Міністерство освіти і науки України
Луцький національний технічний університет
Факультет комп'ютерних та інформаційних технологій
Кафедра інженерії програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «БАКАЛАВР»

РЕАЛІЗАЦІЯ ВЕБСАЙТУ ДЛЯ СПІЛКУВАННЯ ТА ІНТЕРАКТИВНИХ АКТИВНОСТЕЙ У РЕАЛЬНОМУ ЧАСІ ЗА ДОПОМОГОЮ ПРОТОКОЛУ WEBSOCKET

IMPLEMENTATION OF A WEBSITE FOR COMMUNICATION AND INTERACTIVE ACTIVITIES IN REAL TIME USING WEBSOCKET PROTOCOL

спеціальність 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

освітня програма «Інженерія програмного забезпечення»
(назва освітньої програми)

Виконав: здобувач вищої освіти
групи ІПЗс-31
Григола Владислав Андрійович
_____
(підпис)

Керівник:
к.т.н., доцент
Суринович Олена Миколаївна
_____
(підпис)

Кваліфікаційну роботу
допущено до захисту
«8» 06 2023 р.
Гарант освітньої програми:
к.т.н., доцент
Ліщина Наталія Миколаївна
_____
(підпис)

Луцьк – 2023 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет *комп'ютерних та інформаційних технологій*
Кафедра *інженерії програмного забезпечення*
Ступінь вищої освіти: *бакалавр*
Галузь знань: *12 Інформаційні технології*
Спеціальність: *121 Інженерія програмного забезпечення*
Освітня програма: *«Інженерія програмного забезпечення»*

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____
«_02_» _____02_____ 2023 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

Григолі Владиславу Андрійовичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи: Реалізація вебсайту для спілкування та інтерактивних активностей у реальному часі за допомогою протоколу WebSocket

Керівник роботи: к.т.н., професор Суринович Олена Миколаївна
затверджені наказом закладу вищої освіти від «*23*» *грудня 2022* р. № *961-01-02*
2. Строк подання здобувачем вищої освіти кваліфікаційної роботи *«8» червня 2023 р.*
3. Вихідні дані до роботи: *технічне та програмне забезпечення, вимоги до розробки програмного забезпечення, область для проведення аналізу сучасного стану проблеми. Мова програмування серверної та клієнтської частин – TypeScript, Технології програмування клієнтської частини – uWebSockets.js Next.js, React, HTML, SCSS, MUI.*
4. Зміст розрахунково-пояснювальної записки (перелік питань, що потрібно розробити): *Аналіз сучасного стану проблеми, існуючих методів і засобів її розв'язання, аналіз і вибір засобів проєктування, бібліотек, протоколів, опис функціонального наповнення об'єкта проєктування, прототипування, розробка й обґрунтування системного наповнення, опис алгоритмів, підходів до реалізації поставленої задачі, організація структур даних, розробка інтерфейсів, тестування розробленої системи на відповідність функціональним вимогам*
5. Перелік графічного матеріалу: *16 рисунків.*

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз предметної області	*Суринович О. М.*		
Специфікація вимог до розробленої системи	*Суринович О. М.*		
Розробка об'єкта проєктування	*Суринович О. М*		
Нормоконтроль	*Суринович О. М.*		
Гарант ОП	*Ліщина Н. М.*		
Показник запозичень тексту	9,59%		
Академічна доброчесність	*Яцук А.А.*		

7. Дата видачі завдання *«2» лютого 2023 р.*

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи бакалавра	Строк виконання етапів роботи	Примітки
1	*Огляд літературних джерел по темі кваліфікаційної роботи бакалавра*	*05.02.2023 р.*	вик.
2	*Аналіз проблеми розробки та впровадження об'єкту проєктування*	*27.02.2023 р.*	вик.
3	*Обґрунтування вибору шляхів, технологій (алгоритмів) і засобів вирішення поставленого завдання*	*10.03.2023 р.*	вик.
4	*Розробка функціонально-структурної схеми роботи об'єкта проєктування та проєктування бази даних*	*17.04.2023 р.*	вик.
5	*Практична реалізація об'єкта проєктування та розробка бази даних*	*18.05.2023 р.*	вик.
6	*Тестування та налагодження об'єкта проєктування*	*01.06.2023*	вик.
7	*Здача чистового варіанту кваліфікаційної роботи бакалавра на кафедру*	*08.06.2023 р.*	вик.

Здобувач вищої освіти _____ Трисюра В.А.
(підпис) (прізвище, ініціали)

Керівник кваліфікаційної роботи _____ Суринович О.М.
(підпис) (прізвище, ініціали)

Міністерство освіти і науки України
Луцький національний технічний університет

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Здобувач вищої освіти: *Григола Владислав Андрійович*
(ПІБ)

Тема кваліфікаційної роботи:
Реалізація вебсайту для спілкування та інтерактивних активностей у реальному часі за допомогою протоколу WebSocket.
(повна назва згідно наказу)

Коротка характеристика кваліфікаційної роботи та прийнятих рішень:
Робота відрізняється високим рівнем теоретичної основи. Проведено докладний аналіз існуючих платформ та технологій, поставлено завдання та встановлено специфікацію вимог до розроблюваної системи. Предмет дослідження чітко відповідає темі, а розроблена система коректно реалізовує створені алгоритми та методи вирішення поставлених задач.

Самостійні розробки і пропозиції автора:
Автор зосередився на створенні системи, яка забезпечує ефективну комунікацію та рекреаційні активності реального часу через вебдодаток. Було розроблено алгоритми та конфігурації для розгортання вебсайту, обробки ігрових подій, інтерактивні активності із низькою затримкою, методи управління подіями на основі часових параметрів. Також було розроблено прототипи та реалізовано доступний дизайн додатку. Тестування системи відповідає сучасним стандартам програмної інженерії.

Практичне значення роботи:
Робота має практичну цінність, враховуючи зростаючу популярність онлайн комунікацій та активностей. Автор звернувся до сучасних технологій та методів програмування. Розроблені алгоритми вирішення поставлених завдань релевантні та можуть бути використані у інших проєктах.

Недоліки:
Демонстрація деяких реалізованих алгоритмів є достатньою у рамках роботи, проте може бути доповнена блок-схемами для додаткової ясності.

Загальний висновок:
Загалом, робота є вдалим поєднанням теоретичного дослідження та практичної реалізації. Вона доводить здатність автора до самостійної розробки та застосування сучасних технологій. Робота може бути оцінена як відмінна.

Рецензент _____ (к.т.н., доц. доц. каф. КН Лукінчук Ю.І.)
(підпис) (науковий ступінь, вчене звання, посада, ПІБ)

«8» червня 2023 р.

АНОТАЦІЯ

Григола В. А. Реалізація вебсайту для спілкування та інтерактивних активностей в реальному часі за допомогою протоколу WebSocket. Рукопис.

Кваліфікаційна робота бакалавра за спеціальністю 121 Інженерія програмного забезпечення. Луцький національний технічний університет. Луцьк, 2023. 59 с.

Кваліфікаційна робота присвячена розробці вебсайту для спілкування та інтерактивних активностей в реальному часі за допомогою протоколу WebSocket. Робота складається з трьох розділів, в яких проаналізовані проблематика та поставлені завдання за темою роботи, визначені вимоги до розроблюваної системи та обрані засоби для її реалізації, описана практична реалізація вебсайту з використанням Next.js та uWebSocket.js, проведене тестування та налагодження системи, а також описані заходи щодо SEO оптимізації та аналізу перспективи розробки. Результати роботи можуть бути корисними для розробників вебдодатків та спеціалістів у галузі інженерії програмного забезпечення.

Ключові слова: вебсайт, JavaScript, HTTP, WebSocket, Nginx, Next.js, uWebSocket.js, Socket.io, MaterialUI, реальний час, інтерактивні активності.

ABSTRACT

Hryhola V. A. Implementation of a website for communication and interactive activities in real time using WebSocket protocol. Manuscript.

Bachelor's thesis. in the specialty of 121 Software Engineering. Lutsk National Technical University. Lutsk, 2023, 59 p.

The bachelor's thesis is devoted to the development of a website for communication and interactive activities in real time using WebSocket protocol. The work consists of three chapters that analyze the problem and define the tasks, determine the requirements for the developed system and select the means for its implementation, describe the practical implementation of the website using Next.js and uWebSocket.js, conduct testing and debugging of the system, and also describe measures for SEO optimization and potential of the project. The results of the work can be useful for web application developers and specialists in software engineering.

Keywords: website, JavaScript, HTTP, WebSocket, Nginx, Next.js, uWebSocket.js, Socket.io, MaterialUI, real-time, interactive activities.

ЗМІСТ

ВСТУП

У сучасному світі інформаційних технологій та швидкого розвитку Інтернету, вебтехнології зазнають значних змін і вдосконалень. Однією з найбільш перспективних та актуальних тем у сфері програмної інженерії є розробка вебсайту для спілкування та інтерактивних активностей у реальному часі. Використання протоколу WebSocket, який дозволяє забезпечити двостороннє зв'язування між клієнтом та сервером, створює нові можливості для веброзробки та ґеймдев (від англ. video game development – розробка відеоігор) індустрії.

Метою кваліфікаційної роботи бакалавра є реалізація вебсайту для комунікації та інтерактивних активностей у реальному часі з використанням протоколу WebSocket, створеного на основі Next.js та uWebSocket.js. Він спрямований на задоволення потреб різних аудиторій, включаючи працівників корпорацій, які можуть використовувати такий сайт як засіб відпочинку під час тімбілдінґів (англ. team building – побудова команди), неформальних мітингів та інших рекреаційних подій. Однією з ключових переваг сайту є простота його використання, продуктивність, адже для доступу до кімнати активностей користувачам не потрібно реєструватися вказуючи свою електронну адресу чи номер телефону – достатньо лише ввести своє ім’я.

Завдання дослідження:

- дослідження технологій протоколу WebSocket:
  - аналіз протоколу та його відмінність від інших;
  - аналіз бібліотек та технологій для використання WebSocket;
- створення вебсайту для інтерактивних активностей, а саме:
  - моделювання структури сайту;
  - розробка ігор та інших інтерактивних активностей;
  - розгортання створеного сайту.

Об'єкт дослідження – вебсайти для спілкування та інтерактивних активностей у реальному часі.

Предмет дослідження – реалізація протоколів, засобів, методів та алгоритмів для ефективної реалізації вебсайту з інтерактивними активностями, зокрема, використання протоколу WebSocket та порівняння нинішніх JavaScript бібліотек, які його реалізовують.

Новизна роботи полягає у використанні найбільш сучасних, актуальних, та конкурентоспроможних технологій, бібліотек та протоколів.

Практична цінність роботи визначається розробленими підходами використання технологій, унікальними методами їх поєднання що дозволить використовувати такі підходи у майбутньому для полегшення розробки та більшої продуктивності.

РОЗДІЛ 1
АНАЛІЗ ІСНУЮЧИХ ПЛАТФОРМ І ПОСТАНОВКА ЗАВДАНЬ НА КВАЛІФІКАЦІЙНУ РОБОТУ

1.1 Аналіз сучасного стану проблеми

На момент написання цієї роботи уже існують схожі платформи для проведення рекреаційних активностей у реальному часі. Проаналізувавши існуючі рішення та виявивши їх переваги та недоліки, отримаємо деякі вимоги які допоможуть реалізувати ефективне та конкурентоспроможне рішення.

1.1.1 Критерії оцінки існуючих рішень

Для оцінки існуючих платформ потрібно користуватись чіткими критеріями, однаковими для усіх платформ – це дозволить отримати об’єктивні дані стану речей і порівнювати рішення між собою. Такими критеріями будуть:

1) кросплатформність. Рішення має бути доступне для якомога більшої кількості користувачів, що означає доступність на найпоширеніших платформах, а саме: смартфони, планшети, персональні комп’ютери;

2) доступність. Для нових користувачів доступ на основного наповнення платформи – до активностей, має бути якомога простішим та швидшим. Необхідність реєстрації, вказання мобільного телефону, пошти, верифікація і т.д. будуть негативними факторами для цього критерія. При цьому наявність самої можливості перелічених процедур не є недоліком сама по собі, а лише у випадну необхідності для доступу до активностей;

3) синхронічність. Події у рекреативних активностях мають відбуватись одночасно для усіх учасників активності, із якомога меншою затримкою;

4) різноманіття. Платформа має містити якомога більшу кількість доступних якісних активностей. Якісність активності включає у себе якість подання (дизайн), та якість наповнення – функціональні можливості, механіки які надає активність – вони мають бути захоплюючими, викликаючими інтерес.

Оцінка за критеріями буде складатись із перерахування переваг та недоліків. За трьох-бальної шкалою:

− 3/3, у випадку якщо переваги значні та недоліки незначні;

− 2/3, у випадку якщо переваги та недоліки у рівній кількості;

− 1/3, у випадку якщо переваги незначні та недоліки значні;

Встановивши критерії, можна перейти до аналізу існуючих рішень.

1.1.2 Discord

Discord – це сервіс голосового, текстового та відеоспілкування. У 2023-му році було представлено активності (див. рис. 1.1) [1]. На момент написання роботи існує 13 активностей. До безкоштовних, відносяться три: Watch Together (сумісний перегляд відео із відеохостингу YooTube), Gartic Phone (гра на відгадування малюнків), Chess In The Park (шахи). Проте, для отримання доступу до решти необхідно оформити платну підписку вартістю 9.99 доларів на місяць.

1) Кросплатформність – 3/3:

- переваги: сервіс доступний у веббраузері, додатку для персонального комп’ютера та у вигляді для додатка на Android та iOS;

- недоліки: немає;

2) доступність – 1/3:

- переваги: немає;

- недоліки: необхідна реєстрація із вказанням електронної пошти;

3) синхронічність – 3/3:

- переваги: сервіс швидко обробляє події, а також надає можливість спілкування у аудіо та відео чатах;

- недоліки: немає;

4) різноманіття – 2/3:

- переваги: наявність 13 різних активностей, активності мають сучасний дизайн який дотримується контексту тематики активності;

- недоліки: 76.92% активностей доступні лише при оформленні платної підписки.

