.exists ?
achievementsDoc.data().value || [] : [];

            const unlocked = ACHIEVEMENTS
              .filter(a => a.condition({ totalCount, energy, maxEnergyLevel, regenSpeedLevel,
multiplier }))
              .map(a => a.id);

            const newlyUnlocked = unlocked.filter(id => !existingAchievements.includes(id));

            if (newlyUnlocked.length > 0) {
              await achievementsRef.set({
                value: firebase.firestore.FieldValue.arrayUnion(...newlyUnlocked),
                updatedAt: serverTime
              }, { merge: true });
              setAchievements([...existingAchievements, ...newlyUnlocked]);
            } else {
              setAchievements(existingAchievements);
            }
            const unlockedSkins = newlyUnlocked
              .map(id => ACHIEVEMENT_SKIN_REWARDS[id])
              .filter(Boolean)
              .filter(skinId => !ownedSkins.includes(skinId));
            if (unlockedSkins.length > 0) {
```

```
            const ownedSkinsRef = getStatsRef('ownedSkins');
            await ownedSkinsRef.set({
              value: firebase.firestore.FieldValue.arrayUnion(...unlockedSkins),
              updatedAt: serverTime
            }, { merge: true });
            setOwnedSkins([...ownedSkins, ...unlockedSkins]);
          }
        } catch (err) {
          console.error('[ACHIEVEMENTS] Помилка при оновленні:', err);
        }
      };

      updateAchievements();
    }, [totalCount, isReady, energy, maxEnergyLevel, regenSpeedLevel, multiplier,
ownedSkins]);

    return {
      achievements
    };
  }
```