

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій

(повне найменування факультету)

Кафедра програмних засобів

(повне найменування кафедри)

Пояснювальна записка

до дипломного проекту (роботи)

бакалавр

(ступінь вищої освіти)

на тему РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ
СИНХРОНІЗАЦІЇ СПІЛЬНОГО ПЕРЕГЛЯДУ МЕДІА-КОНТЕНТУ
КОРИСТУВАЧІВ У БРАУЗЕРНОМУ СЕРЕДОВИЩІ
DEVELOPMENT OF SOFTWARE FOR SYNCHRONIZING SHARED VIEWING
OF MEDIA CONTENT IN A BROWSER ENVIRONMENT

Виконав(ла): студент(ка) 4 курсу, групи КНТ-113сп

Спеціальності 121 Інженерія програмного

(код і найменування спеціальності)

забезпечення

Освітня програма (спеціалізація)

Інженерія програмного забезпечення

СІМІЧОВ О.В.

(ПРИЗВИЩЕ та ініціали)

Керівник СКРУПСЬКИЙ С.Ю.

(ПРИЗВИЩЕ та ініціали)

Рецензент КОЦУР М. І.

(ПРИЗВИЩЕ та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
 (повне найменування закладу вищої освіти)

Факультет КНТ
 Кафедра програмних засобів
 Ступінь вищої освіти бакалавр
 Спеціальність 121 Інженерія програмного забезпечення
 (код і найменування)
 Освітня програма (спеціалізація) Інженерія програмного забезпечення
 (назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
Сергій СУББОТІН
 “ ” 2026 року

З А В Д А Н Н Я
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

СІМІЧОВА Олексій Вікторовича
 (ПРИЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Розробка програмного забезпечення для синхронізації спільного перегляду медіа-контенту у браузерному середовищі. Development of Software for Synchronizing Shared Viewing of Media Content in a Browser Environment.

керівник проєкту (роботи) к.т.н., доцент, СКРУПСЬКИЙ Степан Юрійович,
 (науковий ступінь, вчене звання, ПРИЗВИЩЕ, ім'я, по батькові)

затверджені наказом закладу вищої освіти від “ 07 ” квітня 2026 року
 № 139

2. Строк подання студентом проєкту (роботи) 10 червня 2026 року

3. Вихідні дані до проєкту (роботи) рекомендована література, технічне завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Аналіз предметної області. 2. Розробка архітектури програми. 3. Реалізація програми. 4. Експлуатація та тестування програми.

5. Перелік графічного матеріалу (з з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів)
Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРИЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4 Основна частина	СКРУПСЬКИЙ С.Ю., доцент		
Нормоконтроль	КАЛІНІНА М.В., асистент		

7. Дата видачі завдання « 20 » квітня 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	1 тиждень	Розділ 1
3	Вибір мови програмування та інших технологій	2 тиждень	Розділ 2
	розробки.		
4	Розробка архітектури програми.	2 тиждень	Розділ 3
5	Розробка програми.	3-4 тижні	Розділ 3, 4
6	Тестування та експериментальне дослідження	5 тиждень	Розділ 4
	програмного забезпечення.		
7	Оформлення пояснювальної записки та документів	6 тиждень	Додатки
	до неї.		
8	Нормоконтроль та рецензування.	7 тиждень	
9	Захист роботи.	8 тиждень	

Студент(ка)

(підпис) Олексій СІМІЧОВ
(Ім'я ПРИЗВИЩЕ)

Керівник проєкту (роботи)

(підпис) Степан СКРУПСЬКИЙ
(Ім'я ПРИЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра: 98 с., 2 табл., 33 рис., 3 дод., 16 джерел.

СИНХРОНІЗАЦІЯ, МЕДІА-КОНТЕНТ, БРАУЗЕРНЕ РОЗШИРЕННЯ, ВЕБ-СОКЕТИ, REAL-TIME, MANIFEST V3, NODE.JS, POSTGRESQL.

Об'єкт дослідження – процес інженерії програмного забезпечення для автоматизації синхронізації та колективного споживання медіа-контенту в режимі реального часу.

Предмет дослідження – програмне забезпечення, що забезпечує автоматичну синхронізацію станів відтворення між віддаленими користувачами.

Мета роботи – підвищення ефективності перегляду відеоконтенту групою користувачів.

Матеріали, методи та технічні засоби: мова програмування JavaScript, середовище виконання Node.js, система керування базами даних PostgreSQL, бібліотека для реал-тайм зв'язку WebSockets (ws), стандарт розробки розширень Manifest V3, бібліотека React (для інтерфейсу Popup).

Результати. Створення програмного забезпечення SyncPlayer, що забезпечує автоматизацію передачі команд керування безпосередньо на сторінках відеохостингів. Система дозволяє користувачам створювати віртуальні сесії, керувати правами доступу та забезпечувати ідентичність кадрів у всіх учасників з точністю до мілісекунд.

Висновки. Розроблено програмне забезпечення дозволяє покращити користувацький досвід при колективному перегляді за рахунок повної автоматизації синхронізації.

Галузь використання – організація спільного дозвілля, дистанційне навчання, онлайн-кінотеатри та платформи для роботи з медіа-матеріалами.

ABSTRACT

Explanatory note to the diploma qualifying work of the bachelor: 98 pages, 2 tables, 33 figures, 3 appendixes, 16 sources.

SYNCHRONIZATION, MEDIA CONTENT, BROWSER EXTENSION, WEBSOCKETS, REAL-TIME, MANIFEST V3, NODE.JS, POSTGRESQL.

Object of study is a process of software engineering for the automation of real-time synchronization and collective media content consumption.

Subject of study is a software for the automatic synchronization of playback states between remote users. Materials, methods, and technical tools: JavaScript programming language, Node.js runtime environment, PostgreSQL database management system, WebSockets (ws) library for real-time communication, Manifest V3 extension development standard, React library (for the Popup interface).

Results. The result of the work is the creation of the SyncPlayer software, which automates the transfer of player control commands directly on video hosting pages. The system allows users to create virtual sessions, manage access rights, and ensure frame identity for all participants with millisecond precision.

Conclusions. The developed software improves the user experience during collective viewing due to full automation of synchronization. The use of the modern Manifest V3 standard ensures the security and high performance of the client side.

Field of application. Organization of joint leisure, distance, online cinemas, and platforms for teamwork with media materials.

ЗМІСТ

	С.
Перелік скорочень та умовних познач.....	7
Вступ.....	9
1 Аналіз предметної області.....	10
1.1 Аналіз програмних засобів для організації спільного перегляду медіа-контенту	10
1.2 Порівняльний аналіз розробки та аналогів	14
1.3 Висновки за розділом	16
2 Розробка архітектури програми.....	18
2.1 Вибір засобів проєктування	18
2.2 Сценарії використання програми	20
2.3 Проєктування архітектури системи	22
2.4 Прототипування користувацького інтерфейсу	23
2.5 Структура бази даних	25
2.6 Висновки за розділом 2	28
3 Реалізація програми	30
3.1 Визначення порядку роботи програми	30
3.2 Результати реалізації програмних одиниць застосунку.....	33
3.3 Опис структур даних та об'єктів.....	42
3.4 Висновки за розділом 3	45
4 Експлуатація та тестування програми	47
4.1 Призначення програми	47
4.2 Умови виконання програми	47
4.3 Виконання і тестування програми.....	48
4.4 Висновки за розділом 4	52
Висновки	53
Перелік джерел посилання	55
Додаток А Технічне завдання	57

Додаток Б Текст програми	62
Додаток В Слайди презентації.....	90

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

API	– Application Programming Interface;
DOM	– Document Object Model;
ER-модель	– Entity-Relationship Model,
JSON	– JavaScript Object Notation;
MV3	– Manifest V3;
UI	– User Interface;
URL	– Uniform Resource Locator;
UX	– User Experience;
БД	– база даних;
ПЗ	– програмне забезпечення;
ТЗ	– технічне завдання.

ВСТУП

Цифровізація суспільства призвела до того, що значна частина міжособистісного спілкування перемістилася в онлайн-середовище. Перегляд медіа-контенту таких як фільмів, серіалів, освітніх лекцій або розважальних відео був офлайн формою спільного дозвілля. Проте в сучасних реаліях користувачі часто стикаються з неможливістю фізично зібратися разом через географічну віддаленість, робочі графіки або інші обставини.

Незважаючи на величезну кількість медіа-платформ, підтримка спільного споживання контенту в режимі реального часу залишається недостатньо розвиненою. Навіть найновіші сервіси часто не дають потрібних можливостей. Натомість споживачі одночасно натискати кнопки «Play» за командою в месенджері або використання сторонніх сервісів зв'язку, які не інтегровані з медіа-плеєрами що значно погіршує якість зображення.

Існує значна прогалина між наявністю якісного контенту та відсутністю інструментів для його спільного споживання. Сучасні соціальні мережі дозволяють ділитися посиланнями, але вони не забезпечують середовища для споживання контенту разом. Це особливо актуально для інтернет-друзів або спільнот за інтересами, де спільний перегляд міг би стати головним фактором згуртування групи.

Саме на покращенні механізму синхронізації цього досвіду без зайвих технічних перешкод, зроблено акцент у цій розробці. Програмне забезпечення має стати містком, який дозволить користувачам відчувати присутність один одного через спільне керування медіа-потокм.

Метою роботи є підвищення ефективності перегляду відеоконтенту групою користувачів, за допомогою програмного забезпечення синхронізації спільного перегляду медіа-контенту у браузерному середовищі, що дозволяє забезпечити комфортне спільне дозвілля незалежно від відстані між учасниками.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналіз програмних засобів для організації спільного перегляду медіа-контенту

Під час аналізу було виявлено низку рішень, які дозволяють організовувати спільне споживання медіа-контенту, проте більшість із них не вирішує проблему синхронізації при збереженні максимальної якості зображення [1].

Стрімінгові сервіси такі, як Netflix (рис. 1.1) або Disney+, надають лише інформаційну складову або внутрішні чати для обговорення відео постфактум. Вони не мають вбудованого функціоналу для синхронізації відтворення між віддаленими користувачами [2]. Встановлення контакту між глядачами в таких випадках відбувається за межами медіа-платформи, що призводить до десинхронізації перегляду та втрати ефекту "спільної присутності".

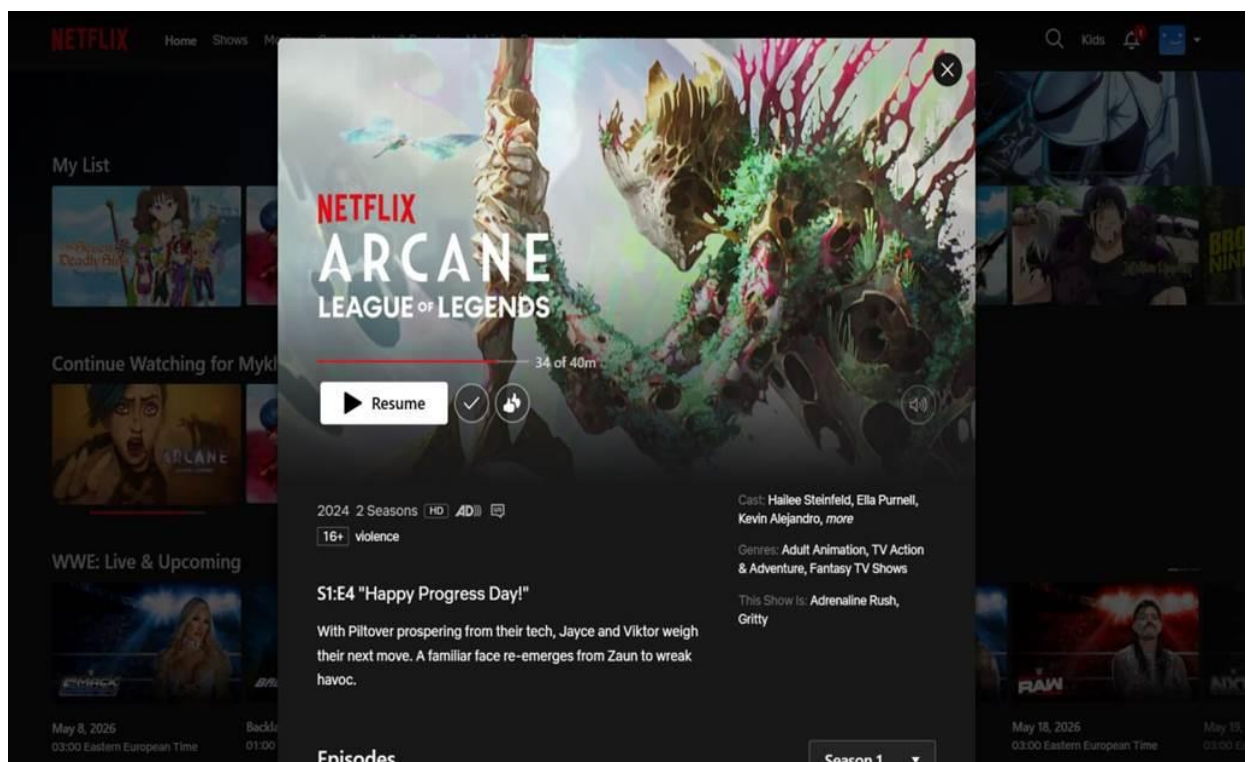


Рисунок 1.1 - Інтерфейс Netflix [3]

Discord (рис. 1.2) та Zoom можна виділити як інструменти, що часто використовуються для спільного перегляду через функцію трансляції екрана. Це дозволяє швидко зібрати групу користувачів, проте такий підхід має суттєві недоліки. Через втрату якості через стиснення користувач використовує мережні ресурси неефективно [4]. Користувач залежить від каналу зв'язку Discord, а не оригінального відеопотоку. Такий спосіб дивитися контент фактично дублює передачу важких відеоданих замість координації станів відтворення, що критично впливає на ресурси системи.