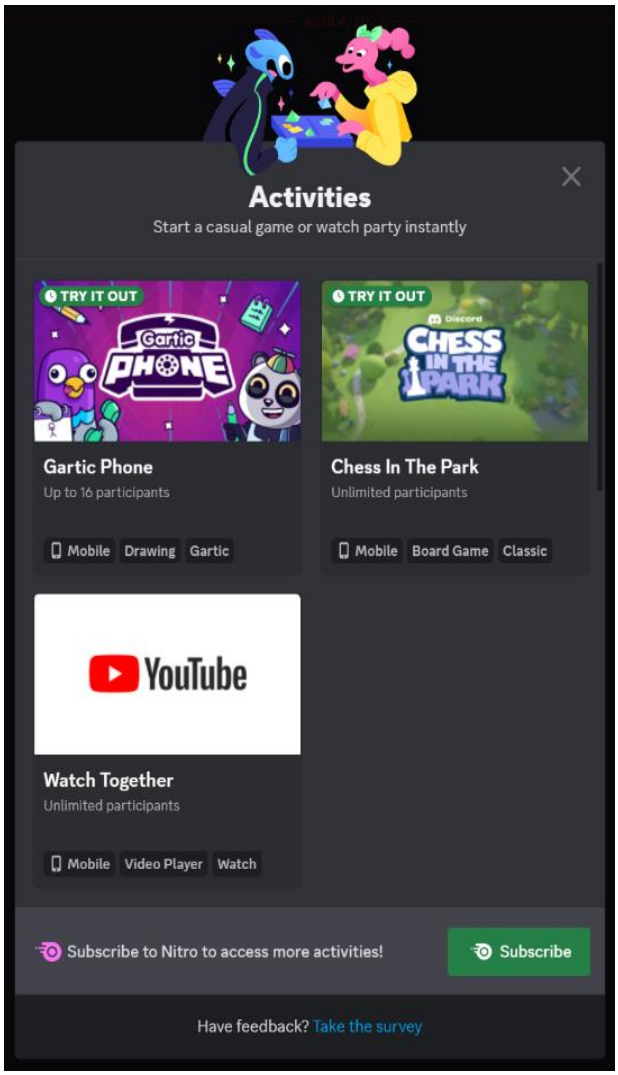

Рисунок 1.1 – Активності сервісу Discord

1.1.3 Slack

Slack – інструмент для спілкування в команді, що дозволяє створювати канали для обговорення різних тем, а також інтегрувати сторонні додатки для розширення функціональності. Slack часто використовується корпораціями для внутрішньої комунікації, а значить може використовуватись для проведень онлайн тімбілдинґів. Активності можуть бути реалізовані за допомогою сторонніх додатків – ботів (див. рис. 1.2) [2].

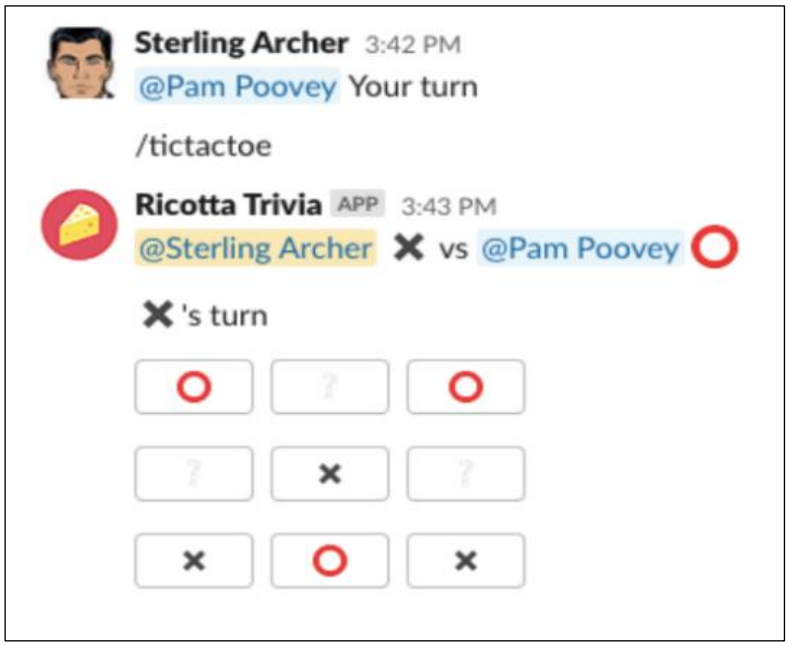

Рисунок 1.2 – Активність «Хрестики Нулики» у Slack

1) Кросплатформність – 3/3:

- переваги: сервіс доступний у браузері, додатку для персонального комп’ютера та у вигляді для додатка на Android та iOS;
- недоліки: немає;

2) доступність – 1/3:

- переваги: немає;
- недоліки: необхідна реєстрація із вказанням електронної пошти, часто акаунти у Slack належать корпорації (якщо це робочий акаунт);

3) синхронічність – 3/3:

- переваги: сервіс швидко обробляє події, а також надає можливість спілкування у аудіо та відео чатах;
- недоліки: немає;

4) різноманіття – 1/3:

- переваги: наявність великої кількості існуючих ботів, які можуть бути інтегровані до чату;
- недоліки: активності доступні лише у форматі чату (тексту), що значно занижує можливість розробки якісного дизайну активності; також, для додання таких ботів у чати необхідно мати відповідні права адміністратора чату.

1.1.4 Cryptex.Game

Вебсервіс online.cryptex.games пропонує велику кількість (понад 50) квестів для проходження командою (див. рис. 1.3) [3]. Після проходження квесту, можна порівнювати результати своєї команди із іншими.

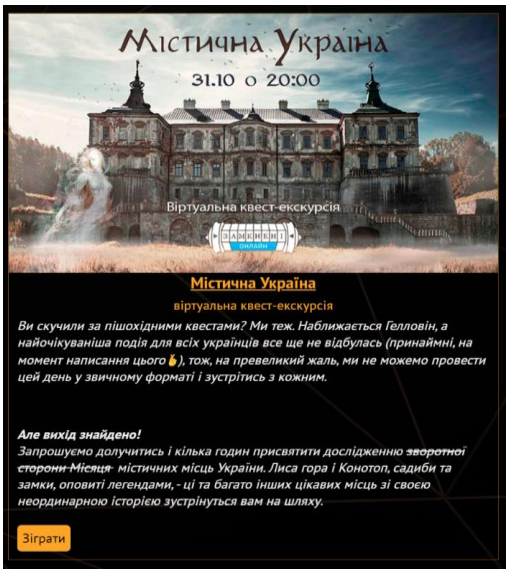

Рисунок 1.3 – Квест на Cryptex.Game

1) Кросплатформність – 2/3:

- переваги: сервіс доступний лише у браузері;
- недоліки: відсутність додатка для смартфонів;

2) доступність – 2/3:

- переваги: відсутність необхідності реєстрації учаснику команди;
- недоліки: необхідна реєстрація для створення команди;

3) синхронічність – 1/3:

- переваги: немає;
- недоліки: сервіс не відправляє повідомлення про події на сторону клієнта автоматично, для отримання нової інформації необхідно перезапускати сторінку;

4) різноманіття – 3/3:

- переваги: наявність великої кількості квестів (понад 50), навіть при наявності платних квестів, кількість безкоштовних та доступних усім – задовільна, враховуючи що один квест може займати години часу;
- недоліки: немає.

1.1.5 SIGame

Цифрова версія гри-вікторини Jeopardy, також відомої як “Своя Гра”. Jeopardy – це телевізійне шоу , яке вперше вийшло в ефір у 1964 році на телеканалі NBC в США і відтоді стало популярним в багатьох країнах світу. Головна ідея гри полягає в тому, щоб гравці відповідали на запитання з різних категорій, використовуючи свої знання з різних областей, таких як історія, наука, література, спорт і т.д. Програмна версія створена розробником Володимиром Хілем (див. рис. 1.4) [4].

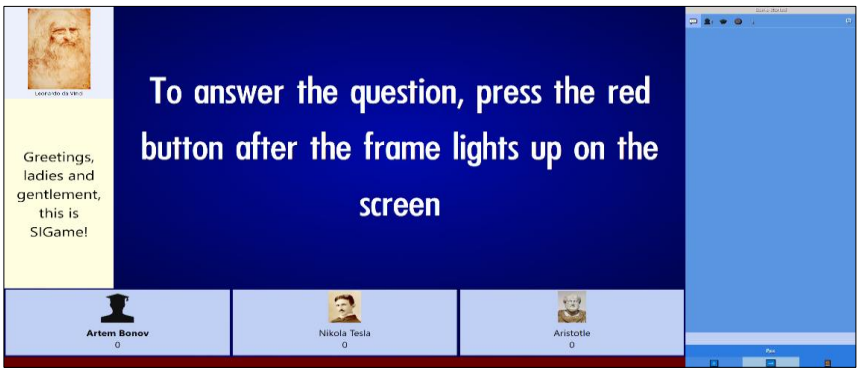

Рисунок 1.4 – Онлайн версія гри Jeopardy (SIOnline)

1) Кросплатформність – 1/3:
- переваги: наявність десктопної та базової веб версії;
- недоліки: повноцінна версія доступна лише для персонального комп’ютера на системі Windows, версія для браузера не містить значної кількості функціонала, версії для смартфонів існують неофіційно та не підтримують значну кількість функціонала;

2) доступність – 3/3:
- переваги: відсутність необхідності реєстрації;
- недоліки: немає;

3) синхронічність – 2/3:

- переваги: більшість події відбуваються одночасно для усіх гравців;
- недоліки: невелика затримка та наявність баґів заважає чесній грі у подіях зав’язаних на реакцію та погіршує досвід від користування сервісом;

4) різноманіття – 3/3:

- переваги: Існує велика кількість безкоштовних створених користувачами наборів питань та конструктор вікторини що робить потенціал різноманіття практично невичерпними;
- недоліки: немає.

1.2 Постановка завдання на кваліфікаційну роботу бакалавра

З урахуванням проблематики та аналізу переваг та недоліків існуючих рішень, розумним рішенням буде:

1) розробка вебсайту. Вебсайт надасть кросплатформність, адже він буде доступним із будь-якого гаджета який містить браузер у своїй операційній системі. Це відкине необхідність створення додатків під кожну окрему операційну систему;

2) вебсайт має швидко надавати доступ користувачам без необхідності реєстрації, реалізовуючи доступність;

3) вебсайт має реалізовувати свій функціонал так, щоб усі учасники активності бачили зміни одночасно, при цьому із якомога меншою затримкою. Саме тут нам допоможе протокол WebSocket реалізувати синхронічність;

4) вебсайт повинен містити принаймні 3 безкоштовні активності, доступні усім користувачам. Активності можуть бути наступних типів: квест, вікторина, гра та ін., таким чином реалізовуючи різноманіття;

5) розробивши вебсайт, також необхідно провести тестування та налагодження оцінити перспективи розширення та майбутній розвиток створеної платформи.

Висновки до розділу 1

У розділі були встановлені критерії для аналізу аналогічних рішень, проведений аналіз існуючих платформ із виявленням їхніх переваг та недоліків. Виходячи з оцінки предметної області будо поставлення завдання до кваліфікаційної роботи бакалавра.

РОЗДІЛ 2
СПЕЦИФІКАЦІЯ ВИМОГ ДО РОЗРОБЛЮВАНОЇ СИСТЕМИ

2.1 Аналіз, визначення вимог до розроблюваного програмного забезпечення та проєктування програмного забезпечення

Вимоги до проєкту поділено на різні типи описуючі різні аспекти програмного забезпечення та досвіду його використання.

2.1.1 Базові технічні вимоги.

До базових технічних вимог відносимо:

– наявність коду сайту розміщеного на сервісі контролю версій (GitHub);

– наявність публічної HTTP адреси яка надаватиме доступ до сайту;

– наявність розгорнутого вебсервера який буде відповідати на запити.

2.1.2 Функціональні вимоги.

До функціональних вимог відносимо:

– можливість швидкого доступу до сервісу без вказання електронної адреси чи номеру телефона – простий доступ за псевдонімом (англ. nickname);

– можливість завантаження користувачати зображення профілю;

– можливість зміни акцентного кольору користувача;

– наявність глобального чату, де користувачі можуть ділитись враженнями та порадами щодо користування сайтом;

– публічний моніторинг онлайн користувачів;

– можливість створення кімнат (англ. lobby). Кімнати матимуть наступний функціонал:

1) вибір активності при створенні кімнати;

2) можливість встановлення пароля при створенні кімнати;

3) наявність вибору із трьох та більше рекреаційних активностей при створенні кімнат;

– наявність локального чату, доступного лише учасникам активності;

– можливість проведення перевірки готовності перед активністю (англ. ready check);

– функціонуюча активність;

– вимоги щодо мобільної сумісності (англ. mobile compatibility). Усі функції сайту повинні бути доступними із мобільних, сенсорних пристроїв (смартфони, планшети);

– наявність можливості контролювання гучності звуків на сайті компонентами самого ж сайту (без змінення системних налаштувань звуку).

2.1.3 Нефункціональні вимоги.

До нефункціональних вимог відносимо:

– широка обробка помилок. Часто у вебзастосунках відбуваються помилки, такі як timeout error, connection error, internal error і так далі. Для користувача такі помилки не повинні бути критичними (такими що унеможливлюють роботу сайту);

– швидка відповідь. При надсиланні запитів до сервера, час відповіді не повинен становити понад 10 секунд при нормальних умовах;

– базова SEO оптимізація:

1) сайт повинен бути доступним для індексації пошуковими системами, що означає наявність SSR (server side rendering);

2) сайт повинен містити meta теги що покращують шанси сайти бути знайденими пошуковими системами;

– сайт повинен мати консистентний дизайн (зовнішній вигляд), мати послідовний CSS що надавало б очікуваний, інтуїтивний вигляд для користувача. Включає у себе оптимізацію дизайну під мобільні пристрої;

– ресурси які можуть бути завантаженні асинхронно, мають бути завантаженні асинхронно. А саме: звуки, стилі та компоненти і т.д.. Приклад: завантажувати файли щодо конкретної активності необхідно лише тоді, коли ця активність використовується. Такий підхід зменшить обсяг необхідних даних для першого завантаження, а значить збільшить швидкість завантаження сайту та використання інтернет-трафіку.

2.1.4 Прототип сайту

Користуючись засобом прототипування Figma, були створені прототипи основних сторінок та компонентів, їхню взаємодію. Ці прототипи не охоплюють дизайн сторінок, а лише показують необхідність тих цих інших компонентів на сторінці, сама візуальна частина готового розробленого продукту може відрізнятись. Також варто зауважити що прототипи можуть охоплювати більший функціонал ніж буде міститись у готовому продукті, адже саме із прототипування починається додання нових функції, сторінок і т.д. які ще не доступні для публічного релізу.

Сторінка Входу (Login page) містить три стани: пуста, заповнена та заповнена із помилкою. Після успішного входу, користувач має бути направлений до домашньої сторінки. Сторінка Домашня (Home page) (див. рис. 2.1) містить компоненти: огляд кімнат, огляд користувачів, глобальний чат, редактор профілю, створювач кімнат, вхід у кімнати.