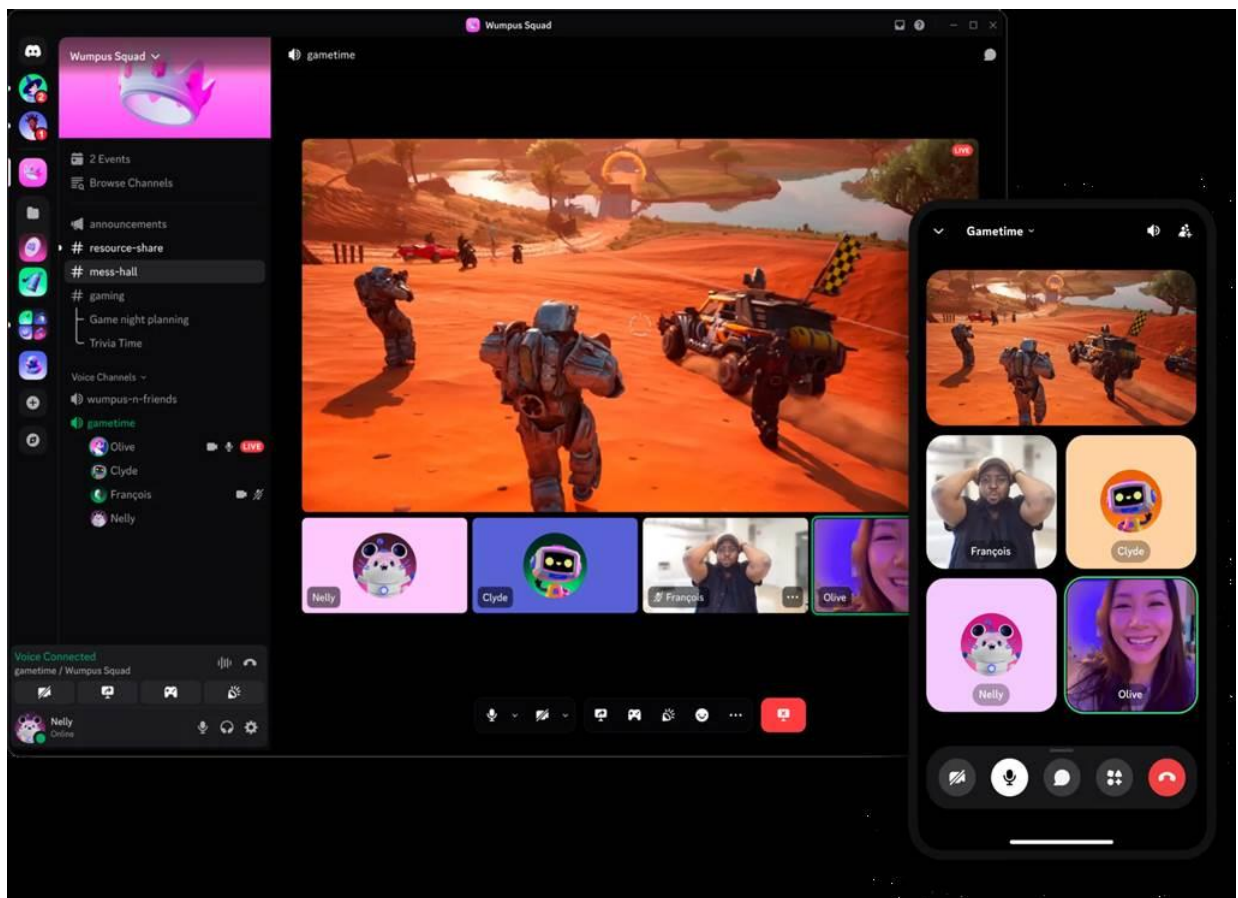


Рисунок 1.2 - Спільний перегляд контенту в Discord [5]

Teleparty (рис. 1.3) - це популярне браузерне розширення, що базується на синхронізації станів відеоплеєра. Головним обмеженням даного застосунку є його вузька спеціалізація підтриманих сайтів. Він підтримує лише обмежений перелік великих платних платформ (як-от Netflix, Disney+ чи Hulu), повністю ігноруючи вільні джерела контенту [6].

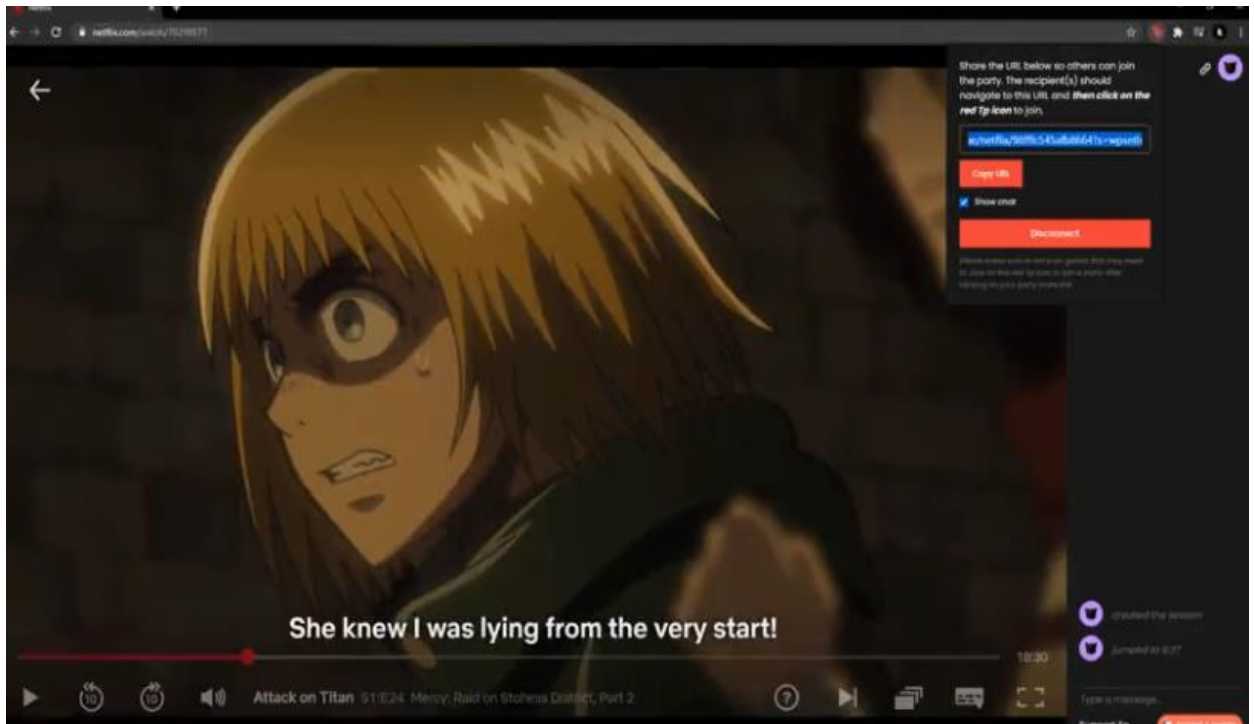


Рисунок 1.3 - Інтерфейс Teleparty [7]

Metastream (рис. 1.4) є прикладом вебзастосунку, що дозволяє додавати посилання з різних джерел у спільну чергу. Він використовує підхід синхронізації сесій, що дозволяє кожному користувачу отримувати потік безпосередньо з першоджерела. Під час використання користувачі можуть отримати дискомфорт та фрустрацію від нестандартного, перевантаженого інтерфейсу цього застосунку. Також процес додавання нових медіа-сайтів складним для звичайного користувача, тому що потребує написання специфічних селекторів для кожного нового програвача [8].

Watch2Gether (рис. 1.5) забезпечує функціонал для створення віртуальних кімнат, проте має суттєві архітектурні недоліки. Зокрема, сервіс базується на використанні вбудованого браузера, що значно звужує перелік доступних джерел відтворення. Такий підхід змушує аудиторію переходити на сторонній веб-ресурс, що позбавляє глядачів можливості залишатися у звичному інтерфейсі оригінального сайту та обмежує загальну гнучкість системи.

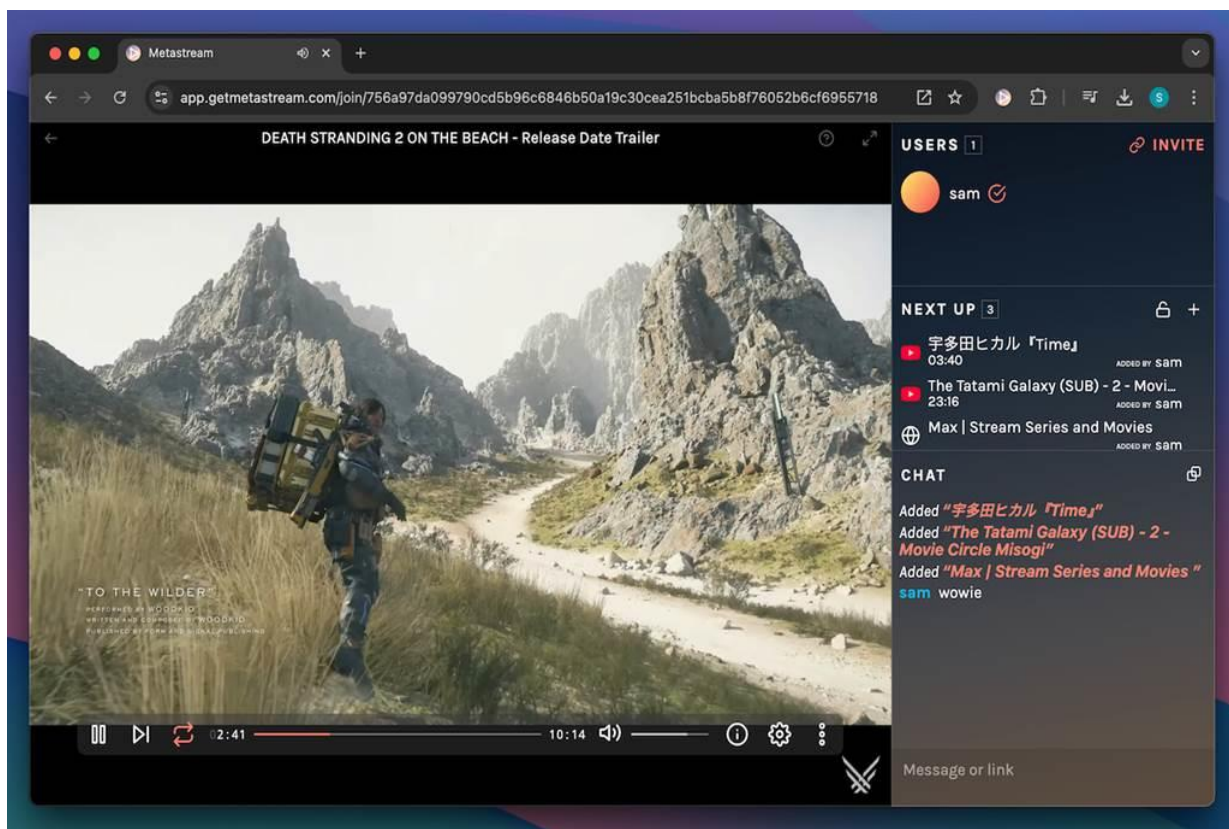


Рисунок 1.4 - Интерфейс Metastream [9]

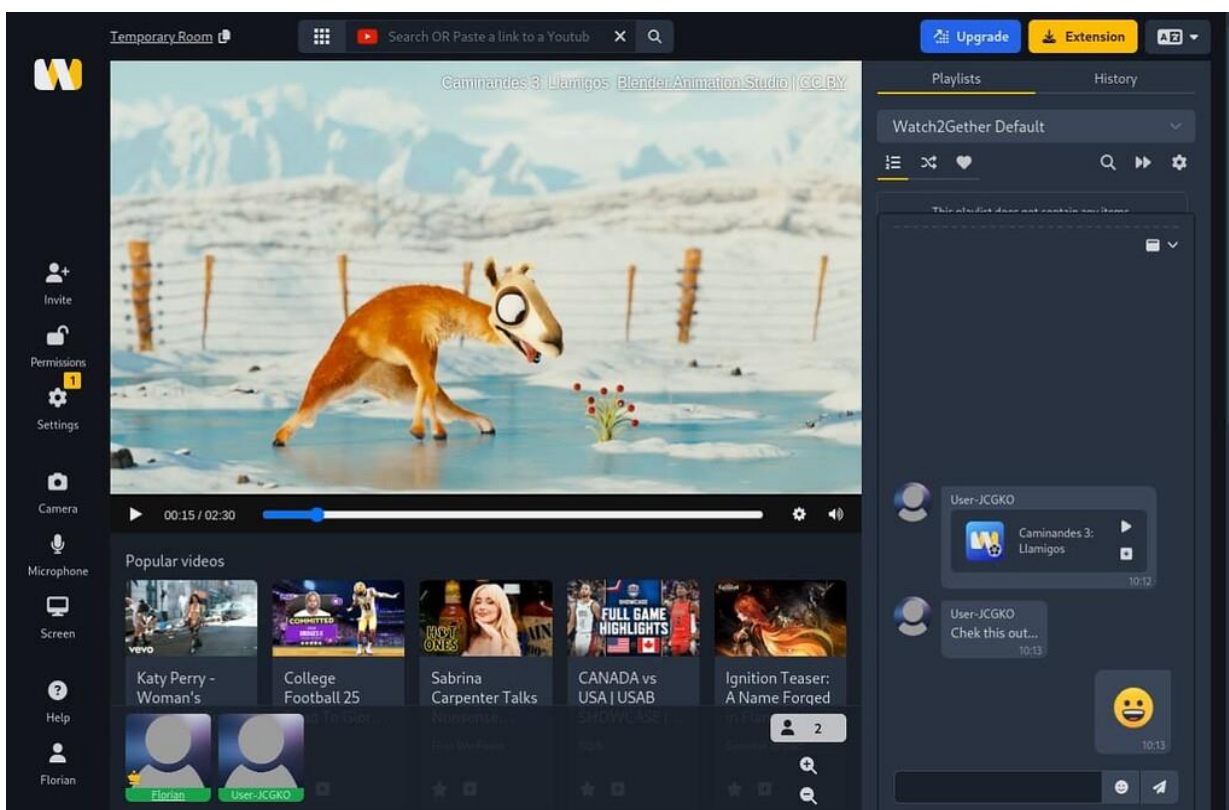


Рисунок 1.5 - Интерфейс Watch2Gether [10]

Аналіз існуючих рішень показує відсутність універсального інструменту. Більшість інструментів приносять у жертву якість зображення, або занадто обмежені списком підтримуваних сайтів. Саме необхідність покращення способу дивитися відеоконтент, щоб працював поверх будь-якого HTML5-плеєра безпосередньо на сторінці першоджерела, обумовлює необхідність даної розробки [11].