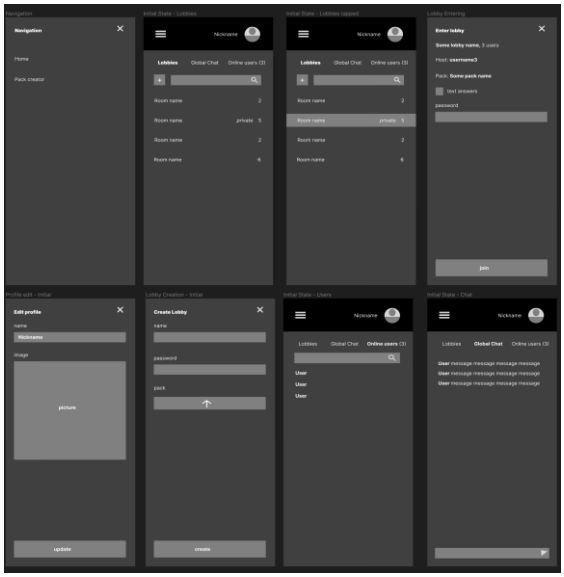

Рисунок 2.1 – Прототипи Домашньої сторінки

Сторінки інтерактивних активностей. Детально пропрацьовано було лише прототип для активності Jeopardy, оскільки решта активностей не мають складної логіки вартої прототипування.

Сторінка активності Jeopardy (див. рис. 2.2). Містить стани: завантаження ресурсів, показ теми, показ питань, вибір питання, показ питання, фінал.

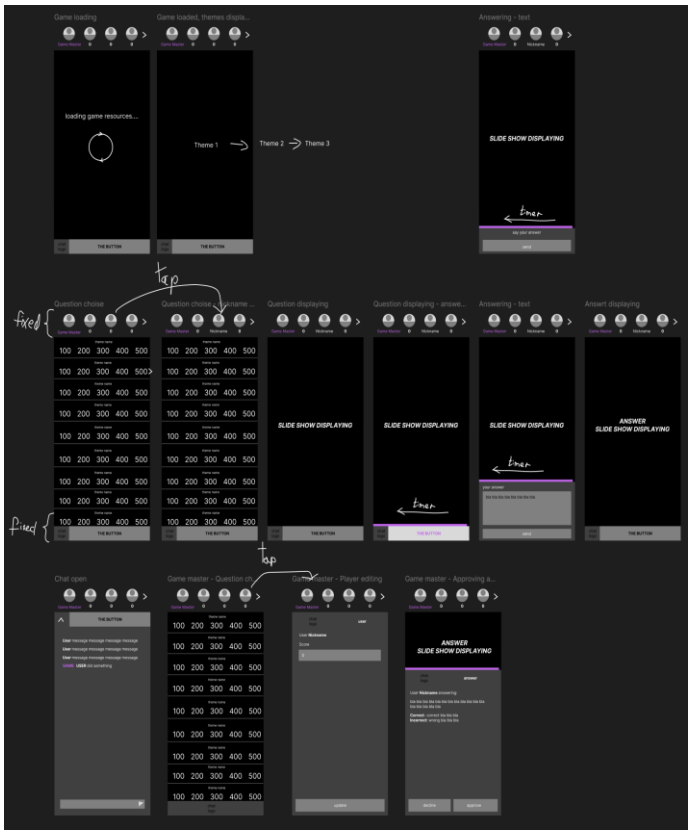

Рисунок 2.2 – Прототипи активності Jeopardy

2.2 Вибір засобів, методів і алгоритмів вирішення поставленого завдання

Для розробки вебсайту був обраний наступний стек технологій: Nginx, Node.js, Next.js, React, uWebSockets.js, MUI. Вибір кожної із цих технологій має свою причину у контексті поставленої задачі. Також у контексті вибору методів для вирішення поставленого завдання був проведений аналіз конкурентних рішень для використання протоколу WebSocket на платформі Node.js.

2.2.1 Nginx – проксі сервер

При розгортанні вебсайту, для того, щоб користувач отримав доступ до нього має функціонувати основний проксі сервер який буде розподіляти запити щодо встановлених правил. Таким чином можна буде працювати із декількома

внутрішніми, локальними серверами, при цьому для користувача буде видним лише один, публічний сервер. Також в обов’язки Ngix входять функції обробки статичних файлів, кешування, балансування навантаження та обробки запитів, що дозволяє забезпечити швидкий та стабільний доступ до вебсайту [5].

2.2.2 JavaScript та платформа Node.js

Написання проєкту на JavaScript як мовою високого рівня має значні переваги. У своїй простоті та поєднанню функціонального та об'єктноорієнтованого програмування для розробника відкриваються широкі можливості та великий обсяг рішень якими можна вирішити одну і ту ж саму проблему. JavaScript є однією із найпопулярніших мов для написання серверної частини вебдодатка. TypeScript розширює можливості мови надаючи строгу типізацію. Для того, щоб мати можливість виконувати код написаний цією мовою, необхідна платформа написана більш низькорівневою мовою (наприклад на C++). Такими є: Node.js, Rhino, JavaScriptCore та інші.

Платформа Node.js побудована на високопродуктивному двигуні V8 – він же використовується у браузерах на базі Chromium. Платформа була обрана через високу продуктивність, кросплатформеність та багату документацію [6].

2.2.3 Next.js та React

Бібліотека React була обрана для спрощення написання складної логіки на стороні клієнта, у тому числі за допомогою зручного синтаксису JSX. Розділяючи логіку на окремі компоненти отримується більше файлів із меншою кількістю коду у кожному із них – модульний підхід. Таким чином спрощується налагодження та тестування компонентів.

Важливим фактором при розробці проєкту, є швидкість налаштування оточення та написання шаблонного коду (англ. boilerplate). Для спрощення цього етапу розробки (у тому числі) був обраний фреймворк Next.js – він містить логіку як для серверної частини, так і для клієнтської. Аналогічні рішення могли б бути налаштовані вручну за допомогою поєднання Express та React, проте Next.js надає додаткові можливості які прості у використанні [7]:

− SSR – англ. server side rendering – сторінка будується на стороні сервера, що покращує досвід користування сайтом та SEO оптимізацію;

− статичне генерування: можна використовувати статичне генерування для попереднього рендерингу сторінок під час збірки;

− маршрутизація на стороні сервера;

− Hot reloading (англ. гаряче перезавантаження) – режим розробки Next дозволяє автоматично оновлювати змінений код без необхідності перезапуску сервера або перезавантаження сторінки;

− розширені можливості: Next.js надає розширені можливості для розробки вебдодатків, включаючи підтримку CSS-модулів, аналітики, розробки API-ендпоінтів, підтримки TypeScript, інтеграцію з зовнішніми бібліотеками та інше.

2.2.4 MUI

Для побудови послідовно, інтуїтивного, оптимізованого інтерфейсу була обрана бібліотека MUI (Material UI) – вона пропонує повний набір інструментів інтерфейсу користувача [8]. Готові компоненти протестовані на високий рівень доступності (a11y – англ. accessibility – доступність для користувачів із вадами зору та іншими обмеженнями). Також бібліотека має широку документацію щодо кожного компонента. Саме тому вона була обрана для поточного проєкту, і використовуючи можливості бібліотеки були створені нові компоненти для усіх потреб користувачів.

2.2.5 Аналіз технологій WebSocket

Як зазначено у статті “The WebSocket Protocol” [9] який став міжнародним стандартом для цього протоколу (RFC 6455), WebSocket – протокол, який забезпечує двосторонній зв'язок між клієнтом, що виконує ненадійний код у контрольованому середовищі, та віддаленим гостом (англ. host), який погодився на зв'язок з цим кодом. Такий підхід дозволяє вебдодаткам отримувати дані в режимі реального часу без необхідності постійного опитування сервера щодо нових даних (англ. polling).

Особливості що відрізняють WebSocket від інших протоколів:

− постійне з'єднання: WebSocket забезпечує постійне з'єднання між клієнтом і сервером. Це означає, що з'єднання встановлюється один раз, і даними можна обмінюватися в обидва напрямки без необхідності створювати нове з'єднання для кожного запиту;

− бідирекціональний обмін даними: WebSocket дозволяє обмінюватися даними між сервером і клієнтом у режимі реального часу в обох напрямках. Це дозволяє серверу надсилати повідомлення клієнту та отримувати повідомлення від клієнта без необхідності виконувати нові HTTP-запити;

− низька накладна вартість (англ. low overhead): означає, що використання протоколу WebSocket вимагає меншої кількості ресурсів (таких як пам'ять, процесорний час, мережевий трафік) у порівнянні з іншими протоколами, наприклад HTTP. Оскільки з'єднання залишається відкритим протягом тривалості сеансу, нема потреби витрачати ресурси на встановлення нових з'єднань для кожного запиту;

− підтримка реального часу: WebSocket дозволяє реалізувати функціональність в режимі реального часу, таку як миттєве оновлення сторінки або отримання сповіщень без затримок. Це особливо важливо для додатків, які потребують актуальних даних і миттєвої відповіді;

− широке застосування: Протокол WebSocket широко використовується в різних сферах, включаючи вебдодатки, графічні ігри, спілкування в реальному часі, фінансові системи та багато іншого. Він є стандартним протоколом для реалізації вебсокетів і має підтримку в більшості сучасних браузерів та серверних мов програмування.

Платформа Node.js немає реалізації протоколу WebSocket у своїй стандартній бібліотеці, при, тому що має реалізацію протоколу HTTP. Отже, для того, щоб побудувати повноцінний додаток на цій платформі доведеться або реалізовувати функціонал протоколу WebSocket самому, або підключати сторонні бібліотеки.

Одними із таких найбільш поширених бібліотек є: ws, Socket.IO, Fastify, uWebSockets.js та інші. Усі ці реалізації мають свої відмінності та нюанси.

2.2.5.1 Бібліотека ws

Бібліотека ws (ліцензія MIT) написана на мові JavaScript, із можливістю встановлення додаткових модулів на C++ для оптимізації (bufferutil та utf-8-validate). Бібліотека підтримує стиснення за принципом permessage-deflate, яке можна вмикати і налаштовувати за допомогою різних опцій [10].

2.2.5.2 Бібліотека Socket.IO

Бібліотека Socket.IO (ліцензія MIT) написана на TypeScript для Node.js полегшує двонапрямний зв'язок на основі подій у реальному часі між сервером Node.js і клієнтською стороною. Пакет має реалізацію як для сервера, так і для клієнтської сторони (кросбраузерно), замість вбудованого у браузер модуля. При цьому важливо розуміти що Socket.IO не є реалізацією WebSocket сама по собі, але використовує WebSocket як транспорт, коли це можливо, додаючи деякі метадані до кожного пакету, включаючи тип пакета, простір імен та ідентифікатор ack. Надає широкий готовий функціонал для кімнат та подій [11].

2.2.5.3 Бібліотека Fastity

Fastify (ліцензія MIT) – це повноцінний вебфреймворк, вважається одним з найшвидших доступних вебфреймворків про що і зазначає у своїй документації [12]. Fastify має потужну архітектуру плагінів – у яку входять і рішення для WebSocket. Бенчмарки розробниками бібліотеки показали (вихідний код тестів доступний у ґітгаб репозиторії fastify/benchmarks), що Fastify (версії 4.0.0) може обробляти до 77 193 запитів на секунду, що робить його одним з найшвидших доступних вебфреймворків. При цьому, вбудований у Node.js модуль http.Server (версії 16.14.2) показує результат 74 513 запитів на секунду, надаючи значно менший функціонал

2.2.5.4 Бібліотека uWebSockets

uWebSockets (ліцензія Apache 2.0) – це легка, високопродуктивна бібліотека для реалізації протоколу WebSocket для сервера (для платформи Node.js – uWebSockets.js), написана повністю на C++ [13]. Вона надає простий API для створення додатків реального часу, що використовують протокол

WebSocket, а також HTTP/HTTPS і події, що надсилаються сервером (SSE – англ. server sent events).

Однією з ключових особливостей uWebSockets.js є швидкість та ефективність, що досягається завдяки використанню керованими подіями, неблокуючої архітектури та низькорівневої реалізації на C++. Бібліотека підтримує шифрування TLS/SSL і стиснення permessage-deflate що є частиною стандарту RFC 6455. Для клієнтської сторони може використовуватись будь-яка реалізація, у тому числі і вбудовані модулі у браузері. Також варто зазначити, що, починаючи із версії 4.4.0 клієнтська та серверна частини Socket.IO почали підтримувати серверну версію uWebSockets.js про що заявив супроводжувач Socket.IO [11].

2.2.5.5 Порівняння та обґрунтування вибору бібліотеки

Бенчмарки проведені розробником Алексом Галтманом показали, що використання бібліотеки uWebSockets.js може значно збільшити продуктивність додатка на Node.js. Експеримент включав порівняльний аналіз продуктивності пристрою Raspberry Pi 4 [14], який міг обслуговувати 100 000 HTTP-запитів на секунду через з'єднання Ethernet (без вентилятора для пристрою). Встановивши Ubuntu Server 20.10 і створивши інструменти збірки uWebSockets C++, створили тестовий додаток на пристрої для обробки HTTP-запитів і перевірили його продуктивність за допомогою тесту на навантаження HTTP – 200 підключень із виконанням не конвеєрних запитів HTTP так швидко, як це дозволяє сервер (вихідний код доступний у гітгаб репозиторії uNetworking/uSockets – http_load_test). Результати показали, що Raspberry Pi 4 може обслуговувати 106 000 маршрутизованих HTTP-запитів за секунду при розгоні до 1,7 ГГц. Для порівняння, запуск того самого тесту з Node.js / Fastify як кластер дав лише 8800 запитів на секунду, що означає, що рішення C++ було в 12 разів швидшим. Навіть якщо ввімкнено шифрування TLS 1.3, µWebSockets перевершив Node.js/Fastify у 8,75 разів. Однак, коли той самий тест проводився з використанням uWebSockets.js для Node.js, результати показали стабільні 75 тис. запитів/с, що все одно було у 8,5 разів швидше, ніж у Node.js/Fastify, і

близько до 75% продуктивності самого uWebSockets. З експерименту можна дійти висновку, що різні програмні рішення можуть мати значний вплив на продуктивність, і що рідні надбудови C++ можуть значно підвищити продуктивність Node.js у багатьох випадках.

Аналогічні бенчмарки, проведені із порівнянням з Socket.IO [15], показують, що uWebSockets може обробляти 100 тис. захищених з'єднань без проблем і зі стабільною швидкістю 50 тис. захищених повідомлень за секунду, тоді як Socket.IO виходить з ладу при 40 тис. захищених з'єднань і ледве справляється належним чином з 10 тис. з'єднань.

Висновок аналізу бібліотек полягає в тому, що у випадку якщо продуктивність, швидкість є критично важливою то доцільно обирати бібліотеку uWebSockets.js. У іншому ж випадку, якщо необхідний ширший функціонал, а швидкість не грає важливої ролі – є сенс використовувати Socket.IO чи фреймворки по типу Fastify із додатковими модулями.