1.2 Порівняльний аналіз розробки та аналогів

Для обґрунтування доцільності розробки було проведено порівняльний аналіз найбільш розповсюджених інструментів для спільного перегляду та запропонованого програмного забезпечення (табл. 1.1). Критерії порівняння охоплюють технічні аспекти передачі даних, ефективність використання мережевих ресурсів та універсальність застосування в сучасних браузерних екосистемах [12].

Таблиця 1.1 - Порівняння засобів організації спільного перегляду медіа-контенту

Критерій	Discord (Screen Share)	Teleparty	Watch2Gether	Запропонова на розробка
1	2	3	4	5
Метод синхронізації	Потокова трансляція відеосигналу	Синхроні- зація станів плеєра	Вбудований програвач / API	Синхроні- зація станів через розширення
Якість зображення	Знижена	Максимальна	Залежить від джерела	Максимальна
Універсаль- ність (підтримка сайтів)	Висока (будь-яке вікно)	Низька (лише популярні стрімінги)	Середня (лише підтримувані сервіси)	Висока (будь-який HTML5- плеєр)

Кінець таблиці 1.1

1	2	3	4	5
Навантаження на систему та мережу	Високе (кодування відео)	Низьке	Середнє	Мінімальне (лише метадані)
Середовище перегляду	Окремий застосунок	Сайт стрімінгу	Сторонній вебсайт	Сайт першоджерела
Вимоги до учасників	Наявність застосунку	Однакові підписки	Відсутні	Браузерне розширення

Порівняння продемонструвало, що існуючі аналоги не забезпечують одночасно високу якість контенту та універсальність джерел. Наприклад, популярний метод трансляції екрана призводить до значної деградації зображення через особливості кодування відео в реальному часі. Тоді як спеціалізовані розширення жорстко обмежують користувача списком платних сервісів, що робить їх непридатними для перегляду контенту з довільних сайтів. Використання сторонніх ресурсів для агрегації відео часто обмежує функціонал оригінальних плеєрів та потребує від користувачів зміни звичного інтерфейсу [13].

Запропонована розробка базується на принципі синхронізації лише керуючих команд поверх будь-якого стандартного HTML5-плеєра, що якісно відрізняється від аналогів. Це дозволяє:

- отримувати відео безпосередньо з першоджерела, зберігаючи оригінальну роздільну здатність (4K/Full HD) без додаткового стиснення;

- працювати практично на будь-якому вебсайті, що використовує стандартні веб-плеєри, без необхідності специфічної інтеграції з боку власників контенту;

- забезпечити мінімальну затримку синхронізації навіть при низькій швидкості інтернет-з'єднання, оскільки об'єм даних, що передаються для синхронізації (статус "Play/Pause", часова мітка), є мізерним.

Таким чином, розробка поєднує універсальність засобів трансляції екрана з досконалістю спеціалізованих синхронізаторів станів, що базуються на сучасних протоколах передачі даних у реальному часі [14].

1.3 Висновки за розділом

У межах першого розділу було визначено основні орієнтири дослідження та поставлено завдання для подальшої розробки. З основних принципів - це базування проєкту на актуальних запитах сучасного ринку медіа-послуг та потребах користувачів у поглибленні соціальної взаємодії [15]. Проведене дослідження підтвердило, що трансформація цифрового дозвілля вимагає впровадження нових технологічних підходів до координації віддалених груп. До ключових етапів роботи віднесено:

- аналіз існуючого програмного забезпечення для синхронізації спільного перегляду медіа-контенту;
- визначення архітектурних рішень та засобів проєктування системи;
- формулювання технічних та функціональних вимог до ПЗ;
- програмну реалізацію системи.

Виконання завдання з аналізу існуючих рішень дозволило виявити суттєву прогалину на ринку програмних продуктів. Сучасні сервіси використовують метод трансляції потокового відео, що призводить до значної деградації якості зображення та високого навантаження на мережу. Або мають закриту екосистему, що обмежена лише кількома стрімінговими

майданчиками. Окрім суто технічних обмежень, такі продукти часто вимагають наявності ідентичних платних підписок у всіх учасників сесії, що створює високий економічний поріг входу для пересічного користувача.

Аналіз показав, що наразі відсутній універсальний інструмент, який би дозволяв користувачам об'єднуватися безпосередньо на сторінках першоджерел із використанням сучасних архітектурних стандартів безпеки та продуктивності [16].

Саме на ліквідацію цих технічних недоліків та покращення комфорту користувачів під час перегляду медіа-контенту у реальному часі спрямована розробка, опис якої буде представлено у наступних розділах роботи.

2 РОЗРОБКА АРХІТЕКТУРИ ПРОГРАМИ

2.1 Вибір засобів проєктування

Програмний комплекс SyncPlayer планується до розробки на основі клієнт-серверної архітектури. Клієнтська сторона буде розширенням для браузера. Вона буде забезпечувати автономний аналіз DOM-структури для ідентифікації плеєра, локальну фільтрацію подій відтворення та програмну компенсацію десинхронізації між учасниками. А віддалений сервер буде у ролі координатора сесій. Мовою програмування для обох компонентів було обрано JavaScript, що є найбільш звичним та ефективним інструментом для створення браузерних розширень та серверних систем у реальному часі.

Для реалізації серверної логіки було проведено порівняльну роботу між двома підходами до реалізації двостороннього зв'язку. Бібліотекою Socket.IO та низькорівневою бібліотекою ws. На основі аналізу їхніх характеристик (табл. 2.1) було обрано ws, оскільки вона забезпечує мінімальний обсяг службових даних (overhead) та найвищу швидкість передачі "сирих" повідомлень, що є критичним для точної синхронізації відеопотоку.

На основі аналізу було обрано бібліотеку ws як інструмент, що забезпечує найменшу затримку сигналу, що є критично важливим для синхронізації відео.

Клієнтську частину (браузерне розширення) вирішено реалізовувати на стандарті Manifest V3. Це дозволяє використовувати Service Workers для фонові обробки даних. Це гарантує раціональне використання ресурсів системи. Для швидкої взаємодії з локальними даними сесії обрано асинхронне сховище chrome.storage.local, що підтримується розширеннями.

Серверну частину логіки буде реалізовано на базі Node.js, що дозволить підтримувати тисячі одночасних WebSocket-з'єднань завдяки подієвій моделі введення-виведення. Для управління серверними процесами

обрано модуль `http`, що в поєднанні з `WebSockets` дозволить уникнути зайвих рівнів абстракції.

Таблиця 2.1 - Вибір засобів для розробки програми

Критерій	Socket.io	Бібліотека ws
Продуктивність	Додатковий рівень абстракції сповільнює обробку пакетів	Максимальна швидкість обробки подій у Node.js
Обсяг даних	Великі заголовки пакетів (понад 100 байт на пакет)	Мінімальний overhead (важливо для синхронізації)
Складність реалізації	Вимагає використання власного клієнтського SDK	Працює з нативним браузерним WebSocket API
Сумісність з MV3	Потребує складних налаштувань для Service Worker	Пряма сумісність із сучасними стандартами розширень

Для автентифікації та розмежування прав доступу буде застосовуватися механізм генерації унікальних анонімних ідентифікаторів.

Як систему керування базами даних для зберігання інформації про кімнати, їхні налаштування обрано PostgreSQL. Її вибір обґрунтований надійністю. Водночас, для забезпечення максимальної швидкості синхронізації станів відтворення (play/pause, timestamp) використовується гібридний підхід: ці дані оброблятимуться в оперативній пам'яті сервера, що дозволяє уникнути затримок, характерних для дискових операцій запису.

2.2 Сценарії використання програми

Користувачі системи не мають жорсткого поділу на ролі, проте при створенні кімнати один із них автоматично отримує статус власника з розширеними правами керування сесією.

Основні сценарії використання (Main Flow):

- генерація унікального ID користувача: при першому запуску розширення автоматично створює ідентифікатор сесії `myUserId`;
- створення кімнати: ініціатор генерує нову сесію з випадковим 3-значним кодом та унікальним набором емодзі для візуальної назви;
- встановлення зв'язку з відео-плеєром: користувач через контекстне меню обирає конкретне відео на сторінці для синхронізації;
- трансляція стану плеєра: при натисканні «Play» або «Pause» розширення перехоплює подію та надсилає об'єкт `playerState` (статус, секунда, таймстемп) на сервер;
- автоматична синхронізація: сервер розсилає отриманий стан усім учасникам кімнати (broadcast), крім відправника;
- запит актуального стану (Get State): новий учасник при вході автоматично запитує у сервера поточну позицію відео, щоб вирівняти свій плеєр з групою;
- керування налаштуванням кімнати: власник кімнати може увімкнути або вимкнути можливість керування плеєром для інших учасників.

Альтернативні сценарії:

- спроба входу в неіснуючу кімнату: система видає повідомлення про помилку "Room not found";
- синхронізація без активної кімнати: якщо користувач намагається підключити відео, не увійшовши в кімнату, система пропонує спочатку створити сесію або приєднатися до існуючої;
- автоматичне перепідключення: при оновленні сторінки розширення автоматично відновлює зв'язок із сервером, використовуючи

збережений у сховищі syncedTabId;

– видалення порожньої кімнати: якщо останній користувач залишає кімнату, сервер автоматично очищує пам'ять від цієї кімнати.

Схема, що відображає прецеденти програми для обох ролей користувачів, представлена на рис. 2.1.

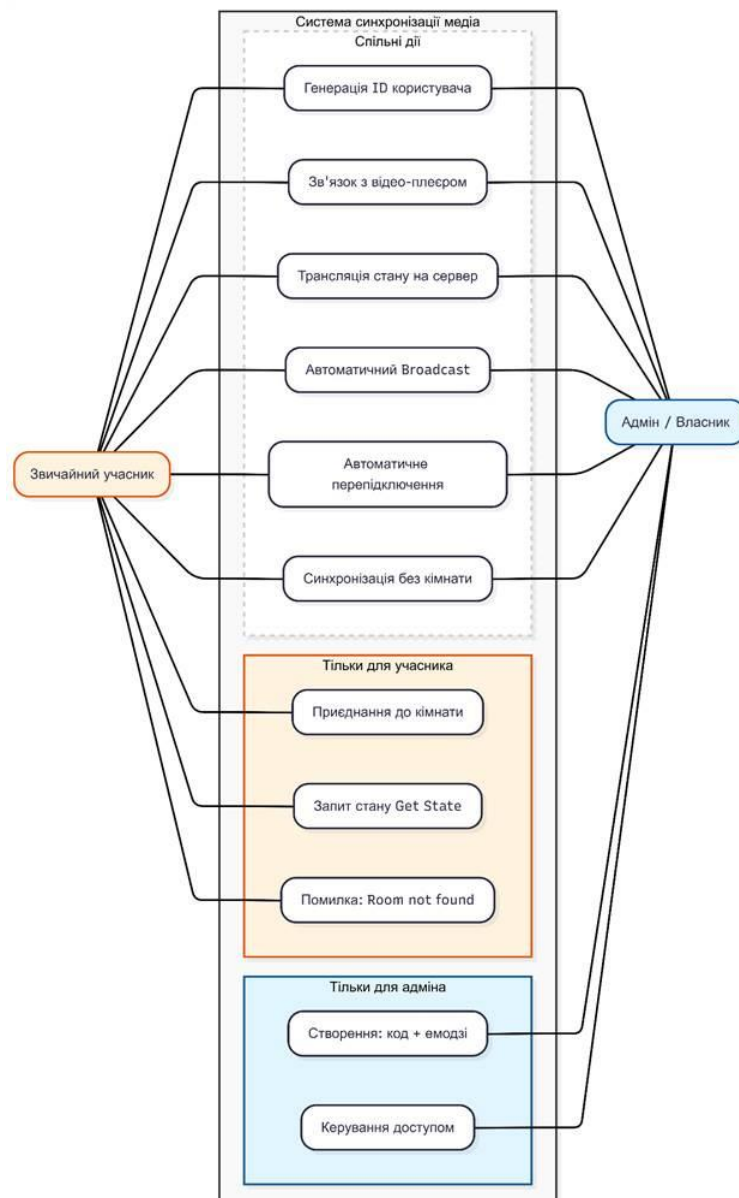


Рисунок 2.1 - Діаграма прецедентів програми

2.3 Проектування архітектури системи

Проектування архітектури програмного комплексу базується на принципах модульності та чіткого розподілу відповідальності між компонентами. Обрана структура зумовлена специфікою роботи сучасних веб браузерів та обмеженнями безпеки, які накладаються на розширення стандартом Manifest V3

Логічно, система поділяється на два глобальні середовища: браузерне (клієнтське) та серверне (рис. 2.2). Така децентралізація дозволяє кожному учаснику сесії завантажувати медіаконтент безпосередньо з першоджерела, тоді як сервер лише координує часові стани відтворення.