Оскільки для розробленого вебсайту для інтерактивних активностей необхідна висока продуктивність, було обрано саме бібліотеку uWebSockets.js

Висновки до розділу 2

Був проведений аналіз технологій та обґрунтування вибору фреймворків, бібліотек у контексті розробки сайту для рекреативних активностей. Був проведений аналіз технологій на базі протоколу WebSocket та відмінності від інших протоколів, досліджені різні бібліотеки для платформи Node.js які забезпечують роботу у рамках даного протоколу, обрана технологія для використання при розробці сайту для інтерактивних активностей.

РОЗДІЛ 3
РОЗРОБКА ВЕБСАЙТУ ДЛЯ ІНТЕРАКТИВНИХ АКТИВНОСТЕЙ У РЕАЛЬНОМУ ЧАСІ

3.1 Практична реалізація об’єкта проектування

Провівши аналіз сучасних актуальних вебтехнологій та обґрунтувавши вибір стека, можна переходити до поєднання цих технологій у відповідному проєкті. Перед тим як перейти до розробки самих інтерактивних активностей необхідно вибудувати архітектуру на який ці активності будуть працювати. Ця архітектура включатиме декілька рівнів абстракції, що дозволить коду самих активностей буде високорівневим та інтуїтивним у написанні та читанні, а також надасть можливості для подальшого розширення.

3.1.1 Архітектура вебсерверів

Бібліотека uWebSocket.js надає можливість створення окремого локального сервера який відповідає за запити протоколу WebSocket (також є можливість для HTTP, проте цей функціонал не використовується у контексті розробленого проєкту). Цей вебсервер матиме свій окремий порт, відмінний від основного локального сервера додатка на Next.js. Для того що зі сторони клієнта було просто приєднатися до цих серверів не вказуючи порт, використовується маршрутизація за допомогою налаштування правил для публічного сервера на Nginx (Reverse Proxy).

Сайт містить функціонал який включає завантаження ресурсів користувачами. Ці ресурси будуть групуватись у окрему директорію на сервері, а тому, щоб їх можливо було отримати на стороні клієнта – знову використовується маршрутизація на Nginx. Схема взаємодії серверів при реагуванні на запит зображена на рисунку 3.1

.

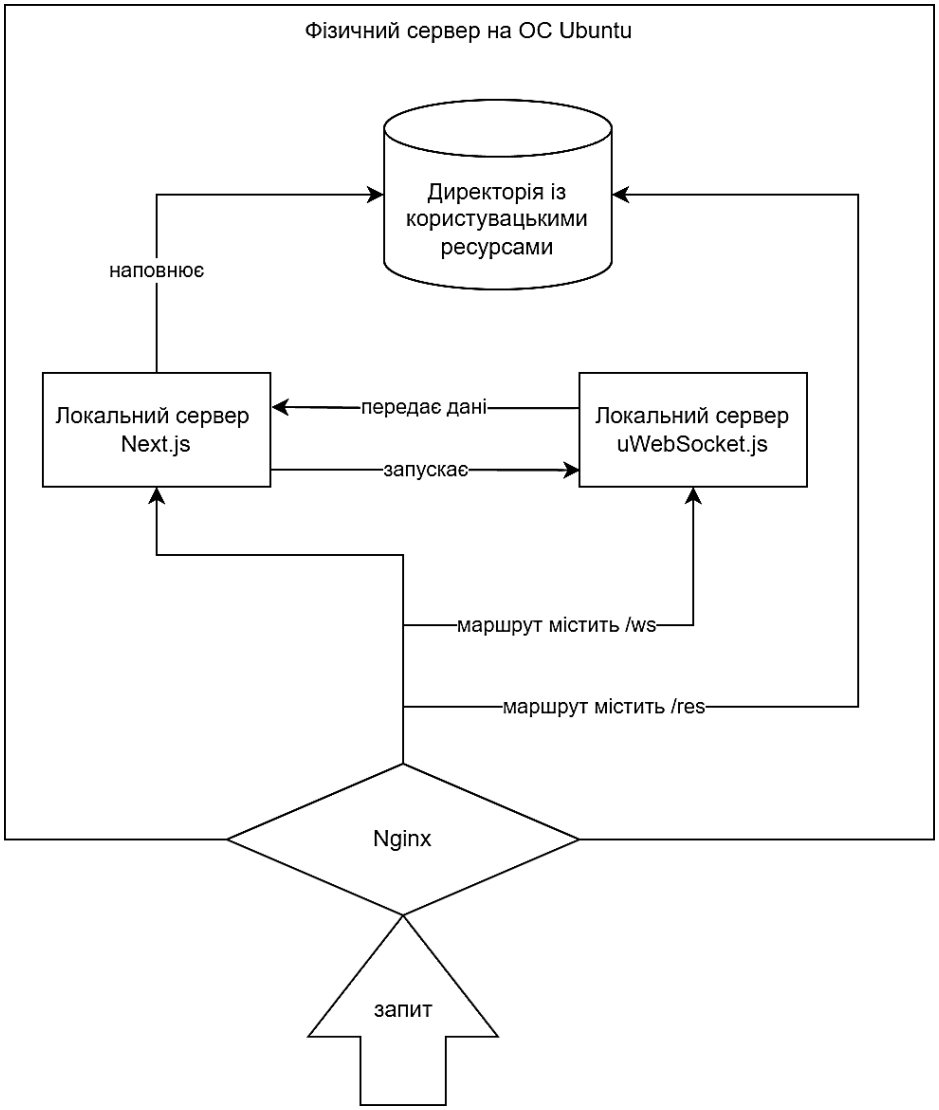

Рисунок 3.1 – Схема архітектури серверів

Для того щоб така архітектура працювала коректно та надавала користувачу найкращий досвід користування сайтом, необхідно щоб обидва локальні сервери були розгорнуті із використанням SSL. Таким чином, для підключення та комунікації сайт використовуватиме протоколи HTTPS та WSS (WebSocket SSL) [16]. Інформація надіслана цими протоколами буде зашифрована криптографічними методами – таке шифрування буде найбільш корисне при під’єднанні із публічних мереж (кафе, транспортні зупинки і т.д.).

Для отримання такого сертифікату був використаний сервіс Let's Encrypt, який надає можливість безкоштовного отримання таких сертифікатів – єдина умова це необхідність налаштування автоматичного перестворення

сертифікату, адже вони не довговічні та надаються на 90 днів. Автоматизувати цей процес на ОС Ubuntu можна завдяки використанню утиліти certbot [17].

Після завершення налаштувань Nginx щодо Reverse Proxy [18] та SSL конфігураційний файл має виглядати аналогічним до Лістингу 3.1. Повний код лістингу наведено у додатку А.

Лістинг 3.1 – Конфігурація Nginx (Reverse Proxy та SSL)

```
client_max_body_size 100M;
server {
        root /var/www/game-club.click/html;
        index index.html index.htm index.nginx-debian.html;
        server_name game-club.click www.game-club.click;
        location / {
}
```

Кінець Лістингу 3.1

3.1.2 Абстрагування над uWebSocket.js

Бібліотека uWebSocket.js надає базовий, відносно низькорівневий функціонал для серверної частини для реалізації протоколу WebSocket. Екземпляр об’єкту WebSocket має наступні методи що найчастіше використовуються:

- send – надіслати повідомлення до клієнта (рядок);
- end – надіслати повідомлення про закриття сокета та закрити його;
- close – закрити вебсокет без попередження;
- ping – надіслати контрольне повідомлення типу ping;
- subscribe – підписати клієнта на деяку тему;
- unsubscribe – відписати клієнта від деякої теми;
- isSubscribed – перевірити чи клієнт підписаний на деяку тему;
- getTopics – отримати список тем на які підписаний клієнт;
- publish – опублікувати повідомлення у деякій темі.

Ці базові методи оперують простими рядками, які можуть представляти будь-які дані, у будь якому форматі. Для того щоб мати чітку структуру обміну

інформацією було обрано формат JSON для обміну повідомленнями – цей формат є нативним для JavaScript а робота із ним є найбільш швидкою, порівнюючи із іншими форматами такими як XML чи YAML.

Наступним шляхом є визначення структури, якої може набувати надіслане повідомлення. Будь-яке буде містити три поля:

− token: string – унікальний токен, за допомогою якого можна отримати інформацію про клієнта;

− ctx: string – контекст у якому було надіслане повідомлення. Маючи лише одне підключення на користувача, різні компоненти сторінки можуть використовувати це єдине підключення для обміну даними у своєму контексті;

− data: any – безпосередні дані які потрібні для передачі у конкретному контексті.

Використовуючи TypeScript, тип поля data було асоційовано із відповідним контекстом яке передається у полі ctx. Таким чином набагато складніше допустити помилки при написанні коду які приведуть до помилок при виконанні коду (англ. runtime error), адже ці помилки будуть одразу показані при компіляції коду.

Для такої асоціації типів необхідно притримуватись наступної структури при написанні логіки для кожного із контекстів. Структура наведена у Лістингу 3.2.

Лістинг 3.2 – Приклад обробника зі сторони сервера для конкретного контексту

```
import { Handler } from 'uWebSockets/uws.types'
export type Request = {
    // Перелік вхідних даних запиту
}
export type Success = {
    success: true
}
export type Failure = {
    success: false
}
export const handler: Handler<Request, Success | Failure> = (act, state, data) => {
    // Логіка обробки повідомлення
```

Продовження лістингу 3.2

```
act.res({
        success: true
    })
}
```

Кінець Лістингу 3.2

Необхідно визначити вхідні дані (тип Request), та можливі типи результату обробки повідомлення (типи Success, Failure) – усі типи потрібно декларувати використовуючи ключове слово export. Ці типи будуть імпортовані у частині коді із визначенням функцій для надсилання повідомлень зі сторони клієнта. Реалізовується за допомогою узагальнених типів (англ. generic type) де вхідним параметром є назва контексту, а отримуємо ми тип для обробники конкретно цього контексту. Виглядає це наступним чином:

Лістинг 3.3 – Декларація типів для функції надсилання даних зі сторони клієнта

```
type RequestData<R extends HandlerName> = Parameters<typeof import('uWebSockets/ws')['handlers'][R]>[2]
type HandlerSend = <H extends HandlerName>(context: H, data?: RequestData<H>) => void
type RequestHandlerRegistrar = <C extends HandlerName>(context: C, handler: RequestHandler<C>) => void
```

Кінець Лістингу 3.3

У Лістингу 3.3 використовується тип HandlerName. Він представляє собою назву контексту для обробника. Цією назвою є ім’я файлу у якому задекларовано обробник. Далі, імпортуючи усі обробники із певної директорії можна отримати тип який представлятиме собою назву обробника – назву контексту, при цьому це буде не звичайний рядок (тип string), а об’єднуючий тип (union type) який включатиме у себе усі конкретні назви цих контекстів. Для того, щоб автоматизувати імпорт обробників із директорії, було використано розширення Generate Index для IDE VSCode.

Лістинг 3.4 Декларація типу HandlerName

```
// @index('./*.ts', f => `import { handler as ${f.name.replaceAll('-', '')} } from '${f.path}'`)
import { handler as AuthLogin } from './Auth-Login'
import { handler as ChatSend } from './Chat-Send'
/* і так далі ... */
// @endindex

export const handlers = {
    // @index('./*.ts', f => `'${f.name}': ${f.name.replaceAll('-', '')},`)
    'Auth-Login': AuthLogin,
    'Chat-Send': ChatSend,
    /* і так далі ... */
    // @endindex
}

export type HandlerName = keyof typeof handlers
```

Кінець Лістингу 3.4

Використовуючи ці типи було написано методи для надсилання та обробки повідомлень зі сторони клієнта. Міститься компонент який створюються за допомогою ReactContext і надає доступ для будь-якого вкладеного компонента використовувати хуки та методи для обробки на надсилання повідомлень для будь-якого сокетного контексту гарантуючи при цьому безпеку для типів та IntelliSense для розробника. Частина коду наведена у Лістингу 3.5 (повний код у Додатку Б).

Лістинг 3.5 – Декларація React-хуків для обробки повідомлень

```
export const useWS = () => {
    return useContext(WSContext)
}
export const useRequestHandler: RequestHandlerRegistrar = (context, handler) => {
    const { on, unsubscribe } = useWS()

    useEffect(() => {
        on(context, handler)
        return () => {
            unsubscribe(context, handler)
        }
    }, [])
}
```

Кінець лістингу 3.5

Використовуючи вище описані підходи були побудовані усі компоненти вебсайту. Використання створеного React-контексту на прикладі компоненту для показу усіх поточних користувачів наведено у Лістингу 3.6.

Лістинг 3.6 – Використання React-контексту у компоненті

```
export const GlobalUsersList: React.FC = () => {
    const ws = useWS()
    const [users, setUsers] = useState<{ userNickname: string; id: string }[]>([])
    useRequestHandler('Users-Get', data => {
        setUsers(data.data)
    })
    useEventHandler('UserRegistry-OnlineUpdate', data => {
        setUsers(data.list)
    })
    const onIsConnected = () => {
        ws.send('Universal-Subscription', {
            mode: 'subscribe',
            topic: 'UserRegistry-OnlineUpdate',
            scope: 'global'
        })
        ws.send('Users-Get', {
            scope: 'global'
        })
    }
    useEffect(() => {
        return () => {
            ws.send('Universal-Subscription', {
                mode: 'unsubscribe',
                scope: 'global',
                topic: 'UserRegistry-OnlineUpdate'
            })
        }
    }, [])
    useEffect(() => {
        if (ws.isConnected) {
            onIsConnected()
        }
    }, [ws.isConnected])
    return <UsersListBox users={users} />
}
```

Кінець Лістингу 3.6

Компонент GlobalUsersList при монтуванні одразу підписується на оновлення списку користувачів за допомогою надсилання повідомлення із

контекстом Universal-Subscription передаючи у даних режим subscribe та інші дані, а при демонтуванні надсилає повідомленні у тому ж самому контексті у режимі unsubscribe. Поле scope у значенні global означає що це підписка на глобальну зміну даних, не прив'язану до певної кімнати. Поле topic позначає саму тематику на яку підписуються. Також у коді видно два типи обробників подій: useRequestHandler та useEventHandler. Їхня відмінність полягає у тому що перший відповідає за обробку відповідей на запити які надсилає сам клієнт, а другий за події які відбуваються на сервері – саме на таку тематику компонент підписується при монтуванні.

Таким чином було створено методи для: надсилання повідомлень зі сторони клієнта у конкретно визначеному форматі; обробки повідомлень відповідно до контексту; реакції на результат обробки зі сторони клієнта. Усі функції та методи гарантують безпеку типів.

3.1.3 Організація даних та взаємодія через сокети

Реалізація інтерактивних активностей передбачає, що будуть існувати деякі дані, які будуть спільними для декількох користувачів – учасників даної активності. Єдиним джерелом істини для усіх учасників має бути сервер – сервер повинен валідувати кожну дію яку здійснює учасник активності, і у випадку коли все проходить успішно і ця дія відповідає усім правилам роботи цієї активності – оновлюється стан і надсилається оновлення для усіх клієнтів.