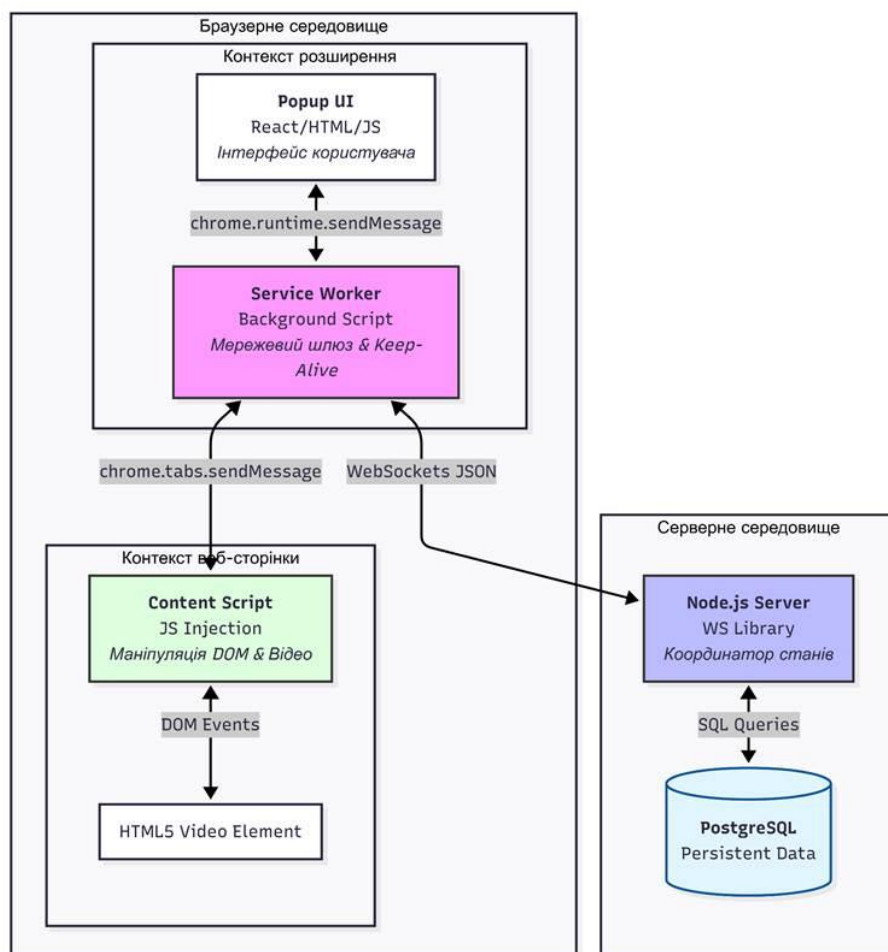


Рисунок 2.2 – Діаграма компонентів системи та логіка їх взаємодії

2.4 Прототипування користувацького інтерфейсу

Прототипування графічного інтерфейсу системи базується на принципах функціонального мінімалізму, де кожен елемент має чітке призначення у процесі встановлення та підтримки синхронної сесії.

Взаємодія з розширенням відбувається безпосередньо під час перегляду медіа-контенту. Інтерфейс спроектований, щоб мінімізувати кількість кліків та не перевантажувати користувача.

Основна увага приділена проектуванню декількох ключових станів: головної панелі, панелі активної кімнати, засобів адміністрування та контекстних інструментів інтеграції з веб-сторінкою (рис. 2.3).



Рисунок 2.3 – Прототипи графічного інтерфейсу та основних вікон користувача

Першим екраном, з яким буде взаємодіяти користувач, є головна панель розширення. Вона відповідає за вхід у систему. Тут реалізовано два

базові сценарії: створення нової віртуальної кімнати та приєднання до існуючої сесії. Для спрощення користувацького шляху поле для введення коду автоматично фокусується при відкритті вікна. Також кнопка створення сесії винесена як окремий пріоритетний елемент. Таке проектування дозволяє почати спільний перегляд за лічені секунди, що є критичним для утримання уваги аудиторії.

Після успішного входу користувач потрапляє на панель керування кімнатою. Центром уваги тут є унікальний тризначний код сесії та назва у вигляді емоджі. Вони будуть спрощувати вербальну передачу параметрів підключення. Важливою частиною цього інтерфейсу є блок статусів синхронізації. Він містить динамічні індикатори стану з'єднання із сервером та статусу прив'язки до відео-плеєра. Проектування цих статусів у вигляді кольорових бейджів коли зелений для стабільного зв'язку та червоний для помилки. Це забезпечує миттєвий зворотний зв'язок, дозволяючи користувачеві швидко діагностувати технічні проблеми без необхідності відкривати консоль розробника.

Для користувачів із роллю власника кімнати передбачено розширений функціонал у вигляді адмін-панелі. Проектування цього блоку включає перемикач режиму спільним керуванням контентом під назвою «Демократія». Він визначає, чи можуть інші учасники керувати станом відтворення. Це технічне рішення дозволяє використовувати систему не лише для розваг, а й для освітніх цілей. Наприклад, коли модератор (лектор) повинен мати повний контроль над плеєром аудиторії, а слухачі ні. Також адмін-панель містить кнопку примусової десинхронізації відео, що дозволяє швидко змінити контент для синхронізації.

Окремим аспектом інтерфейсу є механізм «під'єднання» плеєра. Він буде реалізований через системне контекстне меню браузера. Проектування цього вузла передбачає виклик специфічного пункту меню «Sync this video», який ініціює передачу координат DOM-елемента до Content Script для подальшої обробки розширенням. Це дозволяє зробити процес прив'язки

довільного HTML5-відео максимально нативним та зручним, оскільки він не потребує від користувача знань про структуру веб-сторінки.

2.5 Структура бази даних

Для забезпечення надійності, масштабованості та мінімальної затримки при передачі команд у реальному часі, архітектура даних проєкту базується на гібридному підході. Традиційні дискові операції запису в реляційних БД не можуть забезпечити необхідну швидкість для синхронізації відео (де оновлення станів відбуваються кілька разів на секунду). Тому було прийнято рішення розділити структуру на персистентне сховище та оперативне сховище.

Логічна модель системи представлена кількома ключовими сутностями. Це дозволяє системі точно координувати дії користувачів.

Таблиця БД Rooms визначає основні метадані віртуального простору, які зберігаються персистентно в PostgreSQL для забезпечення стабільності сесій. Вона містить такі стовпці:

- `id` - унікальний текстовий ідентифікатор кімнати, що складається з 3-х цифр (Primary Key). Такий формат обрано для максимальної зручності передачі коду між користувачами через месенджери або голосом, що значно спрощує поріг входу в сесію;
- `name` - назва кімнати у вигляді випадкової комбінації емодзі. Цей параметр впроваджено для покращення користувацького досвіду (UX), що дозволяє глядачам значно легше ідентифікувати свою кімнату візуально серед інших активних вікон розширення;
- `owner_id` - ідентифікатор користувача, який створив кімнату (зовнішній ключ до таблиці Users). Наявність цього поля необхідна для суворого розмежування прав доступу, оскільки лише власник має право на управління критичними налаштуваннями кімнати та доступ до адмін-панелі;
- `video_url` - пряме посилання на сторінку з медіа-контентом, де

відбувається синхронізація. Збереження цього значення дозволяє новим учасникам переходити на відповідний ресурс при потребі;

- `created_at` - дата та час створення запису. Параметр використовується для системного моніторингу та реалізації майбутнього очищення бази даних від застарілих або покинутих сесій.

Таблиця БД `Room_Settings` визначає конфігурацію взаємодії в межах кімнати:

- `room_id` - ідентифікатор кімнати (Primary/Foreign Key). Забезпечує зв'язок налаштувань із конкретною сесією;

- `is_democracy` - логічне поле (boolean). Налаштування прав доступу, яке дозволяє обмежити керування плеєром лише для організатора або надати таку можливість кожному учаснику. Це критично важливо для різних сценаріїв використання - від навчальних вебінарів до дружніх кінопереглядів;

- `auto_sync_url` - прапорець автоматичної синхронізації сторінок. Забезпечує можливість примусового перенаправлення всіх учасників на нове відео, якщо власник кімнати вирішив змінити контент у процесі перегляду.

На рис. 2.4 представлена ER-модель, що відображає структуру таблиць у PostgreSQL та їхні взаємозв'язки, що гарантує цілісність даних на рівні бази.

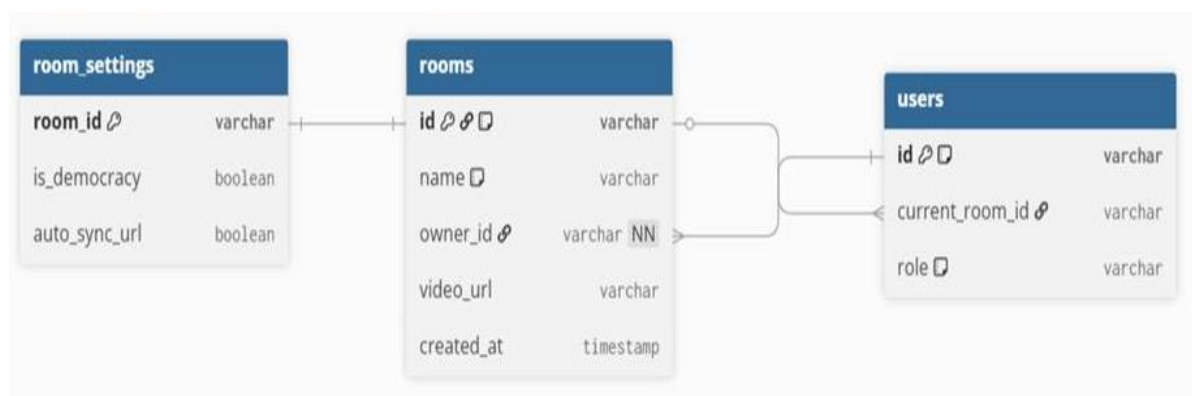


Рисунок 2.4 - ER-модель структура таблиць у PostgreSQL

Таблиця БД Users описує учасників системи та їхній поточний стан:

- `id` - унікальний анонімний ідентифікатор, що генерується розширенням (Primary Key). Це дозволяє ідентифікувати користувача без збору персональних даних, що повністю відповідає сучасним вимогам цифрової приватності та безпеки;
- `current_room_id` - ідентифікатор кімнати, в якій перебуває користувач (зовнішній ключ до таблиці Rooms). Цей параметр забезпечує логічний зв'язок типу «один-до-багатьох», дозволяючи серверу чітко визначати пул сокет-з'єднань для розсилки команд у межах конкретної сесії;
- `role` - роль користувача в межах кімнати ('OWNER' або 'PARTICIPANT'). Параметр необхідний для валідації запитів на зміну стану плеєра, що є основою функціонування режимів доступу.

Об'єкт `Player_State` зберігається виключно в оперативній пам'яті сервера. Це дозволяє досягти миттєвої реакції системи на дії користувачів.

Структура об'єкта включає:

- `room_id` - посилання на ідентифікатор кімнати. Забезпечує миттєвий пошук актуального стану відтворення для всіх активних з'єднань;
- `player_status` - мітка поточного стану плеєра. Параметр безпосередньо вказує клієнтським скриптам, яку команду необхідно ініціювати в локальному плеєрі кожного учасника;
- `second` - точна позиція відео в секундах. Використання числа з плаваючою комою забезпечує синхронізацію відео з точністю до кадру;
- `timestamp` - серверний час у мілісекундах. Параметр є основою алгоритму компенсації мережових затримок, що дозволяє серверу динамічно розраховувати «дрейф» часу та коригувати позицію відео для учасників із нестабільним інтернет-з'єднанням.

На рис. 2.5 продемонстровано архітектурну схему взаємодії між персистентним сховищем PostgreSQL та оперативною пам'яттю сервера. Зв'язок між ними реалізований через координаційний модуль Node.js, який

при створенні кімнати ініціалізує запис у БД, а під час активного перегляду оперує виключно даними в пам'яті для забезпечення найвищої швидкодії.

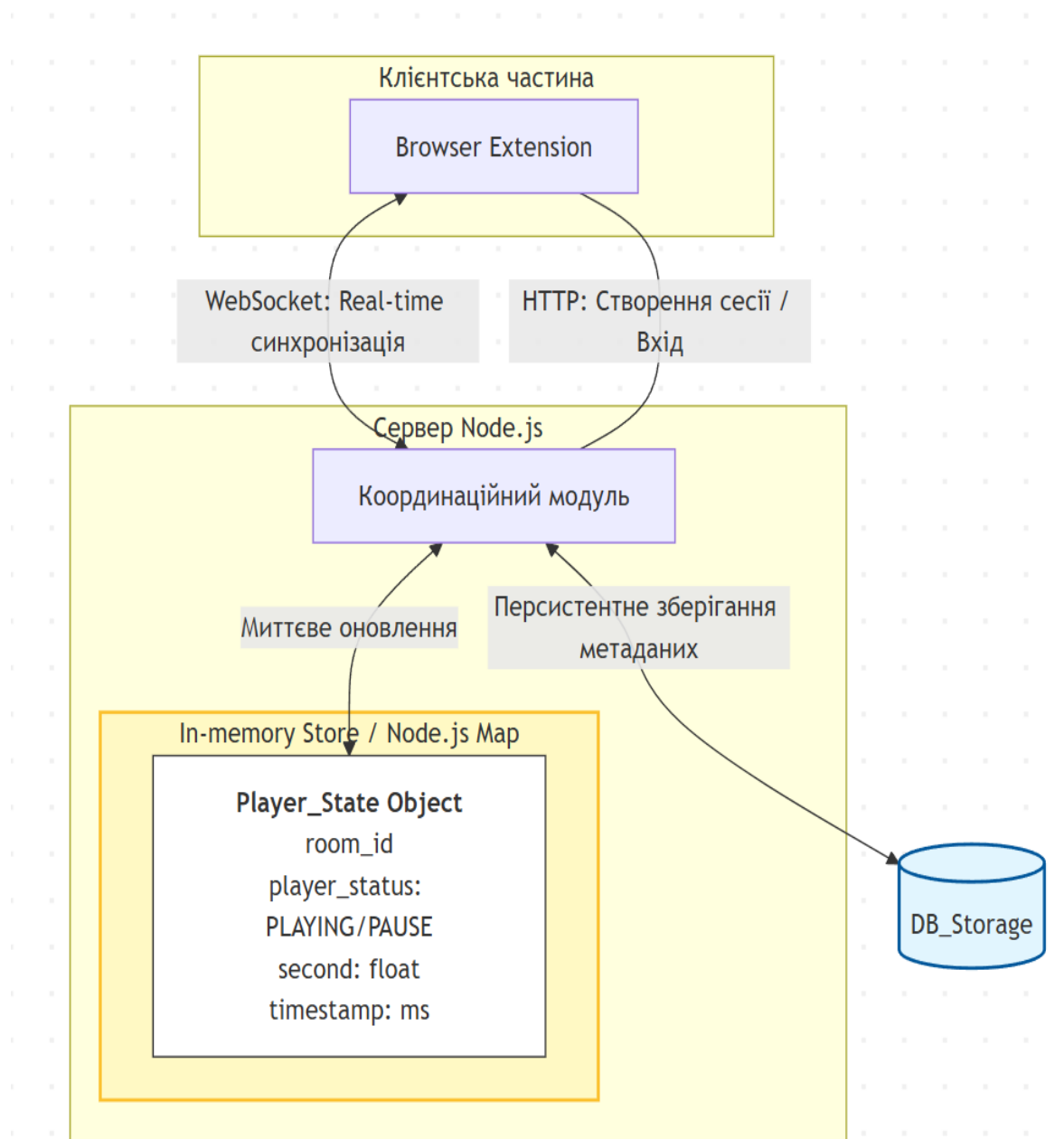


Рисунок 2.5 - архітектурна схема взаємодії PostgreSQL та оперативної пам'яті сервера

Представлена гібридна структура дозволяє системі поєднувати надійність реляційної бази даних із продуктивністю рішеннями використання операційної пам'яті. Це є ключовим фактором для стабільної синхронізації медіа-контенту в реальному часі.