Реалізація передбачає що на стороні сервера (основного сервера на Next.js) існуватиме об'єкт як буде представляти собою стан сайту, зберігатиме усі дані і буде виступати у ролі високоінтерактивної бази даних, напряму під'єднанню до сервера на uWebSockets.js щоразу надсилаючи дані необхідним користувачам про зміни.

Типовий об'єкт для такого представлення стану вебсайту виглядатиме наступним чином (Лістинг 3.7):

Лістинг 3.7 – Приклад класу сутності стану вебсайту

```
export abstract class StateEntity {
```

Продовження лістингу 3.7

```
    readonly state: {
        readonly field1: string
        readonly field2: string // І так далі
    }
    lobby: Lobby; // Прив'язка до деякого контексту
    constructor(lobby: Lobby) { // Необхідна ініціалізація
        this.lobby = lobby;
          this.emiter = new EventEmitter();
        this.state = {
            field1: 'default value 1',
            field2: 'default value 2'
        }
    }
    update(newData: Partial<StateEntity['state']>) {
        Object.assign(this.state, newData)
        this.lobby.publishUpdate(this)
        this.emitter.emit('update', newData)
    }
    onUpdate(callback: onUpdateCb) {
        this.emitter.on('update', callback)
    }
    data(viewer: any) {
        return {
            ...this.state,
            // Додаткова інформація
        }
    }
}
export type StateEntityData = ReturnType<StateEntity['data']>
```

Кінець Лістингу 3.7

Така декларація класу сутності складається із:

– поле state – об'єкт, безпосередні дані які представляє дана сутність;

– поле lobby (може відрізнятись залежно від сутності) – об'єкт який презентує контекст у якому ця сутність існує, контекст у якому вона може оновлюватись. Містить метод для публікації змін для клієнтів у цьому контексті;

– метод update – приймає нові дані – деякі змінені поля стану об'єкта, застосовує зміни та публікує їх;

– метод onUpdate – надає можливість для сторонніх сутностей підписуватись на оновлення поточної сутності;

– метод data – повертає об’єкт який можна конвертувати у формат JSON та відправити через сокетне з’єднання, повертає дані про поточний стан сутності. Приймає дані про глядача, який має отримати дані – таким чином можна видозмінювати, фільтрувати поля у залежності від ролі глядача;

– тип StateEntityData – є результатом виконання методу data, користуючись цим типом можна на клієнтській стороні бути впевненим у даних які отримуються зі сторони сервера що забезпечує безпеку типів.

Прикладом такої сутності є User (Лістинг 3.8). Окрім раніше перерахованих полів, такий клас містить додаткові поля та методи необхідні для роботи цього класу.

Лістинг 3.8 – Клас User

```
export class User {

    static readonly AutoLogoutMs = 5000
    private readonly emitter: EventEmitter
    private logoutTimeout: NodeJS.Timeout
    private lobbies: Lobby[] = []
    ws: WebSocket
    readonly token: string
    readonly id: string
    readonly state: { /* поля */ }
    constructor(id: string, ws: WebSocket) { /* ініціалізація полів */
        this.logoutTimeout = setTimeout(() => {
            this.update({ userIsOnline: false })
        }, User.AutoLogoutMs)
    }
    refreshOnlineChecker() {
        clearTimeout(this.logoutTimeout)
        if (!this.state.userIsOnline) {
            this.update({ userIsOnline: true })
        }
        this.logoutTimeout = setTimeout(() => {
            this.update({ userIsOnline: false })
        }, User.AutoLogoutMs)
    }
    linkLobby(lobby: Lobby) {
        this.lobbies.push(lobby)
    }
```

Продовження лістингу 3.8

```
unlinkLobby(lobby: Lobby) {
        this.lobbies = this.lobbies.filter(l => l !== lobby)
    }
    get hasLobbies() {
        return this.lobbies.length > 0
    }
    get lobby() {
        return this.lobbies[0]
    }
    data()
}
```

Кінець Лістингу 3.8

Клас User містить додаткову логіку, нетипову для звичайної сутності. Основним є прив'язка до кімнат у яких знаходиться користувач, та оновлення маркеру isOnline. Встановлюється таймер на 5 секунд (статичне поле AutoLogoutMs) який має встановити поле isOnline у значення false. Проте, якщо користувач дійсно знаходиться на вебсайті він кожну секунду відправлятиме повідомлення типу ping та скидатиме цей таймер, встановить isOnline у значення true та знову запустить таймер. Таким чином це поле завжди буде актуальну інформацію щодо онлайн статусу користувача. Додаткові методи та ґеттери для кімнат, спрощують реалізацію логіки взаємодії користувача і відповідних кімнат. Поле ws містить екземпляр об'єкту підключення WebSocket та надає можливість надсилати повідомлення саме цьому клієнту.

Кожна сутність містить додаткові методи та поля необхідні саме для неї, роблячи взаємодію із сутністю інтуїтивною, легкою та продуктивною – водночас кожна сутність побудована за раніше встановленими принципами показаними на прикладі класу StateEntity.

Основними класами створеними для роботи сервісу є:

– AppState – головний клас, містить усі дані і є єдиним джерелом істини;

– User – клас користуча. Представляє загальні дані користувача;

– UserRegistry – клас який відповідає за множину усіх користувачів на сайті. Усі нові користувачі мають бути додаватись та видалятись за допомогою цього класу;

− Lobby – клас кімнати до якої підключаються користувачі для інтерактивних активностей;

− LobbiesRegistry – клас який відповідає за множину усіх кімнат на сайті, взаємодію із ними – створення та видалення;

− LobbyMember – клас який представляє користувача у конкретній кімнаті. Містить посилання на клас User;

− Chat – клас який презентує чат. Може бути локальним (для конкретної кімнати – містить посилання на Lobby) або глобальним;

− ReadyCheck – клас для перевірки готовності. У будь якій кімнаті перед початком активності можна перевірити чи усі користувачі готові;

− класи пов’язані із активностями описано у розділі 3.1.5.

Пов’язаність основних класів видно на наступному рисунку 3.2.

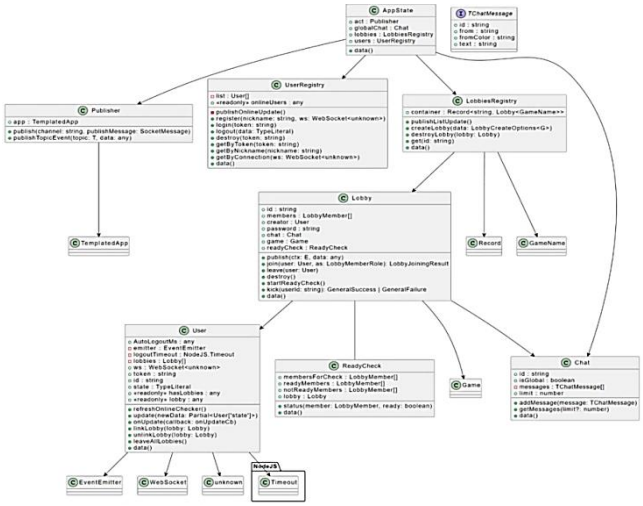

Рисунок 3.2 – UML-схема основних класів стану веб-сайту

На рисунку 3.2 деякі класи не містять опису, Game та GameName будуть описані у наступних підрозділах. Клас TemplatedApp та WebSocket є класами бібліотеки uWebSockets.js саме через них відбувається комунікаціє із клієнтом. Record та unknown є нативними типами мови TypeScript. Класи EventEmitter та Timeout надаються платформою Node.js.

3.1.4 Управління подіями на основі часових параметрів

Інтерактивні активності передбачають виконання сценаріїв, подій чітко пов’язаних із часовими параметрами, основними такими параметрами є: час коли має розпочатись подія, довжина цієї події та час закінчення події. При цьому події можуть бути скасованими взагалі, чи відбуватись раніше ніж було заплановано.

Управління такими подіями було реалізовано на базі бібліотеки node-schedule, вона дозволяє планувати завдання (англ. jobs) – довільні функції, для виконання на певні дати з додатковими правилами повторення. Для того щоб отримати ширший, функціонал зручний для використання у поточному проєкті, був створений модуль Chronos. Модуль містить три класи (див. рис. 3.3):

– DelayedEvent – клас подій які мають бути виконані через деякий час. Конструктор приймає функцію яка буде виконана у момент коли відкладений час буде вичерпаною. Надає можливість скасовувати події, запускати їх раніше та ставити їхнє очікування на паузу. Містить поле completion типу Promise – використовуючи це поле можна дочекатись виконання події через ключове слово await, маючи можливість писати код у синхронному стилі, або використовувати метод then() пишучи у стилі promises-chaining;

– TimeHall – клас який представляє собою множину пов’язаних подій і надає можливість оперувати усіма подіями одночасно. Особливо корисно коли події у деякій кімнаті потрібно ставити на паузу, а потім продовжити гру;

– Chronos – клас який надає можливість створювати екземпляри типу TimeHall.

Окрім раніше описаних типів на схемі зображено клас Job – це клас наданий бібліотекою node-schedule для планування подій.

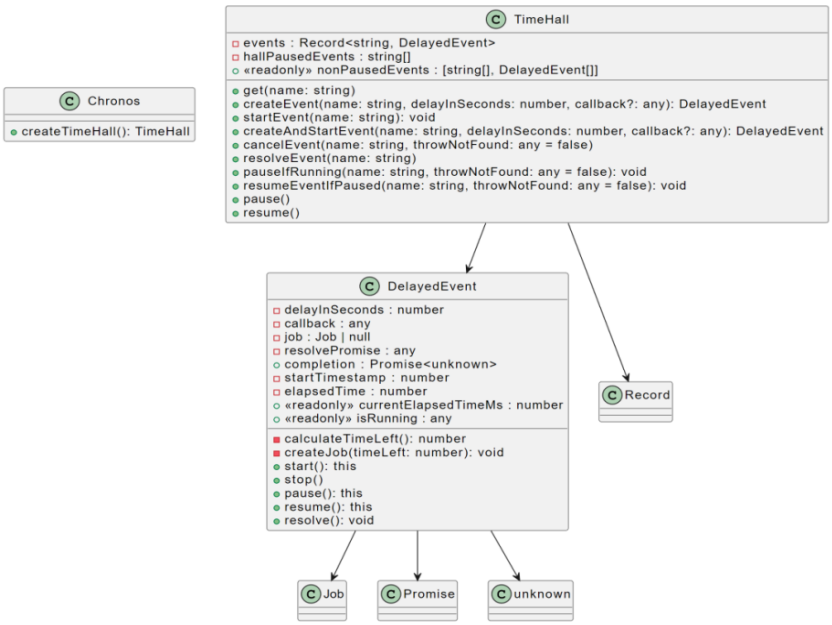

Рисунок 3.3 – UML-схема класів модуля Chronos

3.1.5 Підхід до реалізації активностей

Для реалізації активностей було створено абстрактні класи Game, GameSession, Player (див. рис. 3.4). Кожна із активностей наслідує функціонал цього класу та доповнює його.

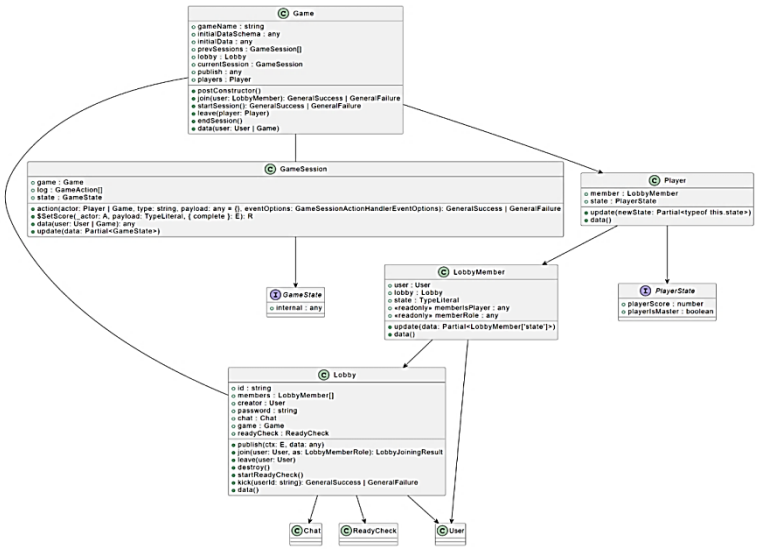

Рисунок 3.4 – UML-схема абстрактного класу активності (гри)

Варто розрізняти між грою та ігровою сесією. Гра містить методи для початку сесії, множину гравців, необхідні дані для старту і т. д. Сама ж ігрова сесія містить логіку для обробки дій користувачів, сценарії та події.

3.1.5.1 Flux-подібний підхід обробки дій

Для того щоб усі класи мали єдиний підхід було реалізовано підхід схожий до архітектурного паттерну Flux (див. рис. 3.5) – дані зберігаються у деякому об’єкті, можуть бути змінені лише чітко визначеними діями, лише однонаправлений потік даних.

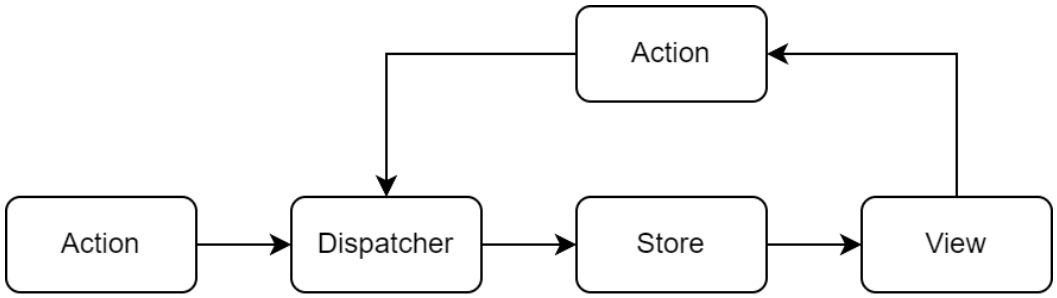

Рисунок 3.5 – Схема архітектурного паттерну Flux

Для потреб проєкту, був розроблений наступний підхід (див. рис. 3.6).

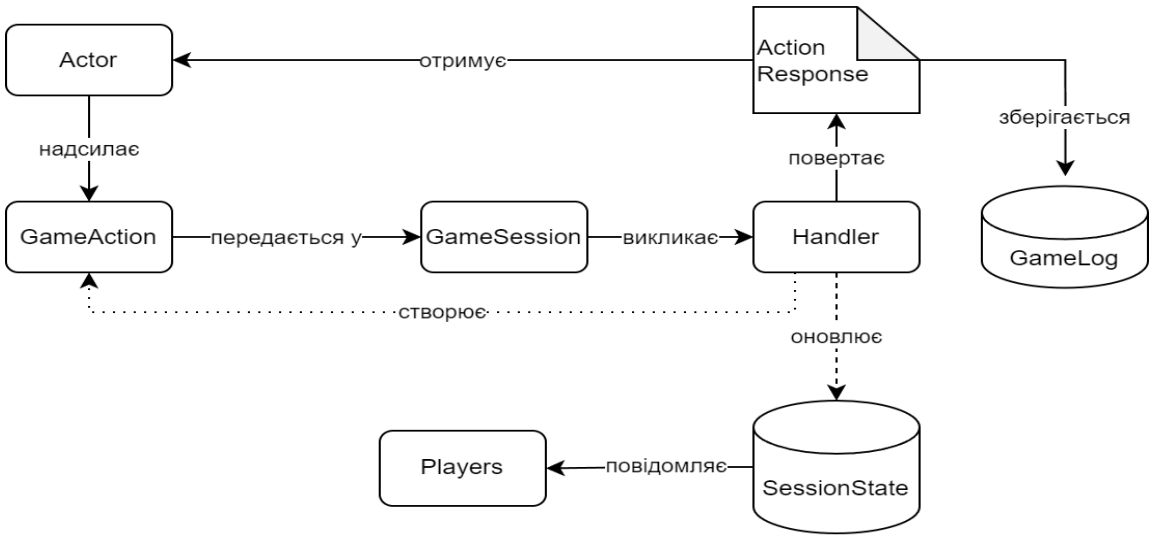

Рисунок 3.6 – Схема обробки ігрових дій

Будь яка зміна даних має відбуватись у рамках конкретного обробника, визначеного як метод сесії. На рисунку 3.4 позначено метод який буде

доступний усім іграм – $SetScore. Адміністратор кімнати може довільно змінювати рахунок кожному із гравців. Усі обробники починаються із символу $ для того щоб відрізнити їх від інших методів і мати можливість створення допоміжних типів.

На схемі під Actor розуміється гравець, який здійснює деяку дію, або сама гра, коли деяка дія є результатом іншої. Це позначається необов'язковим зв'язком від Handler до GameAction – інколи одні ігрові дії будуть запускати інші, інколи ні – усе залежить від ігрових правил.

Результатом виклику обробника завжди буде тип ActionResponse. Оскільки запити із діями приходять зі сторони клієнта, завжди потрібно валідувати правомірність цих дій для запобігання зловмисних дій та відлову помилок. Перед виконанням обробника, слід переконатись що актор який хоче здійснити дію, дійсно має право на таку дію на даний момент. У випадку якщо усі вимоги до актора та даної дії виконані, повертається об'єкт із полем success: true, у іншому випадку success: false та поле message: string із описом помилки. Усі результати дій зберігаються у ігровий журнал, що допомагає ретроспективно аналізувати ігрові сесії.

Також під час обробки дії, гра може змінювати стан сесії. При будь якій онові гравцям надсилається повідомлення про зміну стану і гравці тут же це бачать. Під час обробки, гра може запукати інші дії та створювати події за допомогою модуля Chronos.

3.1.5.2 Обробники дій: Return Early патерн та декоратори

Як було зазначено раніше, будь яка дія надіслана користувачем має проходити валідацію на правомірність такої дії: чи дійсно такий гравець існує, чи дійсно саме він має винувати таку дію, чи може бути виконана дія саме у даний момент гри і т.д. Маючи велику множину перевірок яка має бути проведена перед виконанням дії дуже просто написати важкий код який міститиме велику кількість вкладених операторів if. Такий код буде читатись важко і буде його подальше розширення та налагоддження буде складним. Лістинг 3.9 містить приклад із двома умовами (може бути більше):

Лістинг 3.9 – Приклад обробника із вкладеними операторами if

```
$SomeAction(actor: A, payload: { someValue: string }, { complete }: E): R {
    if (у м о в а 1) {
          if (умова 2) {
            /* основна логіка */
            return {
                success: true
              }
        }
          return {
            success: false,
            message: “reason 2”
          }
    }
    return {
        success: false,
        message: “reason 1”
    }
}
```

Кінець Лістингу 3.9

При застосуванні Return Early патерну [19] цей метод буде відрефакторений до наступного стану:

Лістинг 3.10 – Приклад обробника із Return Early паттерном

```
$SomeAction(actor: A, payload: { someValue: string }, { complete }: E): R {
    if (!умова 1) return {
        success: true,
        message: “reason 1”
    }
    if (!умова 2) return {
        success: true,
        message: “reason 2”
    }
    /* основна логіка */
    return {
        success: true
    }
}
```

Кінець Лістингу 3.10

Такий підхід уже робить код легшим для читання та підтримки, проте його теж можна покращити. Оскільки проєкт написано із використанням TypeScript, при встановлені параметру для компіляції experimentalDecorators: true є можливість використання декораторів для методів класів, а отже такі рядки із операторами if можуть бути винесені у спеціальні декоратори.

Лістинг 3.11 – Приклад обробника із використанням захисних декораторів

```
@Guard(умова 1, "reason 1")
@Guard(умова 2, "reason 2")
$SomeAction(actor: A, payload: { someValue: string }, { complete }: E): R {
    /* основна логіка */
    return {
        success: true
    }
}
```

Кінець Лістингу 3.11

Код стає значно коротшим, при цьому залишаючи еквівалентну логіку. Тепер обробник містить лише самий необхідний код, а усі перевірки правомірності винесені ще до назви самого обробника дії.

Для ігрових сесій було реалізовано декілька таких захисних декораторів, реалізуючих складну логіку. У декораторі ActedBy (Лістинг 3.12) перевіряються дані гравця та стан поточної сесії. Решта декораторів – GameOnlyActed, GameAndMasterOnlyActed, MasterOnlyActed, PlayersOnlyActed – побудовні на основному ActedBy, надаючи ще більшу читаємість коду.

Лістинг 3.12 – Декларація захисних декораторів для обробників ігрових дій

```
type GuardPredicate<GS extends GameSession = GameSession, AC extends Player | Game = Player | Game> = (session: GS) =>
(actor: AC) => boolean
const ActedBy =
    <GS extends GameSession = GameSession, AC extends Player | Game = Player | Game>(
        predicate: GuardPredicate<GS, AC>,
        message = 'This actor have no permission to act this action!'
    ) =>
    (_target: any, _propertyName: string, descriptor: PropertyDescriptor) => {
        const originalMethod = descriptor.value
```

Продовження лістингу 3.12

```
descriptor.value = function (actor: A, payload: P, { complete }: E) {
            if (!predicate(this as GS)(actor as AC)) {
                complete()
                return {
                    success: false,
                    message
                }
            }
            return originalMethod.call(this, actor, payload, { complete })
        }
        return descriptor
    }
const GameOnlyActed = ActedBy(() => actor => actor instanceof Game)
const GameAndMasterOnlyActed = ActedBy(() => actor => actor instanceof Game
    || actor.state.playerIsMaster)
const MasterOnlyActed = ActedBy(() => actor => actor instanceof Player && actor.state.playerIsMaster)
const PlayersOnlyActed = ActedBy(() => actor => actor instanceof Player)
```

Кінець Лістингу 3.12

3.1.6 Реалізація клієнтської сторони

Усі компоненти для відображення інтерфейсу активностей було реалізовано за допомогою бібліотеки React у поєднанні із можливостями фреймворку Next.js

3.1.6.1 Реалізація React-компонентів ігор за допомогою фабрики

На стороні сервера спільна логіка усіх активностей була реалізована за допомогою наслідування: класи Jeopardy, TicTacToe, Clicker є дочірніми класами абстрактного класу Game. Проте використати аналогічне наслідування на клієнтській стороні не є можливим – компоненти написані у функціональному стилі який не передбачає використання ключового слова extends, натомість було реалізовано фабрику яка приймає загальний тип гри та React-компонент і повертає хуки та компоненти які і використовуються для відображення гри. Результуючий компонент автоматично отримує функціонал для обробки таких подій як:

– приєднання нових гравців;

– вихід гравців із гри;

– початок сесії;

– кінець сесії;

– початкове отримання даних гри.

Результуючі хуки використовуються для:

– отримання даних сесії із будь якого вкладеного компоненту;

– додання обробників ігрових дій для будь якого вкладеного компоненту;

– отримання функції для надсилання дій.

Завдяки дому що ми передаємо узагальнений тип до фабрики компонентів – усі результуючі методи безпечні з точки зору типів та надають IntelliSense для розробника.

Використання фабрики виглядає наступним чином (на прикладі гри Jeopardy):

Лістинг 3.13 – Декларація компоненту JeopardyView

```
export const [JeopardyView, useJeopardy, useJeopardyAction, useActionSender] = createGame<Jeopardy>(() => {
    const [isPackLoading, setIsPackLoading] = useState(true)

    return (
        <>
            <JeopardyPlayersHeader />
            <JeopardyCanvas isPackLoading={isPackLoading} setIsPackLoading={setIsPackLoading} />
            <JeopardyPreSession isPackLoading={isPackLoading} />
            <JeopardySounds />
            <JeopardyControls />
        </>
    )
})
```

Кінець Лістингу 3.13

Повний код декларації функції абстрактної фабрики наведено у Додатку В.

3.1.6.2 Динамічний імпорт даних зі сторони клієнта

Компоненти для ігор мають завантажуватись для клієнти лише у випадку коли користувач знаходиться у кімнаті із цією грою. Реалізовано це за допомогою динамічних імпортів на базі Next.js. У момент монтування компоненту Lobby, виконується хук для завантаження компоненту гри.

Лістинг 3.14 – Динамічний імпорт компоненту гри

```
useEffect(() => {
    switch (lobby.gameName) {
        case 'Clicker':
            game.current = dynamic(() => import('client/features/games/clicker/ClickerView')
                .then(mod => mod.ClickerView), { loading: () => <LoadingOverlay isLoading={true} /> })
            break
        case 'TicTacToe':
            game.current = dynamic(() => import('client/features/games/tic-tac-toe/TicTacToeView')
                .then(mod => mod.TicTacToeView), { loading: () => <LoadingOverlay isLoading={true} /> })
            break
        case 'Jeopardy':
            game.current = dynamic(() => import('client/features/games/jeopardy/JeopardyView')
                .then(mod => mod.JeopardyView), { loading: () => <LoadingOverlay isLoading={true} /> })
            break
        default:
            return
    }
    setIsLoaded(true)
}, [])
```

Кінець Лістингу 3.14

Реалізувати динамічний імпорт використовуючи шаблонізовані рядки (англ. template string) передаючи назву гри у рядок який передається як аргумент у функцію import не можливо, оскільки компілятор очікує на чітко визначений рядок – тому використовується оператор switch. У випадку якщо кількість активностей значно зростатиме – використання плагіну Generate Index спростить написання такого коду.

Окрім імпорту React-компонентів, також була проведена оптимізація для завантаження аудіо-фалів використаних для ігор. Був створений React-контекст, що надає змогу програвання звуків із будь-якого компоненту. У момент коли звук має бути програним уперше, відбувається запит до сервер на завантаження ресурсу, після завантаження ресурс зберігається і при подальших подіях коли цей ресурс має бути програним – він буде програним із локального кешу. Повний код декларації компоненту наведений у Додатку Г.

Активність Jeopardy може містити для своєї ігрової сесії велику кількість ресурсів: аудіо, відео, зображення. Сумарний розмір усіх ресурсів може

перевищувати 100 мегабайт, тому окремо для цієї активності було реалізоване завантаження усіх ресурсів перед початком сесії – таким чином ресурси для усіх користувачів будуть уже завантажені ще перед початком гри, що робить ігровий процес справедливим для усіх гравців. Реалізовано це аналогічно, за допомогою хука useEffect (Лістинг 3.15).

Лістинг 3.15 – Завантаження ресурсів Jeopardy перед початком гри

```
const Resources = useRef<JeopardyMedia>({
    Audio: {}, Images: {}, Video: {}
})
useEffect(() => {
    ;(async () => {
        props.setIsPackLoading(true)
        const packArchive = await fetchPack(game.initialData.pack.value)
        Resources.current = await getMediaFilesFromPack(packArchive)
        props.setIsPackLoading(false)
    })()
}, [game.initialData])
```

Кінець Лістингу 3.15

3.2 Тестування та налагодження інформаційно-комп'ютерної системи

Тестування програмного продукту необхідна для: перевірки коректності функціоналу, впевненості у якості користувацького досвіду, виявлення дефектів та помилок, відповідності вимогам та очікуванням.

3.2.1 Unit тестування

Unit-тести повинні перевіряти роботу критично важливого та часто використовуваного функціоналу – певних функцій, класів, модулів. Основними модулями для яких були реалізовані unit-тести є класи для управління подіями на основі часових параметрів (Chronos).

Для написання unit-тестів було використано бібліотеку Jest. Тестування складається із опису можливих сценаріїв використання модуля та очікуваних результатів. У випадку якщо один модуль залежить від іншого, необхідно створити імітацію цього модуля (англ. mock).

Приклад для класу TimeHall та його методу createEvent наведено у Лістингу 3.16 (решта методів тестуються аналогічно).

Лістинг 1.16 – Unit-тестування класу TimeHall

```
import { TimeHall } from './TimeHall'
import { DelayedEvent } from './DelayedEvent'
jest.mock('./DelayedEvent', () => {
    return {
        DelayedEvent: jest.fn().mockImplementation(() => {
            return {
                start: jest.fn(),
                pause: jest.fn(),
                resume: jest.fn(),
                resolve: jest.fn(),
                isPaused: false,
                completion: Promise.resolve()
            }
        })
    }
})
describe('TimeHall', () => {
    beforeEach(() => jest.clearAllMocks())
    describe('createEvent', () => {
        it('should create a new DelayedEvent and store it', () => {
            const timeHall = new TimeHall()
            const event = timeHall.createEvent('event1', 10)

            expect(DelayedEvent).toHaveBeenCalledTimes(1)
            expect(DelayedEvent).toHaveBeenCalledWith(10, expect.any(Function))
            expect(timeHall['events']).toHaveProperty('event1', event)
        })
    })
})
```

Кінець Лістингу 3.15

Після виконання тестів, розробник має впевнитись що усі вони виконуються успішно та без помилок (див. рис. 3.7).

```
Hryhola@KIBER-PC MINGW64 /d/Code/next-game (main)
$ yarn test
yarn run v1.22.5
$ jest
 PASS  util/chronos/TimeHall.test.ts (5.963 s)
 PASS  util/chronos/DelayedEvent.test.ts (11.953 s)

Test Suites: 2 passed, 2 total
Tests:       15 passed, 15 total
Snapshots:   0 total
Time:        12.575 s
Ran all test suites.
Done in 14.60s.
```

Рисунок 3.7 – Повідомлення про успішне виконання unit-тестів

Перед релізом кожної версії програмного забезпечення необхідно бути впевненим що усі тести виконуються успішно. Зробити це можна за допомогою git-хуків.