2.6 Висновки за розділом 2

У другому розділі було проведено комплексне проєктування архітектури системи та обґрунтовано вибір технологічного стека для реалізації програмного комплексу. На основі проведеного аналізу визначено, що поєднання середовища Node.js із протоколом WebSockets (бібліотека ws) є оптимальним рішенням для забезпечення мінімальної затримки при передачі керуючих команд у реальному часі.

Особливу увагу приділено розробці гібридної моделі управління даними. Було обґрунтовано використання реляційної БД PostgreSQL для персистентного зберігання структурної інформації та сховищ в операційній пам'яті сервера для опрацювання динамічних станів плеєра, що дозволяє уникнути затримок при дискових операціях.

Описано детальні сценарії взаємодії користувачів. Спроектowana структура дозволяє системі підтримувати високу цілісність інформації та актуальний стан медіа-плеєрів для великої кількості незалежних груп користувачів. Обрана архітектура забезпечує масштабованість системи та створює фундамент для програмної реалізації інструменту, що гарантує стабільну синхронізацію контенту без деградації його якості.

3 РЕАЛІЗАЦІЯ ПРОГРАМИ

3.1 Визначення порядку роботи програми

Функціональна схема розробленого програмного забезпечення базується на ієрархічній взаємодії компонентів браузерного розширення та серверного вузла. Порядок роботи системи побудований за принципом строгої послідовності етапів від ідентифікації клієнта до встановлення реал-тайм каналу зв'язку. Такий підхід обрано для забезпечення цілісності даних, тому що система не може ініціювати синхронізацію медіа-контенту без попереднього встановлення параметрів сесії та перевірки прав доступу користувача.

Логіка роботи програми розділена на два ключові цикли: життєвий цикл сесії (управління кімнатою) та подієвий цикл синхронізації (Event Loop плеєра).

3.1.1 Ініціалізація та управління сесією

Першочерговим етапом є підготовка робочого середовища розширення. Це необхідно для того, щоб сервер міг коректно ідентифікувати кожного учасника та розмежовувати права «власника» та «учасника». При запуску браузера фоновий скрипт `service_worker.js` перевіряє наявність унікального `myUserId` у локальному сховищі `chrome.storage.local` (рис. 3.1). Якщо ідентифікатор відсутній, система генерує його автоматично, що дозволяє уникнути складної процедури реєстрації, зберігаючи при цьому можливість персоналізації прав та збереження сесії при перезавантаженні сторінок.

Після ідентифікації користувач взаємодіє з інтерфейсом `room.html` для створення або входу в кімнату. Сервер виконує роль координатора, який не лише генерує код доступу, а й ініціалізує логічні структури в пам'яті для

зберігання майбутніх станів плеєра. Процес управління доступом та створення сесії представлений на алгоритмі нижче.

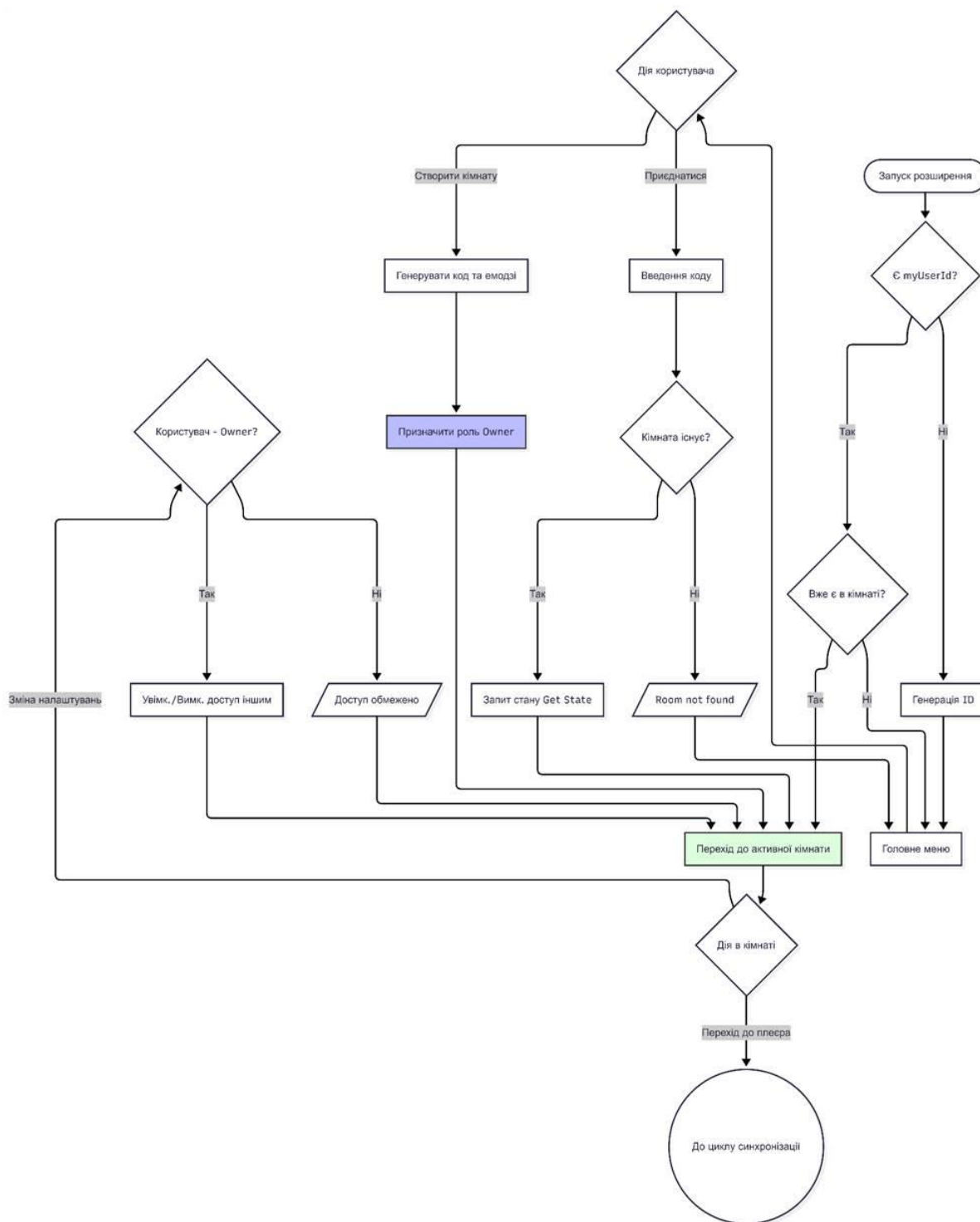


Рисунок 3.1 - Алгоритм управління сесіями та правами доступу користувачів

Цей етап є фундаментом системи, оскільки він визначає режим взаємодії, що безпосередньо впливає на подальшу роботу алгоритмів синхронізації та обробку вхідних команд від різних учасників.

3.1.2 Вибір медіа-об'єкта та підключення слухачів

Порядок дій, за якого користувач вказує об'єкт синхронізації лише після входу в сесію, обумовлений вимогами безпеки та оптимізації продуктивності. Такий підхід дозволяє уникнути постійного фонового сканування всіх відкритих вкладок на наявність медіа елементів, зберігаючи ресурси процесора. Процес активації базується на використанні системного контекстного меню, де фоновий скрипт виступає координатором між інтерфейсом браузера та змістом сторінки.

При виборі команди синхронізації фоновий процес ініціює передачу повідомлення до скрипту вмісту, який за допомогою координат останньої взаємодії ідентифікує конкретний тег відео в DOM-дереві. Для забезпечення якісного користувацького досвіду до знайденого плеєра додається візуальна індикація у вигляді кольорової рамки. Це підтверджує успішне захоплення об'єкта системою та гарантує, що подальші команди відтворення будуть коректно транслюватися іншим учасникам кімнати.

3.1.3 Циклічна синхронізація станів (Sync Loop)

Кінцевим і найбільш динамічним етапом є безпосередня синхронізація. Вона працює за принципом перехоплення подій. Система реагує на локальні дії користувача або на команди, що надходять від сервера. Важливою особливістю є перевірка прав у реальному часі: якщо в кімнаті вимкнено режим «Демократії», система блокує спроби звичайних учасників змінити стан плеєра. Детальна логіка обробки подій та взаємодії між клієнтами представлена на схемі циклу синхронізації (рис. 3.2).

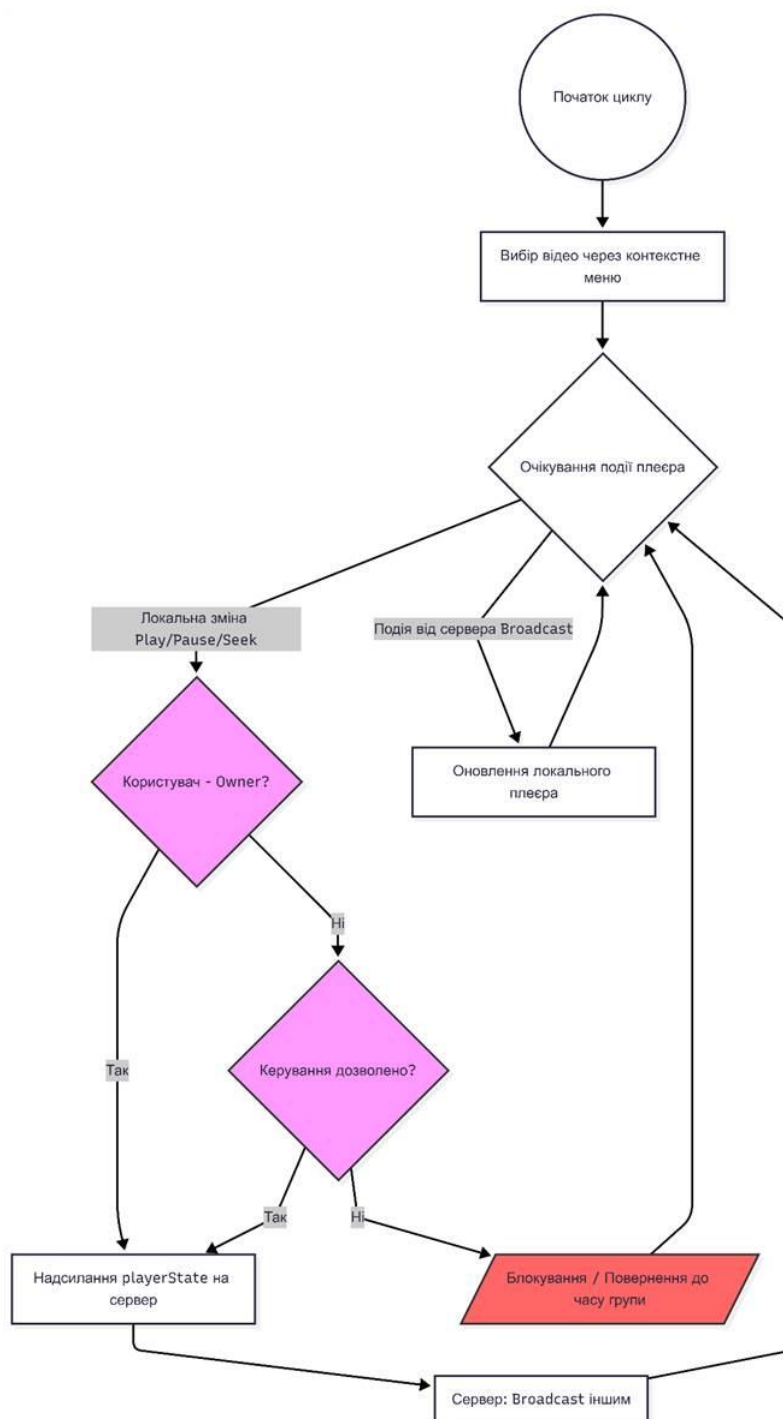


Рисунок 3.2 - Алгоритм синхронізації стану відео-плеєра в реальному часі

Така послідовність роботи забезпечує повну автономність кожного учасника. Тому що відеопотік завантажується безпосередньо з першоджерела, а сервер оперує лише короткими JSON-пакетами з координатами часу та статусом відтворення. Це дозволяє досягти мінімального навантаження на сервер та стабільного відтворення навіть при великій кількості одночасних підключень, оскільки розширення, не

втручаючись у сам процес передачі важких медіа-даних, коригує положення повзунка відтворення.

3.2 Результати реалізації програмних одиниць застосунку

3.2.1 Реалізація основних сценаріїв використання

Для ідентифікації кожного учасника в системі реального часу необхідно розрізняти власника кімнати та звичайних глядачів без використання класичної форми реєстрації. Оскільки проєкт орієнтований на швидку взаємодію, розширення автоматично генерує анонімний ідентифікатор при самому першому запуску. Це дозволяє системі впізнавати користувача під час повторних підключень, зберігаючи його права доступу та історію перебування в сесіях навіть після повного закриття браузера.

Технічна логіка реалізації функціонала в лістингу 3.1 базується на використанні асинхронного API `chrome.storage.local`, яке призначене для збереження даних в ізольованому сховищі розширення.