3.2.2 Ручне функціональне тестування

«Мета функціонального тестування (англ. functional Testing) – виявлення невідповідностей між реальною поведінкою реалізованих функцій і очікуваною поведінкою відповідно до специфікації і вимог. Функціональні тести повинні охоплювати всі реалізовані функції з урахуванням найбільш ймовірних типів помилок» [13].

При ручному тестуванні (англ. manual testing) тестувальники самостійно виконують тести, не використовуючи ніяких засобів автоматизації. Основна ідея ручного тестування полягає у тому, що людина (тестер) взаємодіє з програмним забезпеченням так, як це робили б користувачі, та перевіряє його роботу з різних перспектив, вводить різні дані, виконує різні дії та перевіряє результати.

Усі очікувані сценарії зазначені у прототипах були успішно перевірені на розгорнутому сайті із використанням:

– різних браузерів (останніх версії на момент написання роботи):

1) Google Chrome;

2) Mozilla Firefox;
3) Opera;
4) Edge;

− різних девайсів: ПК, планшет, смартфони.

Основний фокус тестування був на мобільні девайси, адже усі компоненти та загальний дизайн розроблявся притримуючись підходу mobile-first.

Приклади розроблених та вручну протестованих компонентів наведено на рисунках 3.8 – 3.10.

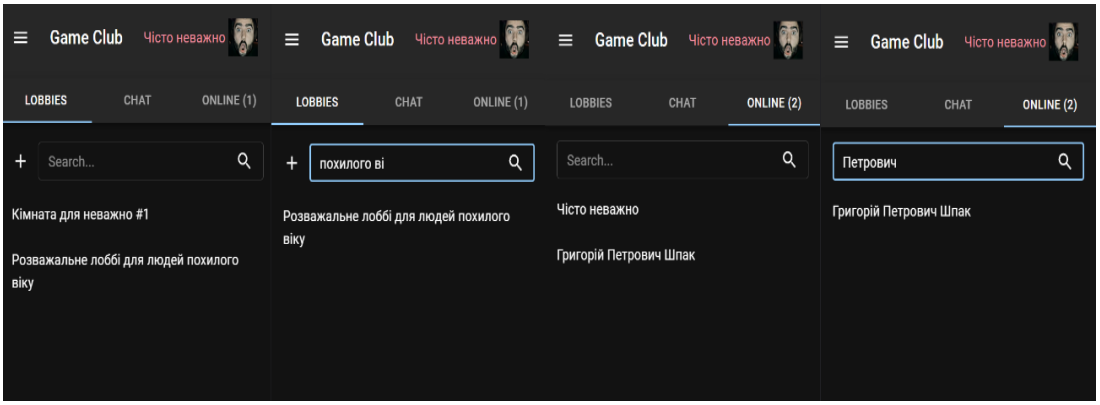

Рисунок 3.8 – Тестування компонентів пошуку на домашній сторінці

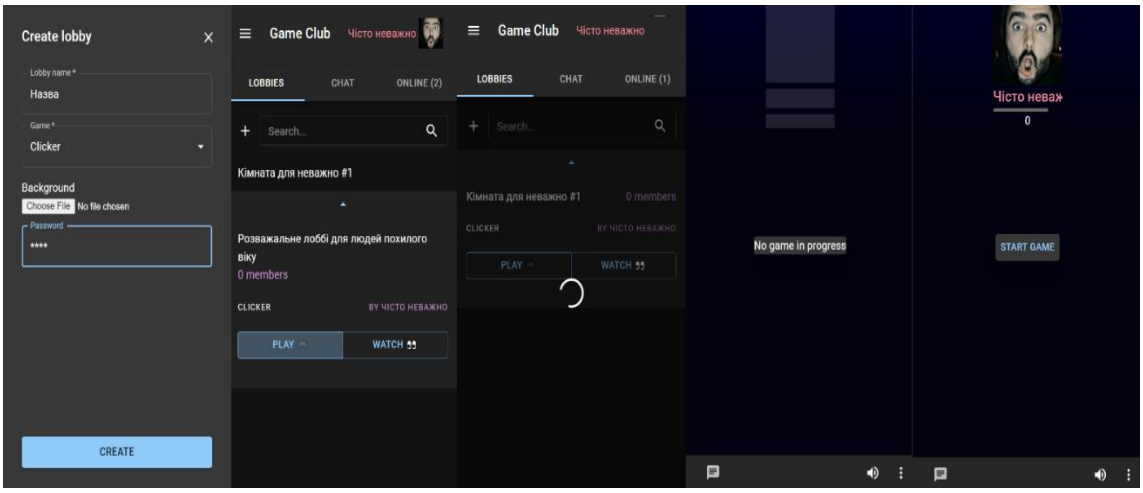

Рисунок 3.9 – Тестування процесу створення та виходу у кімнату

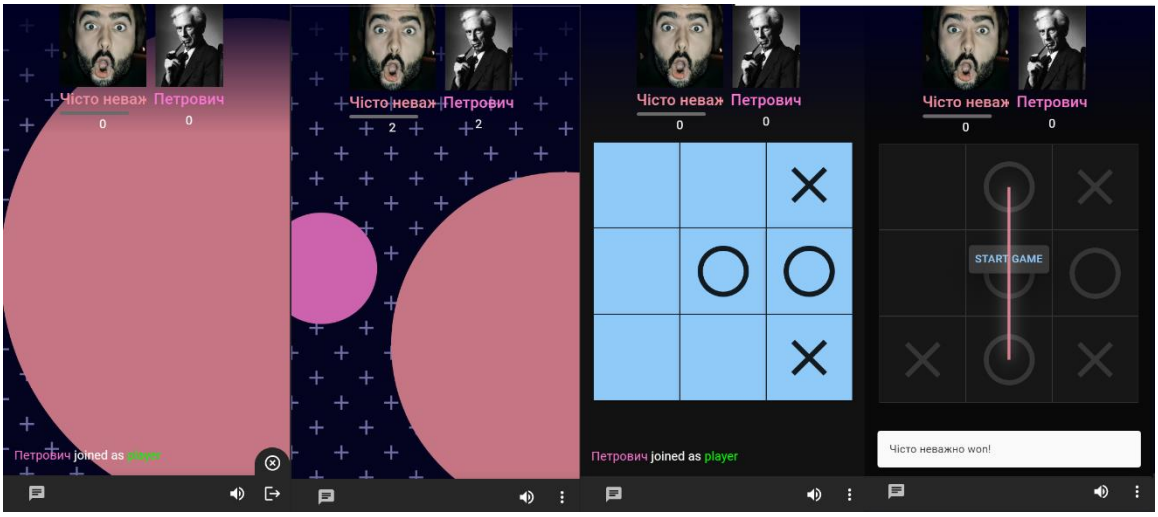

Рисунок 3.10 – Тестування інтерактивних активностей

3.3 SEO оптимізація

Оптимізація вебсайту за допомогою Next.js та мета-тегів була здійснена для покращення продуктивності та SEO.

Застосування Next.js дозволило виконувати серверний рендеринг вебсайту. Це означає, що сторінки генеруються на сервері перед їх відправкою до клієнта. Це поліпшує швидкодію сайту, оскільки користувачі отримують вже готові сторінки, що сприяє швидшому відображенню контенту.

Використання мета-тегів дозволило контролювати метадані сторінок, такі як заголовки, описи, ключові слова тощо. Це допомагає пошуковим системам краще індексувати та розуміти структуру сайту. У подальшому також можна використовувати динамічні мета-теги для кожної сторінки, щоб забезпечити унікальність та релевантність метаданих.

Важливим фактором для залучення користувачів є висока доступність (a11y). Проведений аналіз розробленого вебсайту додатком Lighthouse показав наступне (див. рис. 3.11).

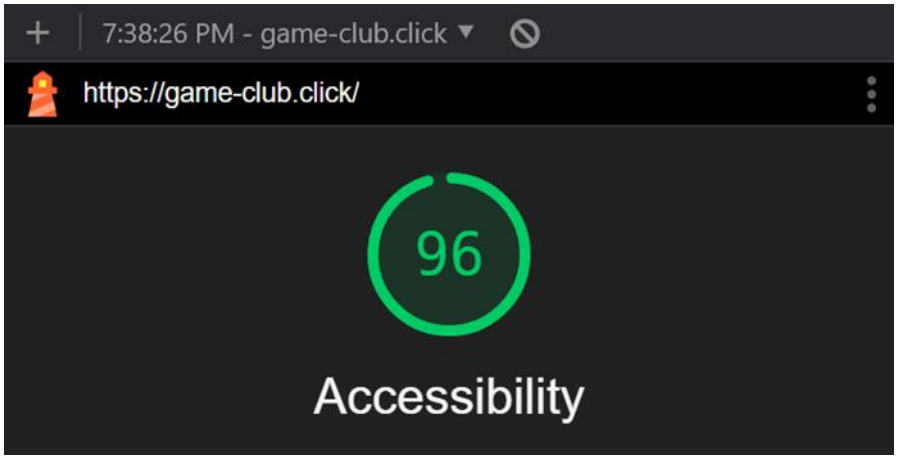

Рисунок 3.11 – Результат тестування додатком Lighthouse

Висока доступність була реалізована за допомогою бібліотеки MUI що містить готові компоненти розроблені професіоналами для підтримки високого рівня доступності.

3.4 Критика та перспективи проєкта

Будь який проєкт не буде ідеальним та міститиме місця для покращень. При подальшому розвитку проєкту варто буде врахувати наступні пункти:

− перехід до повноцінної СУБД. Наразі усі дані зберігаються напряму у оперативній пам’яті серверу. Це надає надзвичайну швидкість та інтуїтивність, проте при значному розширенні проєкта такий код може бути складним для підтримки. Перехід до MongoDB, Firebase, MySQL і т.д. може вирішити таку потенційну проблему;

− точніше управління ігровими подіями. Події із часовими параметрами потребують точності, проте однопотокова мова JavaScript при високому навантаженні може допускати відхилення у ту чи іншу сторону, що викликатиме неочікувану поведінку. Реалізація модуля для управління подіями на більш продуктивній мові нижчого рівня (такій як C++, Rust) може вирішити потенційну проблеми;

− протидія DDoS. Задля запобігання атакам, є сенс інтеграції із сервісами захисту вебсайтів, такими як reCAPTCHA;

− фінансовий розвиток. Для підтримки проєкту є сенс інтеграції із сервісами реклами, а також реалізація платних активностей.

Висновки до розділу 3

У даному розділі було проведено практичну реалізацію об’єкта проєктування для розробки вебсайту з інтерактивними активностями в реальному часі. Було розглянуто архітектуру вебсерверів та використання абстрагування над бібліотекою uWebSocket.js. Організація даних та взаємодія через сокети була реалізована, що дозволяє забезпечити швидку та ефективну комунікацію між клієнтською та серверною частинами.

Управління подіями на основі часових параметрів було реалізовано так, що дозволяє планувати та керувати виконанням активностей відповідно ігрових правил. Використовувався Flux-подібний підхід та паттерн Return Early для обробки дій, що сприяє чистоті та структурованості коду.

Реалізація клієнтської сторони використовувала React-компоненти та динамічний імпорт даних для швидкої та плавної роботи ігор у браузері.

Тестування системи включало unit-тестування для перевірки окремих компонентів та ручне функціональне тестування, що дозволяло перевірити роботу системи в реальних умовах.

Для поліпшення видимості вебсайту у пошукових системах була проведена SEO оптимізація, що включала використання мета-тегів та інших методів для поліпшення індексації та ранжування сторінок у пошукових системах.

Проєкт має перспективи для подальшого розвитку. Розроблені функціональні можливості та оптимізація сайту дозволяють надати користувачам зручну та захоплюючу взаємодію з інтерактивними активностями у реальному часі.

ВИСНОВКИ

У даній кваліфікаційній роботі було розглянуто процес розробки, алгоритми та принципи для реалізації вебсайту для комунікації та інтерактивних активностей у реальному часі з використанням протоколу WebSocket. Було проведено аналіз технологій протоколу WebSocket, визначено його відмінності від інших протоколів і розглянуто та проаналізовано різні бібліотеки та технології для використання WebSocket.

На основі обраних технологій та розроблених вимог було розроблено вебсайт з використанням Next.js та uWebSocket.js. Сайт має просту структуру і забезпечує доступ до інтерактивних активностей (розроблено три гри), які можуть використовуватися для комунікації та розваги у реальному часі. Особлива увага була приділена продуктивності і простоті використання сайту, зокрема, вимога необов'язковості реєстрації користувачів для доступу до активностей шляхом введення їх імені.

Розроблена система пройшла тестування, включаючи unit-тестування та ручне функціональне тестування. Також було здійснено SEO оптимізацію для поліпшення видимості сайту у пошукових системах.

Було здійснено розгортання сайту.

Робота має практичну цінність, оскільки використовує сучасні технології і методи розробки для створення вебсайту з інтерактивними активностями. Розроблені підходи та методи можуть бути використані у майбутньому для полегшення розробки і покращення продуктивності подібних проєктів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Clyde. Discord activities: play games and watch together. *Discord*. 2023. URL: https://discord.com/blog/server-activities-games-voice-watch-together (дата звернення: 29.05.2023).

2. Slack. Enabling interactions with bots. *Slack*. 2023. https://api.slack.com/bot-users (дата звернення: 29.05.2023).

3. Замкнені. Квести війни. *Замкнені.* 2023. *URL:* https://online.cryptex.games/ (дата звернення: 29.05.2023).

4. В. Хіль. SIGame: Комп'ютерна гра. *VladimirKhil*. 2023. URL: https://vladimirkhil.com/si/game (дата звернення: 29.05.2023).

5. A High-Performance Web Server and Reverse Proxy Server. URL: https://www.linuxjournal.com/article/10108 (дата звернення: 29.05.2023).

6. Biswas N. Relationship Between Node.Js and V8: A Complete Overview. *KnowledgeHut.* 2023. URL: https://www.knowledgehut.com/blog/web-development/nodejs-and-v8 (дата звернення: 29.05.2023).

7. Neutkens T. Markbåge S. Next.js 13.4. *NextJS.* 2023. URL: https://nextjs.org/blog/next-13-4 (дата звернення: 29.05.2023).

8. Sycamore S. An introduction to the MUI ecosystem. *MUI*. 2022. URL: https://mui.com/blog/mui-product-comparison/ (дата звернення: 29.05.2023).

9. The WebSocket Protocol. Internet Engineering Task Force. URL: https://datatracker.ietf.org/doc/html/rfc6455 (дата звернення 29.05.2023).

10. Stangvik E., Kazemier A., Pinca L. ws: a Node.js WebSocket library. *GitHub*. 2023. URL: https://github.com/websockets/ws (дата звернення 29.05.2023).

11. Arrachequesne D. Socket.IO 4.4.0. *Socket.IO*. 2021. URL: https://socket.io/blog/socket-io-4-4-0/ (дата звернення 29.05.2023).