Лістинг 3.1 - Функція генерації унікального ID користувача

```
function getUserId(cb) {
  chrome.storage.local.get('myUserId', (res) => {
    if (res.myUserId) {
      cb(res.myUserId);
    } else {
      const id = 'user_' + Math.floor(Math.random() *
100000);
      chrome.storage.local.set({ myUserId: id });
      cb(id);
    }
  });
}
```

Оскільки операції з пам'яттю браузера є неблокуючими, функція використовує механізм зворотного виклику для передачі результату в параметр `cb`, що гарантує коректну роботу з даними лише після завершення їх зчитування.

Метод `get` ініціює пошук існуючого ключа, і в разі його відсутності програма переходить до фази генерації нової мітки. Використання комбінації статичного префікса та математичної рандомізації через `Math.random` дозволяє створити унікальну строкову мітку з мінімальною ймовірністю колізій.

Після генерації метод `set` примусово записує значення у сховище, що забезпечує стабільність сесії користувача. Такий підхід робить ідентифікатор постійним, що є критичним для валідації прав власності на кімнати на стороні сервера та дозволяє уникнути потреби у складних системах авторизації з паролями.

Для того, щоб користувачі могли швидко обмінюватися даними для входу, система використовує короткі цифрові коди замість складних технічних адрес. Процес створення віртуальної кімнати автоматично підбирає вільний номер та генерує унікальну візуальну назву за допомогою емоджі, що дозволяє легко ідентифікувати сесію та почати спільний перегляд за лічені секунди без ризику потрапити в чужу групу.

Логіка роботи функцій у лістингу 3.2 реалізована на стороні сервера `Node.js` із використанням колекції `Map` для оперативного зберігання сесій.

Лістинг 3.2 - Реалізація генерації унікального коду кімнати на сервері

```
function generateRoomId() {
  let id;
  do {
    id = Math.floor(Math.random() * 900 + 100).toString();
  } while (rooms.has(id));
  return id;
}

function getRandomEmojis() {
  const EMOJIS = ['popcorn ', ' movie ', ' pizza ', '
rocket ', ' cat '];
  return Array(3).fill().map(() =>
    EMOJIS[Math.floor(Math.random() * EMOJIS.length)]).join('');
}
```

Метод `generateRoomId` виконує генерацію випадкового тризначного числа в заданому діапазоні. Ключовим аспектом є використання циклу `do...while`, який за допомогою методу `rooms.has(id)` виконує перевірку на наявність колізій. Тобто, якщо ідентифікатор уже зайнятий іншою сесією, алгоритм ініціює повторну генерацію до моменту знаходження вільного коду. Одночасно з цим функція `getRandomEmojis` обробляє масив статичних констант, формуючи за допомогою методів `fill` та `map` унікальний рядок із трьох випадкових символів.

Таке поєднання цифрових та візуальних ідентифікаторів дозволяє серверу ефективно адресувати пакети даних конкретним кімнатам, забезпечуючи при цьому високу швидкість пошуку об'єкта в пам'яті. Після успішного завершення цих процедур сервер фіксує ідентифікатор власника (Owner) та ініціалізує початковий об'єкт стану, що робить кімнату доступною для підключення інших учасників.

Для того, щоб розширення могло відстежувати дії користувача в реальному часі, воно має підписатися на всі зміни в обраному відео-плеєрі. Це дозволяє програмі автоматично реагувати на кожне натискання кнопки паузи або зміну позиції відтворення, перетворюючи звичайний перегляд на інтерактивну сесію, де будь-яка дія на одному пристрої стає відомою всій системі.

Технічна реалізація методу `attachListeners` у лістингу 3.3 забезпечує перехід медіа-елемента під повне управління розширення.

Лістинг 3.3 - Метод ініціалізації спостереження за медіа-елементом

```
function attachListeners(video) {
  activeVideo = video;
  activeVideo.style.border = '4px solid #4CAF50';

  const events = ['play', 'pause', 'seeking'];
  events.forEach(event => {
    activeVideo.addEventListener(event, onPlayerEvent);
  });

  safeSendMessage({ action: 'video-attached' });}
```

Спочатку функція зберігає пряме посилання на DOM-об'єкт у змінну `activeVideo`, що дозволяє системі маніпулювати конкретним плеєром без необхідності повторного сканування сторінки.

Встановлення стилю `border` виконує роль візуального зворотного зв'язку, створюючи навколо відео кольорову рамку для підтвердження успішного підключення.

Основна логіка полягає у масовій реєстрації обробників подій через метод `addEventListener` у межах циклу `forEach`. Охоплення подій `play`, `pause` та `seeking` дозволяє системі фіксувати будь-які маніпуляції користувача з часовою шкалою.

Завершальний виклик `safeSendMessage` передає сигнал про успішну прив'язку об'єкта до фонового процесу, що ініціює перехід всієї архітектури синхронізації в активний стан.

Коли один із учасників змінює стан відео, ця інформація повинна миттєво дійти до кожного іншого глядача в кімнаті. Сервер виконує роль центрального диспетчера, який пересилає отримані дані всім підключеним пристроям. При цьому вкрай важливо ігнорувати самого відправника, щоб його плеєр не отримав власну ж команду повторно, що могло б призвести до помилкових стрибків відео або нескінченного циклу оновлень.

Робота методу `broadcastToRoom` у лістингу 3.4 демонструє механізм ретрансляції повідомлень у реальному часі через WebSocket-з'єднання.

Лістинг 3.4 - Метод розсилки стану учасникам сесії

```
function broadcastToRoom(roomId, data, excludeUserId) {
  const room = rooms.get(roomId);
  if (!room) return;

  const msg = JSON.stringify(data);
  room.users.forEach((client, userId) => {
    if (userId !== excludeUserId && client.readyState ===
WebSocket.OPEN) {
      client.send(msg);
    }
  });
}
```

Сервер спочатку витягує об'єкт кімнати з глобальної колекції за її унікальним кодом, виконуючи перевірку на існування сесії для запобігання помилкам адресації. Перед відправкою дані конвертуються у текстовий формат JSON, що забезпечує високу швидкість передачі та сумісність між різними браузерами.

Цикл `forEach` проходить по всіх активних сокетах учасників кімнати, де застосовується критично важлива умова фільтрації через параметр `excludeUserId`. Ця перевірка гарантує, що пакет даних отримають лише «слухачі», виключаючи ініціатора події, що запобігає виникненню рекурсивних команд.

Додаткова валідація стану сокета `WebSocket.OPEN` дозволяє серверу уникати спроб відправки даних на розірвані або нестабільні з'єднання, що підвищує загальну надійність системи синхронізації.

Приєднавшись до перегляду, новий учасник повинен миттєво підлаштувати свій плеєр під поточний прогрес групи, щоб уникнути розсинхронізації. Оскільки відео вже може тривати певний час, розширення автоматично запитує актуальні координати відтворення та статус паузи у сервера відразу після підключення. Це дозволяє новому глядачеві «наздогнати» інших учасників без необхідності вручну шукати потрібний момент на часовій шкалі, забезпечуючи цілісність групового досвіду з першої секунди перебування в кімнаті.

Технічне виконання процедури в лістингу 3.5 базується на отриманні метаданих сесії з локального сховища браузера через інтерфейс `chrome.storage.local`.

Перед відправкою запиту функція виконує критичну перевірку стану `WebSocket`-з'єднання за допомогою властивості `readyState`, що дозволяє уникнути помилок при спробі передачі даних через закритий або нестабільний канал зв'язку.

Сама команда `get-state` пакується в об'єкт JSON, який містить ідентифікатори кімнати та користувача, що дозволяє серверу однозначно визначити, з якої саме сесії потрібно витягнути актуальні дані.

Лістинг 3.5 - Процедура запиту поточного стану кімнати

```
function requestServerState() {
    chrome.storage.local.get(['roomId', 'myUserId'], (res) =>
    {
        if (res.roomId && socket?.readyState ===
WebSocket.OPEN) {
            socket.send(JSON.stringify({
                action: 'get-state',
                roomId: res.roomId,
                userId: res.myUserId
            }));
        }
    });
}
```

Такий підхід ініціює ланцюжок синхронізації, де сервер у відповідь надсилає останній збережений стан плеєра, який згодом примусово застосовується до локального відеоелемента нового учасника.

Для запобігання хаотичному управлінню плеєром у великих групах система потребує чіткого розмежування повноважень між ініціатором сесії та її гостями. Сервер виступає в ролі арбітра, який перевіряє правомірність кожної команди на зміну стану відео перед тим, як розповсюдити її серед інших учасників. Це дозволяє використовувати систему не лише для дружніх переглядів, а й для презентацій чи лекцій, де модератор повинен мати повний контроль над процесом, ігноруючи випадкові або навмисні спроби інших користувачів втрутитися в хід відтворення.

Серверний алгоритм валідації в лістингу 3.6 виступає в ролі інтелектуального фільтра для вхідного трафіку подій.

При отриманні повідомлення `video-update` програма спочатку звертається до колекції активних кімнат для порівняння ідентифікатора відправника з полем `ownerId`.

Лістинг 3.6 - Логіка валідації прав на зміну стану плеєра

```
if (action === 'video-update') {
  const room = rooms.get(roomId);
  const isOwner = room.ownerId === userId;
  const isDemocracy = room.settings.democracy;

  if (isOwner || isDemocracy) {
    room.state = payload.playerState;
    broadcastToRoom(roomId, {
      action: 'apply-status',
      status: room.state,
      initiatorId: userId
    }, userId);  }}
}
```

Логічна умова в конструкції `if` враховує два сценарії: пряму дію власника або активний режим «Демократія», за якого будь-який учасник має права модератора. Якщо перевірка успішна, сервер оновлює глобальний об'єкт стану кімнати в пам'яті та ініціює метод `broadcastToRoom` для розсилки нових координат.

У разі, якщо права доступу обмежені, а запит надійшов від звичайного учасника, сервер просто ігнорує пакет даних, не змінюючи стан плеєра для групи. Це гарантує стабільність сесії та захищає її від десинхронізації через некоректні дії окремих користувачів.

3.2.2 Реалізація альтернативних сценаріїв та обробка помилок

При спробі приєднатися до сесії сервер виконує обов'язкову перевірку існування запитуваного коду у своїй базі даних. Це запобігає спробам входу в неіснуючі або закриті кімнати, гарантуючи цілісність підключень. Якщо ідентифікатор відсутній у колекції активних просторів, сервер надсилає клієнту спеціальне повідомлення про помилку. Інтерфейс розширення обробляє цей сигнал і миттєво виводить візуальне сповіщення, що дозволяє користувачеві відредагувати введені дані без необхідності перезавантажувати вікно або втрачати контекст сторінки.

Механізм активації синхронізації передбачає додатковий рівень захисту від помилкових дій. Якщо спроба прив'язки відео через контекстне меню відбувається без попереднього входу в кімнату, система автоматично перевіряє стан локального сховища на наявність активного ідентифікатора сесії. У разі його відсутності керуючий процес блокує передачу команд та надсилає сигнал до вебсторінки для примусової зупинки процедури. Це запобігає виникненню програмних конфліктів та інформує глядача про необхідність спочатку створити або обрати спільний віртуальний простір.

Реалізовано логіку автоматичного відновлення зв'язку при оновленні сторінки. Тобто система порівнює ідентифікатор поточної вкладки зі збереженими даними й самостійно ініціює запит актуального стану у сервера, роблячи процес синхронізації безперервним і непомітним для користувача.

Автоматичне управління ресурсами на сервері гарантує високу продуктивність системи навіть при великій кількості короткочасних сесій. При розриві зв'язку з будь-яким учасником серверний обробник подій видаляє його ідентифікатор зі списку активних підключень конкретної кімнати. Якщо після завершення цієї операції сесія залишається порожньою, алгоритм повністю очищає кімнату та її налаштування. Такий підхід запобігає накопиченню застарілих даних у пам'яті та забезпечує оптимальне використання серверних потужностей для нових користувачів.

3.2.3 Логічна структура взаємодії модулів

Нижче наведено опис архітектурної взаємодії компонентів розширення, що забезпечує виконання описаної логіки.

- комунікація Popur та Service Worker. Взаємодія відбувається через `chrome.runtime.sendMessage`. Інтерфейс лише ініціює дії, тоді як вся мережева логіка (WebSocket) винесена у фоновий процес;
- комунікація Service Worker та Content Script. Використовується API повідомлень вкладок (`chrome.tabs.sendMessage`). Це дозволяє фоновому

процесу керувати відео-плеєром на будь-якій відкритій сторінці;

- комунікація з сервером. Весь обмін даними відбувається у форматі JSON через єдине WebSocket-з'єднання, що мінімізує накладні витрати на встановлення нових з'єднань при кожній дії.

3.3 Опис структур даних та об'єктів

Для забезпечення стабільної роботи та відмовостійкості розподіленої системи було спроектовано гібридну модель зберігання та передачі даних. Вона базується на поєднанні бази даних PostgreSQL для збереження довгострокової інформації та формату JSON для обміну станами через WebSocket у реальному часі.

Вибір JSON як основного формату обміну обумовлений його легковажністю та нативною підтримкою в Node.js. Логічна модель системи побудована за принципом вкладеності, де центральним вузлом є об'єкт Room. Така архітектура дозволяє серверу оперувати кімнатою як єдиним об'єктом у пам'яті для швидкодії, паралельно синхронізуючи ключові параметри з реляційною базою даних.

3.3.1 Структура ідентифікації користувача (User)

Об'єкт користувача пов'язує клієнта з активним мережевим з'єднанням. У базі даних PostgreSQL користувач представлений записом у таблиці users. Структура цього об'єкта містить наступні поля:

- `userId` містить унікальний анонімний ідентифікатор, який генерується розширенням при першому запуску. Це дозволяє системі зберігати послідовність дій користувача (наприклад, права власника кімнати) навіть після перезавантаження сторінки або тимчасового розриву зв'язку;

- `socket` є технічним дескриптором активного WebSocket-з'єднання, що використовується сервером для адресного надсилання пакетів даних. У

пам'яті сервера це посилання на об'єкт зв'язку, через який здійснюється метод `send()`;

- `roomId` динамічно оновлюється при вході користувача в сесію або виході з неї. Це дозволяє координаційному модулю сервера миттєво визначати пул отримувачів для трансляції (broadcast) подій, групуючи сокети за ідентифікатором кімнати.

3.3.2 Структурна модель віртуальної кімнати (Room)

Кімната є центральним об'єктом системи, який консолідує навколо себе групу користувачів та спільний стан медіа-контенту . Програмна реалізація об'єкта кімнати, що зберігається в оперативній пам'яті сервера, наведена у лістингу 3.7.

Лістинг 3.7 - Структура об'єкта Room у форматі JSON

```
{
  "id": "427",
  "name": "emoji x3",
  "ownerId": "user_82341",
  "settings": {
    "democracy": true,
    "autoSyncUrl": false
  },
  "state": {
    "playerStatus": "PLAYING",
    "second": 124.5,
    "timestamp": 1715245600000
  }
}
```

Поле `id` представлене текстовим тризначним кодом (наприклад, "427"). Такий формат обрано для спрощення вербальної або текстової передачі коду між людьми, що значно знижує поріг входу в сесію порівняно з використанням довгих хеш-строк.

Поле `name` формується як випадкова комбінація емодзі, що виконує роль візуального маркера сесії в інтерфейсі розширення, дозволяючи користувачам швидко ідентифікувати активну кімнату.

Поле `ownerId` зберігає ідентифікатор користувача-ініціатора, що є критично важливим для валідації запитів на зміну глобальних налаштувань та прав доступу.

Вкладений об'єкт `playerState` є найбільш динамічною частиною структури, оскільки він оновлюється при кожній зміні стану відео на стороні клієнта з відповідними правами управління, забезпечуючи актуальність даних для нових учасників.

3.3.3 Стан відтворення медіа-контенту (PlayerState)

Об'єкт стану плеєра забезпечує математичну точність синхронізації між географічно рознесеними клієнтами. Структура цього об'єкта, що передається в межах мережевих пакетів.

На відміну від метаданих кімнати, об'єкт стану плеєра не зберігається в постгресс. Через високу частоту оновлень (декілька разів на секунду), ці дані обробляються в оперативній пам'яті сервера без постійного запису в PostgreSQL для уникнення надлишкового навантаження на дискову підсистему. Структура цього об'єкта містить наступні поля:

- `playerStatus` приймає значення перелічуваного типу (`enum`): `PLAYING` або `PAUSE`. Це дозволяє клієнтським скриптам однозначно визначати метод, який необхідно викликати на об'єкті відео для досягнення ідентичного стану;
- `second` типу `float` фіксує поточний час відтворення з точністю до мілісекунд. Використання числа з плаваючою комою дозволяє уникнути "стрибків" зображення при частому оновленні стану;
- `timestamp` містить час фіксації цього стану за системним годинником сервера. Наявність цього поля дозволяє алгоритму синхронізації

на стороні клієнта враховувати мережеву затримку.

При отриманні пакета клієнтський скрипт обчислює різницю між поточним серверним часом та отриманим значенням timestamp. Ця дельта додається до значення second, що дозволяє нівелювати розсинхронізацію через повільне інтернет-з'єднання та забезпечувати кадрову точність групового перегляду.

3.3.4 Конфігурація параметрів сесії (Settings)

Об'єкт налаштувань визначає бізнес-логіку взаємодії користувачів у межах конкретної кімнати, встановлюючи правила доступу та автоматизації.

Поле democarcy є логічним прапорцем (boolean). Якщо його значення - false, сервер ігнорує будь-які пакети PlayerState, надіслані користувачами, чий userId не збігається з ownerId кімнати. У разі встановлення значення true, система переходить у режим спільного доступу, де кожна подія від довільного учасника транслюється всій групі через механізм broadcast.

Поле autoSyncUrl визначає поведінку системи при переході власника на іншу вебсторінку. Якщо цей прапорець активований, сервер надсилає всім учасникам команду change-url із новою адресою, ініціюючи автоматичне перенаправлення браузерів учасників на відповідний контент. Такий підхід мінімізує потребу в ручній координації дій, забезпечуючи миттєву синхронізацію контексту перегляду для всієї аудиторії сесії.

3.4 Висновки за розділом 3

У третьому розділі було проведено детальне проєктування та реалізацію програмного забезпечення для синхронізації медіа-контенту в реальному часі. Результатом роботи стала багаторівнева система, що поєднує можливості сучасних браузерних API та швидкісних мережевих протоколів. На основі проведеної розробки можна зробити наступні висновки.

Застосована модульна архітектура дозволила чітко розмежувати зони відповідальності. Фоновий процес виконує роль стабільного мережевого шлюзу, скрипти вмісту забезпечують пряму взаємодію з DOM-деревом сторінок, а серверна частина на базі Node.js виступає єдиним джерелом істини для станів сесій.

Розроблений алгоритм перевірки прав та логіка блокування зворотних подій дозволили усунути проблему «ехо-ефектів» та нескінченних рекурсивних оновлень. Використання серверних таймстампів забезпечило точність синхронізації до 0,5 секунди, що є комфортним показником для групового перегляду.

Інтеграція бази даних PostgreSQL для постійного зберігання метаданих кімнат у поєднанні з оперативною пам'яттю для високочастотних оновлень стану плеєра дозволила досягти мінімальних затримок при передачі команд. Використання 3-значних цифрових кодів та емодзі-ідентифікаторів суттєво покращило користувацький досвід, спростивши процес створення та входу в сесію.

Реалізація режимів керування станом забезпечила адаптивність системи до різних сценаріїв використання: від спільного перегляду фільмів у вузькому колі до проведення дистанційних навчальних сесій.

Таким чином, розроблений програмний продукт повністю відповідає поставленим вимогам щодо надійності, швидкодії та зручності використання, забезпечуючи безшовну синхронізацію медіа-контенту без втрати його якості.

4 ЕКСПЛУАТАЦІЯ ТА ТЕСТУВАННЯ ПРОГРАМИ

4.1 Призначення програми

Розширення SyncPlayer призначене для організації спільного перегляду відеоконтенту в режимі реального часу між віддаленими користувачами. Використання орієнтоване на професійне використання під час дистанційного навчання, проведення вебінарів або сумісного аналізу відеоматеріалів командами розробників та дизайнерів. Також підійде для розважального колективного перегляду фільмів, серіалів та трансляцій у колі друзів. Програма забезпечує ефект присутності в єдиному цифровому просторі, де кожен учасник бачить ідентичний кадр у кожен момент часу.

4.2 Умови виконання програми

Обов'язковою умовою експлуатації системи синхронізації медіа-контенту SyncPlayer є наявність адміністратора або технічного фахівця, здатного забезпечити безперебійну роботу серверної частини застосунку.

В обов'язки адміністратора входять первинне розгортання Node.js середовища, налаштування WebSocket-сервера та конфігурація доступу до бази даних PostgreSQL для персистентного зберігання метаданих кімнат. Крім того, адміністратор повинен здійснювати моніторинг активних сесій та, за необхідності, виконувати очищення застарілих записів у базі даних для підтримки високої швидкодії системи.

Технічне забезпечення серверної частини, необхідне для стабільної обробки real-time запитів від великої кількості одночасних користувачів, повинно включати мінімально:

- оперативна пам'ять - 2 Гб (з можливістю розширення при збільшенні кількості активних кімнат);
- дисковий простір - 20 Гб (враховуючи невеликий обсяг текстових даних у БД);

- процесор - Intel Xeon E5-2620 (або з частотою від 2.0 ГГц);
- мережевий інтерфейс - пропускна здатність від 100 Мбіт/с для мінімізації затримок WebSocket-з'єднань.

На боці користувача обов'язковою умовою є наявність браузера на базі двигуна Chromium (версії 88 та вище) з підтримкою Manifest V3. Для коректної синхронізації клієнтське обладнання повинно мати стабільне інтернет-з'єднання зі швидкістю, достатньою для відтворення відео контенту з обраного першоджерела без буферизації.

4.3 Виконання і тестування програми

Реалізацію тестування забезпечено в межах перевірки набору функціональних можливостей, які повинні були бути забезпеченими для роботи користувача. Тестування продемонструвало успішні результати забезпечення всіх запланованих функцій, включаючи стабільність WebSocket-з'єднання, коректність перехоплення подій HTML5-плеєра та роботу механізмів розмежування прав доступу. Далі на основі результатів тестування описано приклади і логіку роботи з програмою.

Головне вікно розширення (рис. 4.1) пропонує користувачеві вибір між двома основними сценаріями: створення нової сесії або приєднання до вже існуючої. При виборі створення кімнати система звертається до сервера, який миттєво генерує унікальний тризначний код та набір емодзі. Цей екран є стартовою точкою взаємодії, де також відображається статус підключення до сервера координації.

Після успішної ініціалізації або введення коректного коду користувач потрапляє до інтерфейсу активної кімнати (рис. 4.2). На цій сторінці відображається ідентифікатор сесії, який необхідно передати іншим учасникам для спільного перегляду. Важливим елементом інтерфейсу є динамічні індикатори, які в реальному часі інформують про стан підключення до сервера (Server: OK) та наявність синхронізованого відео на

поточній сторінці (Video: Synced). Якщо відео не знайдено або не підключено, індикатор підсвічується червоним кольором, сигналізуючи про необхідність вибору медіа-об'єкта.

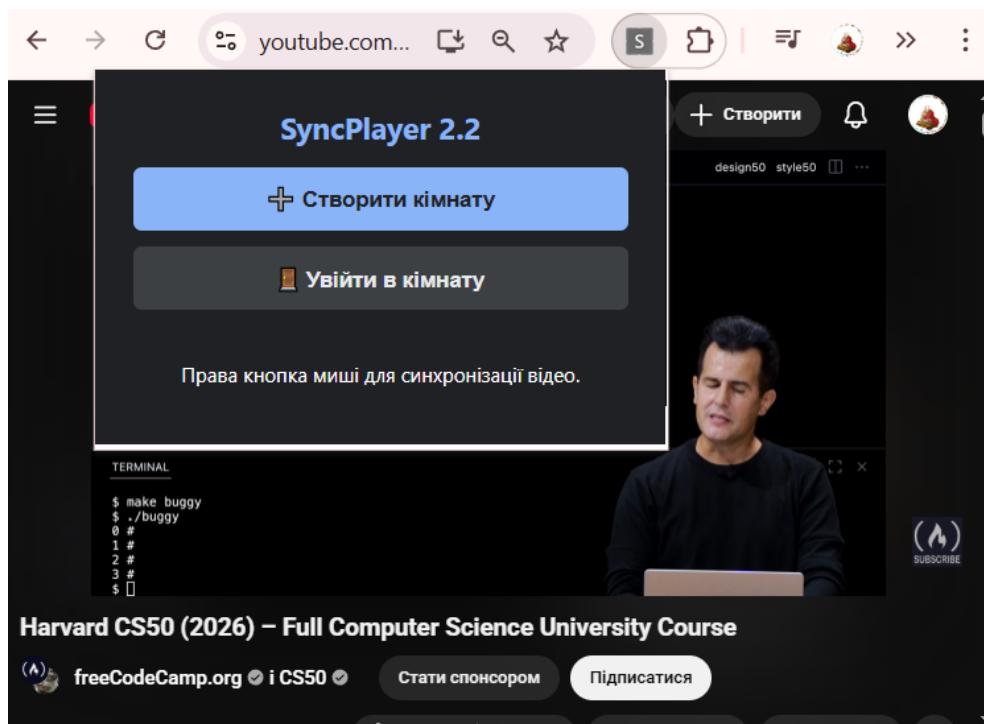
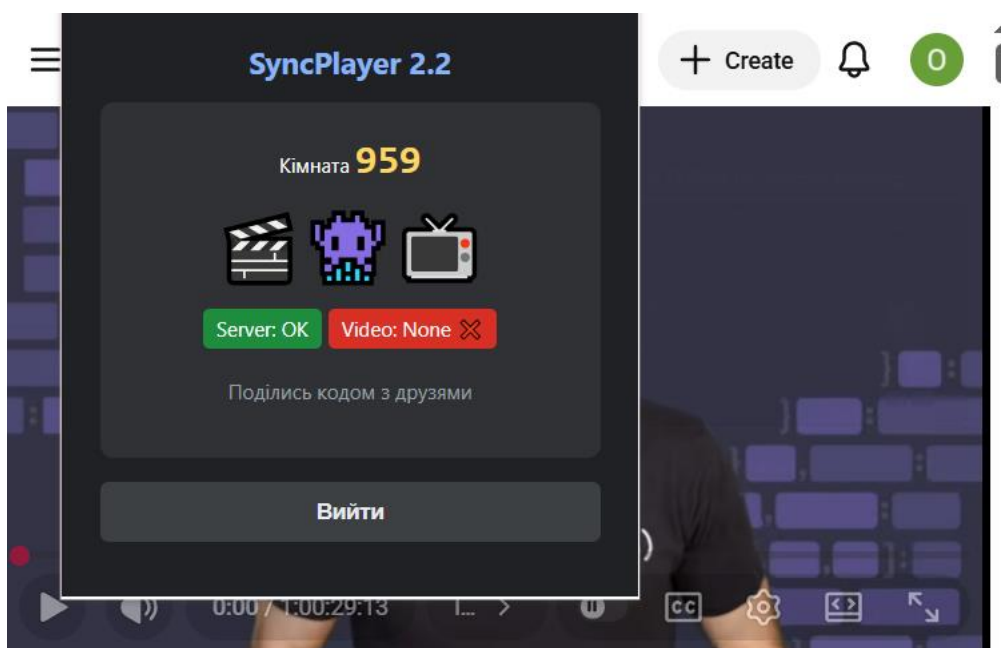


Рисунок 4.1 - Головне меню розширення SyncPlayer



Harvard CS50 (2026) – Full Computer Science University

Рисунок 4.2 - Інтерфейс активної кімнати зі статусами синхронізації

Для активації безпосереднього процесу синхронізації користувач використовує контекстне меню браузера (рис. 4.3). При натисканні правою кнопкою миші на будь-якому відео-плеєрі з'являється пункт «Синхронізувати це відео».

Тестування підтвердило, що після активації цієї функції навколо медіа-елемента з'являється характерна зелена рамка, що слугує візуальним підтвердженням успішного встановлення слухачів подій. Саме ця функція візуального виділення активного об'єкта та інтеграція через контекстне меню забезпечують універсальність роботи на будь-яких веб-ресурсах.