12. Sumners J., Vedova T., Collina M. Fastify: Fast and low overhead web framework, for Node.js 2023. *GitHub.* URL: https://github.com/fastify/fastify (дата звернення 29.05.2023).

13. Hultman A . uWebSockets: Simple, secure & standards compliant web server for the most demanding of applications. *GitHub*. 2023. URL: https://github.com/uNetworking/uWebSockets (дата звернення 29.05.2023).

14. Hultman A. 100k secure WebSockets with Raspberry Pi 4. *Medium*. 2020. URL: https://unetworkingab.medium.com/100ksecure-websockets-with-raspberry-pi-4-1ba5d2127a23 (дата звернення 29.05.2023).

15. Hultman A. Serving 100k requests/second from a fanless Raspberry Pi 4 over Ethernet. *Medium*. 2021. URL: https://unetworkingab.medium.com/serving-100k-requests-secondfrom-a-fanless-raspberry-pi-4-over-ethernet-fdd2c2e05a1e (дата звернення 29.05.2023).

16. Chee E. Intro to μWebSockets.js. *EdisonChee.com*. 2020. URL: https://edisonchee.com/writing/intro-to-%C2%B5websockets.js/ (дата звернення: 29.05.2023).

17. Garnett A. How To Secure Nginx with Let's Encrypt on Ubuntu 22.04. *DigitalOcean*. 2022. URL: https://www.digitalocean.com/community/tutorials/how-to-secure-nginx-with-let-s-encrypt-on-ubuntu-22-04 (дата звернення: 29.05.2023).

18. Tran T. How To Configure Nginx as a Reverse Proxy on Ubuntu 22.04. *DigitalOcean*. 2022. URL: https://www.digitalocean.com/community/tutorials/how-to-configure-nginx-as-a-reverse-proxy-on-ubuntu-22-04 (дата звернення: 29.05.2023).

19. Menaia L. Return Early Pattern. *Medium*. 2020. URL: https://medium.com/swlh/return-early-pattern-3d18a41bba8 (дата звернення: 29.05.2023).

ДОДАТКИ

Додаток А

Конфігурація Nginx (Reverse Proxy та SSL)

```
client_max_body_size 100M;
server {
        root /var/www/game-club.click/html;
        index index.html index.htm index.nginx-debian.html;
        server_name game-club.click www.game-club.click;
        location / {
                proxy_pass https://localhost:3000;
                proxy_http_version 1.1;
                proxy_set_header Upgrade $http_upgrade;
                proxy_set_header Connection 'upgrade';
                proxy_set_header Host $host;
                proxy_cache_bypass $http_upgrade;
        }
        location /ws/ {
                proxy_http_version 1.1;
                proxy_set_header Host $http_host;
                proxy_set_header Upgrade $http_upgrade;
                proxy_set_header Connection "Upgrade";
                proxy_pass https://localhost:5555/ws;
                proxy_redirect off;
                proxy_buffering off;
        }
        location /res/ {
                try_files $uri $uri/ =404;
        }
    listen [::]:443 ssl; # managed by Certbot
    listen 443 ssl; # managed by Certbot
    ssl_certificate /etc/letsencrypt/live/game-club.click/fullchain.pem; # managed by Certbot
    ssl_certificate_key /etc/letsencrypt/live/game-club.click/privkey.pem; # managed by Certbot
    include /etc/letsencrypt/options-ssl-nginx.conf; # managed by Certbot
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem; # managed by Certbot
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem; # managed by Certbot
}
server {
    if ($host = www.game-club.click) {
        return 301 https://$host$request_uri;
    } # managed by Certbot
    if ($host = game-club.click) {
        return 301 https://$host$request_uri;
    } # managed by Certbot
        listen 80;
        listen [::]:80;
        server_name game-club.click www.game-club.click;
    return 404; # managed by Certbot }
```

Додаток Б

Файл wsCtx.tsx

```
import React, { useState, createContext, useContext, useRef, MutableRefObject, useEffect } from 'react'
import type { HandlerName } from 'uWebSockets/ws'
import type { SocketMessage, TopicEventHandler, RequestData, RequestHandler } from 'uWebSockets/uws.types'
import type { StateEventName } from 'uWebSockets/topicEvents'
import { getCookie } from 'cookies-next'
type HandlerOn = <C extends StateEventName | HandlerName>(context: C, handler: Function) => void
type HandlerSend = <H extends HandlerName>(context: H, data?: RequestData<H>) => void
type RequestHandlerRegistrar = <C extends HandlerName>(context: C, handler: RequestHandler<C>) => void
type EventHandlerRegistrar = <C extends StateEventName>(context: C, handler: TopicEventHandler<C>) => void
export interface WSData {
    wsRef: MutableRefObject<WebSocket | null>
    isConnected: boolean | null
    setIsConnected: (value: boolean) => void
    on: HandlerOn
    send: HandlerSend
    unsubscribe: HandlerOn
}
export const WSContext = createContext<WSData>({})
interface Props {
    children?: JSX.Element
}
export const WSProvider: React.FC<Props> = props => {
    const wsRef = useRef<WebSocket | null>(null)
    const listeners = useRef({} as Record<string, Set<Function>>)
    const [isConnected, setIsConnected] = useState<boolean | null>(null)
    const on: HandlerOn = (context: string, handler: Function) => {
        listeners.current[context] = listeners.current[context] || new Set()
        listeners.current[context].add(handler)
    }
    const unsubscribe: HandlerOn = (context: string, handler: Function) => {
        if (listeners.current[context]) {
            listeners.current[context].delete(handler)
        }
    }
    const send: HandlerSend = (context, data) => {
        if (!wsRef.current) {
            console.error('Cannot send because ws is not defined', context, data)
            return
        }

        const message: SocketMessage = {
            ctx: context,
            data: data || null
        }
        console.log('%c' + context + ' %csend', 'color: Chartreuse', '', message.data)
        const token = getCookie('token') as string | undefined
```

```
        if (token) {
            message.token = token
        }
        wsRef.current.send(JSON.stringify(message))
    }
    const messageHandler = (event: MessageEvent<any>) => {
        if (event.data === 'pong') {
            return
        }
        const message = JSON.parse(event.data)
        if (message.ctx in listeners.current) {
            listeners.current[message.ctx].forEach(listener => listener(message.data))
        }
    }
    if (wsRef.current) wsRef.current.onmessage = messageHandler
    return <WSContext.Provider value={{ wsRef, isConnected, setIsConnected, on, send, unsubscribe
}}>{props.children}</WSContext.Provider>
}
export const useWS = () => {
    return useContext(WSContext)
}
export const useRequestHandler: RequestHandlerRegistrar = (context, handler) => {
    const { on, unsubscribe } = useWS()
    useEffect(() => {
        on(context, handler)
        return () => {
            unsubscribe(context, handler)
        }
    }, [])
}
export const useEventHandler: EventHandlerRegistrar = (context, handler) => {
    const { on, unsubscribe } = useWS()
    useEffect(() => {
        on(context, handler)
        return () => unsubscribe(context, handler)
    }, []) }
```

Додаток В

Файл GameFactory.tsx

```
import React, { useEffect } from 'react'
import { PlayerData, GameSessionData, GameSessionActionsName, GameSessionAction, Game as AbstractGame } from 'state'
import { useLobby, useEventHandler, useWS } from 'client/context/list'
import { api } from 'client/network-utils/api'
import { InitialGameData } from 'state/common/game/GameInitialData'

export type GameCtxValue = {
    players: PlayerData[]
    isLoading: boolean
    isSessionStarted: boolean
    session: GameSessionData | null
    initialData?: InitialGameData
}

export const GameCtx = React.createContext<GameCtxValue | null>(null)

export const createGame = <Game extends AbstractGame>(Component: React.ComponentType<{}>) => {
    type ThisPlayerData = ReturnType<Game['players'][0]['data']>
    type ThisSession = NonNullable<Game['currentSession']>
    type ThisSessionData = ReturnType<ThisSession['data']>
    type ThisInitialData = Game['initialData']

    type ThisGameCtxValue = {
        players: ThisPlayerData[]
        isLoading: boolean
        isSessionStarted: boolean
        session: ThisSessionData | null
        initialData: ThisInitialData
    }

    const GameComponent = () => {
        const lobby = useLobby()

        const [players, setPlayers] = React.useState<ThisPlayerData[]>([])
        const [isLoading, setIsLoading] = React.useState(true)
        const [session, setSession] = React.useState<ThisSessionData | null>(null)
        const [initialData, setInitialData] = React.useState<ThisInitialData>({})

        useEventHandler('Game-Join', data => {
            setPlayers(ps => [...ps.filter(p => p.id !== data.player.id), data.player as ThisPlayerData])
        })

        useEventHandler('Game-Leave', data => {
            setPlayers(ps => [...ps.filter(p => p.id !== data.player.id)])
        })
```

```
useEventHandler('Game-PlayerUpdate', data => {
    setPlayers(ps =>
        ps.map(player =>
            player.id === data.id
                ? {
                      ...player,
                      ...data.data
                  }
                : player
        )
    )
})

useEventHandler('Game-SessionStart', ({ lobbyId, session }) => {
    if (lobbyId === lobby.lobbyId) {
        setSession(session)
    }
})

useEventHandler('Game-SessionEnd', ({ lobbyId }) => {
    if (lobbyId === lobby.lobbyId) {
        setSession(null)
    }
})

useEventHandler('Game-SessionUpdate', ({ lobbyId, data }) => {
    if (lobbyId === lobby.lobbyId) {
        setSession(prev => ({ ...prev, ...data }))
    }
})

useEffect(() => {
    ;(async () => {
        const [response, postError] = await api
            .post('lobby-data', {
                lobbyId: lobby.lobbyId
            })
            .finally(() => setIsLoading(false))

        if (!response || !response.success) {
            return console.error(response ? response.message : postError)
        }

        lobby.setMembers(response.lobby.members)
        lobby.setGameName(response.game.name as 'Clicker')

        setInitialData(response.game.initialData)
        setPlayers(response.game.players as ThisPlayerData[])

        if (response.game.session) setSession(response.game.session as ThisSessionData)
```

```
            })()
        }, [])

        const game = {
            players,
            isLoading,
            isSessionStarted: session !== null,
            session,
            initialData
        }

        return (
            <GameCtx.Provider value={game}>
                <GameCtx.Consumer>{() => <Component />}</GameCtx.Consumer>
            </GameCtx.Provider>
        )
    }

    const useGame = () => {
        const ctx = React.useContext(GameCtx)

        if (!ctx) {
            throw new Error('useGame must be used within a GameCtxProvider')
        }

        return ctx as ThisGameCtxValue
    }

    type SessionActionName = GameSessionActionsName<ThisSession>

    type SessionAction<ActionName extends SessionActionName = SessionActionName> = GameSessionAction<ThisSession,
ActionName>

    type SessionActionPayload<ActionName extends SessionActionName> = SessionAction<ActionName>['payload']

    const useActionHandler = <T extends SessionActionName>(name: T, handler: (data: SessionAction<T>) => void) => {
        const lobby = useLobby()
        const lobbyRef = React.useRef(lobby)

        useEffect(() => {
            lobbyRef.current = lobby
        }, [lobby])

        useEventHandler('Game-SessionAction', data => {
            if (!lobbyRef.current || data.lobbyId !== lobbyRef.current.lobbyId || data.type !== name) {
                return
            }

            handler(data as unknown as SessionAction<T>)
        })
```

```
    }

    const useActionSender = () => {
        const ws = useWS()
        const lobby = useLobby()

        const wsRef = React.useRef(ws)
        const lobbyRef = React.useRef(lobby)

        useEffect(() => {
            wsRef.current = ws
        }, [ws])

        useEffect(() => {
            lobbyRef.current = lobby
        }, [lobby])

        return <T extends SessionActionName>(name: T, payload: SessionActionPayload<T>) => {
            if (!wsRef.current || !lobbyRef.current) {
                console.error('ws or lobby is not defined')

                return
            }

            wsRef.current.send('Game-SendAction', {
                lobbyId: lobbyRef.current.lobbyId,
                actionName: name as string,
                actionPayload: payload
            })
        }
    }

    return [GameComponent, useGame, useActionHandler, useActionSender] as const
}

export const useGame = () => {
    const ctx = React.useContext(GameCtx)

    if (!ctx) {
        throw new Error('useGame must be used within a GameCtxProvider')
    }

    return ctx
}
```

Додаток Г

Файл audioCtx.tsx

```
import React, { createContext, useRef, useEffect, useState } from 'react'

export const AudioCtx = createContext({
    play: async (_route: string) => {},
    setVolume: (_value: number) => {},
    toggleMute: () => {},
    volume: NaN
})

interface Props {
    children?: JSX.Element
}

export const AudioProvider: React.FC<Props> = props => {
    const context = useRef<AudioContext | null>(null)
    const gain = useRef<GainNode | null>(null)
    const soundsMap = useRef(new Map<string, AudioBuffer>())

    const [volume, setVolumeState] = useState(50)
    const prevVolume = useRef(volume)

    useEffect(() => {
        context.current = new window.AudioContext()
        gain.current = context.current.createGain()
        gain.current.gain.value = 0.5
        gain.current.connect(context.current.destination)

        return () => {
            context.current?.close()
        }
    }, [])

    const play = async (fileName: string) => {
        if (!context.current) {
            context.current = new window.AudioContext()
            gain.current = context.current.createGain()
            gain.current.gain.value = 0.5
            gain.current.connect(context.current.destination)
        }

        if (!soundsMap.current.has(fileName)) {
            const audioBuffer = await fetch(`/sounds/${fileName}`)
                .then(response => response.arrayBuffer())
                .then(arrayBuffer => context.current!.decodeAudioData(arrayBuffer))
                .catch(error => {
                    console.error(error)
```

```
            })

        if (audioBuffer) {
            soundsMap.current.set(fileName, audioBuffer)
        } else {
            return
        }
    }

    const audio = soundsMap.current.get(fileName)

    if (!audio) return

    const source = context.current.createBufferSource()
    source.buffer = audio
    // source.connect(context.current.destination)
    source.connect(gain.current!).connect(context.current.destination)

    source.start()
}

const setVolume = (value: number) => {
    if (!gain.current) return

    gain.current.gain.value = value / 100

    setVolumeState(value)
}

const toggleMute = () => {
    if (!gain.current) return

    if (gain.current.gain.value === 0) {
        gain.current.gain.value = prevVolume.current / 100
        setVolumeState(prevVolume.current)
    } else {
        prevVolume.current = volume
        gain.current.gain.value = 0
        setVolumeState(0)
    }
}

return (
    <AudioCtx.Provider
        value={{
            play,
            setVolume,
            toggleMute,
            volume
        }}
```

```
        >
            {props.children}
        </AudioCtx.Provider>
    )
}

export const useAudio = () => {
    return React.useContext(AudioCtx)
}
```