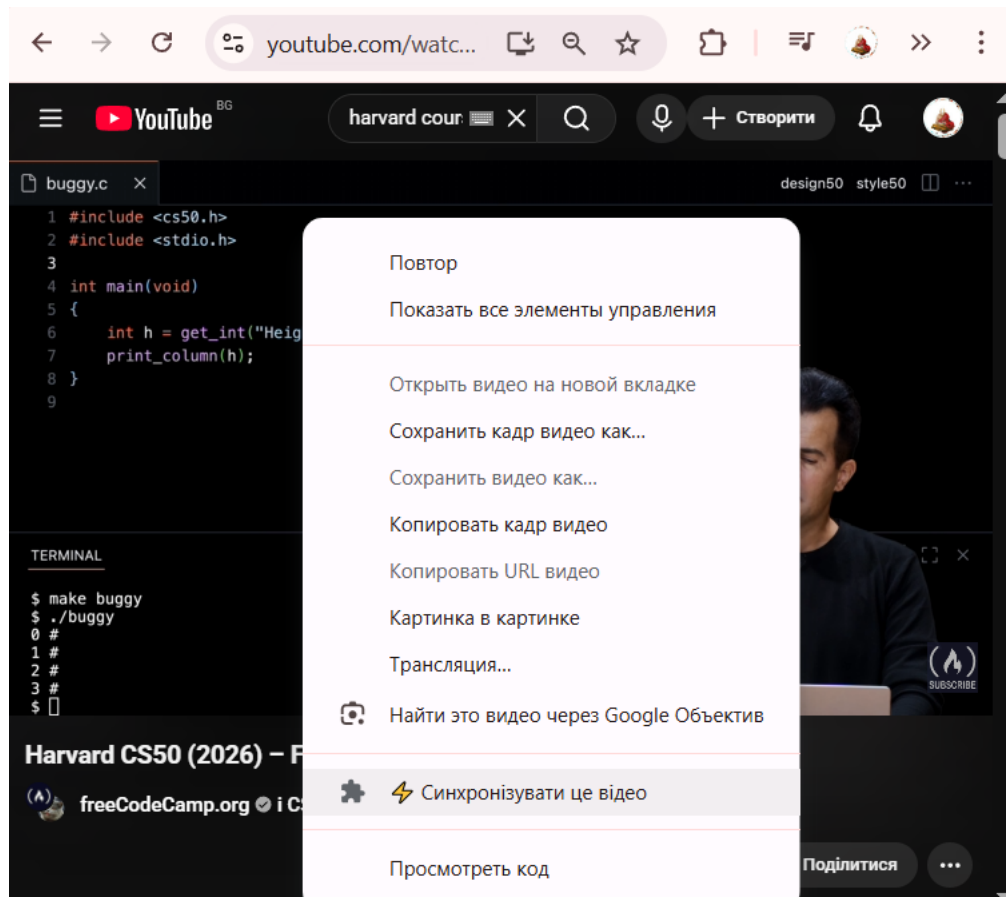


Рисунок 4.3 - Вибір об'єкта синхронізації через контекстне меню

Власник кімнати має доступ до спеціального блоку управління - «Адмін панелі» (рис. 4.4). Тут реалізовано можливість перемикавання режиму «Демократія», що дозволяє або забороняє іншим учасникам керувати станом відтворення.

Тестування показало, що при вимкненому режимі демократії будь-які спроби звичайних учасників змінити час або статус плеєра ігноруються сервером, а їхні локальні плеєри примусово повертаються до часової мітки власника. Також передбачено кнопку «Вийти», яка дозволяє безпечно від'єднатися від сесії.

Пошук та обробка помилкових станів реалізовані через систему миттєвих сповіщень (рис. 4.5).

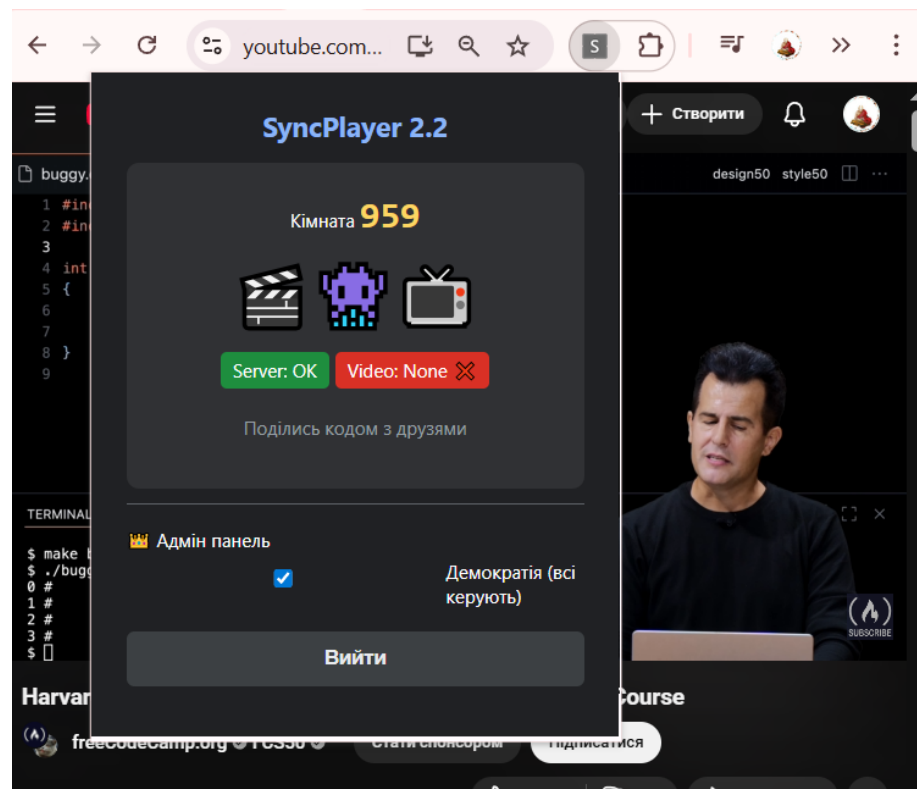


Рисунок 4.4 - Панель адміністратора та налаштування прав доступу

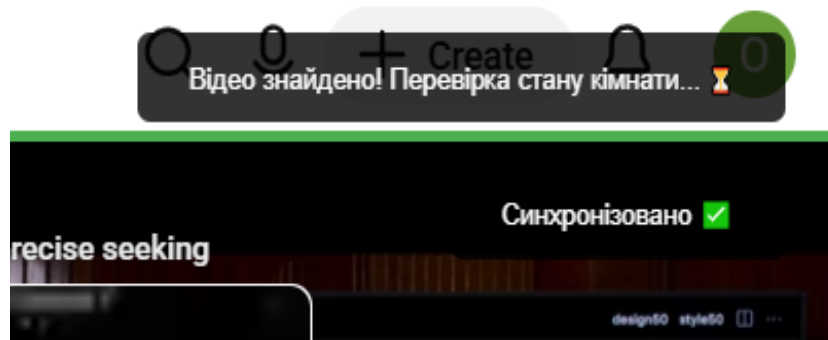


Рисунок 4.5 - Toast-повідомлення

У разі введення неіснуючого коду кімнати або втрати зв'язку із сервером, користувач отримує візуальне попередження у верхньому правому куті вікна браузера. Це забезпечує безперервний зворотний зв'язок та дозволяє оперативно реагувати на технічні проблеми під час експлуатації.

4.4 Висновки за розділом 4

Програму синхронізації спільного перегляду медіа-контенту у браузерному середовищі успішно реалізовано із забезпеченням усіх функціональних можливостей, що були визначені. З наданням відповідного меню для керування послідовністю дій щодо створення сесій, вибору об'єктів синхронізації та модерації прав учасників у реальному часі. Тестування підтвердило стабільність роботи архітектури та високу точність синхронізації станів відтворення на різних веб-ресурсах.

ВИСНОВКИ

Підбиваючи підсумки виконаної роботи, можна констатувати, що поставленої мети розробки системи для синхронізації спільного перегляду медіа-контенту у браузерному середовищі було повністю досягнуто. Цей проєкт став відповіддю на запит сучасної інтернет-спільноти — покращити наявний інструмент для спільного дозвілля, який не жертвує якістю зображення заради синхронності. Під час дослідження було підтверджено, що в умовах глобальної цифровізації формат дистанційного групового споживання контенту стає невід'ємною частиною як повсякденного дозвілля, так і професійних освітніх процесів.

Під час аналізу існуючих рішень було виявлено, що більшість платформ або призводять до значної деградації відео через стиснення потоку, або обмежені вузьким переліком платних стрімінгів. Саме на подолання цих технічних бар'єрів і була спрямована моя розробка, що дозволяє взаємодіяти безпосередньо з оригінальними медіа-ресурсами. Розроблений підхід дозволяє нівелювати залежність від закритих екосистем великих медіа-корпорацій, надаючи користувачам повну свободу у виборі джерел відтворення.

Для реалізації проєкту було обрано сучасний технологічний стек, який базується на мові JavaScript та середовищі Node.js. Це забезпечує високу швидкість розробки та виконання. Використання стандарту Manifest V3 гарантує високий рівень безпеки даних та повну сумісність із сучасними вимогами браузерів до продуктивності розширень. Особливе місце в архітектурі посідає використання протоколу WebSockets та БД PostgreSQL. Це дозволило впровадити гібридну модель управління даними: високошвидкісну обробку станів плеєра в оперативній пам'яті сервера для миттєвої реакції та персистентне зберігання метаданих кімнат у базі даних для надійності системи. Застосування асинхронного підходу Node.js

дозволило оптимізувати обробку великої кількості паралельних запитів, мінімізуючи навантаження на серверні потужності.

Результатом роботи став програмний комплекс SyncPlayer, що охоплює всі визначені вимоги та надає потужні інструменти для управління спільним переглядом. До них належать: зручний вибір будь-якого HTML5-плеєра на сторінці через контекстне меню, автоматична синхронізація стартів, пауз і перемотувань, та гнучка система модерації. Тестування підтвердило, що система стабільно працює на різних веб-ресурсах, забезпечуючи точність синхронізації. Архітектура рішення передбачає можливість подальшого розширення функціоналу, включаючи підтримку складних плеєрів із нестандартними обробниками подій.

У підсумку розроблене програмне забезпечення є повноцінною екосистемою для організації групових заходів у віртуальному просторі. Програма дозволяє користувачам ефективніше дивитися спільно медіа-контент незалежно від їхнього географічного розташування, забезпечуючи при цьому максимальну якість відеоконтенту. Архітектура рішення передбачає можливість подальшого розширення функціоналу, включаючи підтримку складних плеєрів із нестандартними обробниками подій.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Amirtham V. OTT Video Streaming Statistics 2026. *VPlayed*. URL: <https://www.vplayed.com/blog/ott-statistics/> (date of access: 12.05.2026).
2. Palmer R. Disney removes GroupWatch feature. *What's On Disney Plus*. URL: <https://whatsondisneyplus.com/disney-removes-groupwatch-feature/> (date of access: 12.05.2026).
3. About Netflix. *Netflix*. URL: <https://www.netflix.com/> (date of access: 12.05.2026).
4. Zhang L., Wang H. Low Latency and Video Compression Research. *MDPI Electronics*. 2025. Vol. 14, no. 3. URL: <https://www.mdpi.com/2079-9292/14/3/497> (date of access: 12.05.2026).
5. About Discord. *Discord*. URL: <https://discord.com/> (date of access: 12.05.2026).
6. Teleparty Support and FAQ. *Teleparty Official Portal*. URL: <https://www.teleparty.com/support> (date of access: 12.05.2026).
7. About Teleparty. *Teleparty*. URL: <https://www.teleparty.com/> (date of access: 12.05.2026).
8. Driesten C. Synchronization Architecture Issues. *GitHub*. URL: <https://github.com/lichtblick-suite/lichtblick/issues/1048> (date of access: 12.05.2026).
9. About Metastream. *Metastream*. URL: <https://app.getmetastream.com/> (date of access: 12.05.2026).
10. About Watch2Gether. *Watch2Gether*. <https://w2g.tv/en/> (date of access: 12.05.2026).
11. HTMLVideoElement API Documentation. *MDN Web Docs*. URL: <https://developer.mozilla.org/en-US/docs/Web/API/HTMLVideoElement> (date of access: 12.05.2026).
12. Santos-González I., Rivero-García A., Molina-Gil J., Caballero-Gil P. Implementation and Analysis of Real-Time Streaming Protocols. *MDPI*. URL:

<https://www.mdpi.com/1424-8220/17/4/846> (date of access: 12.05.2026).

13. Xiongkuo M., et al. Subjective Quality of Experience in Live Video Sync. *arXiv*. 2024. URL: <https://arxiv.org/abs/2409.17596> (date of access: 12.05.2026).

14. Fette I., Melnikov A. The WebSocket Protocol. *Datatracker*. URL: <https://datatracker.ietf.org/doc/html/rfc6455> (date of access: 12.05.2026).

15. Verwest M. & Muralidharan K. Global Streaming Study 2025. *Simon-Kucher Insights*. URL: <https://www.simon-kucher.com/en/insights/us-results-2025-global-streaming-study-0> (date of access: 12.05.2026).

16. Welcome to Manifest V3. *Chrome for Developers*. URL: <https://developer.chrome.com/docs/extensions/develop/migrate/what-is-mv3> (date of access: 12.05.2026).

ДОДАТОК А
Технічне завдання

Вступ

Споживання медіа-контенту в цифровому середовищі є однією з найбільш динамічних галузей сучасної комунікації. Соціальна взаємодія навколо спільного перегляду фільмів, серіалів та освітніх матеріалів тривалий час була обмежена фізичною присутністю користувачів в одному місці. З розвитком глобальних мереж виникла потреба у створенні ефекту «спільної присутності» для віддалених користувачів. На сьогодні підтримка такої взаємодії здійснюється переважно через трансляцію екрана (Screen Sharing) у месенджерах або через спеціалізовані стрімінгові платформи.

Проте існуючі рішення мають суттєві недоліки: трансляція екрана призводить до значної деградації якості відео та звуку через повторне стиснення потоку, а спеціалізовані платформи часто обмежені лише власним контентом. Не приділяється достатньо уваги реалізації універсальних інструментів, які дозволяли б синхронізувати перегляд безпосередньо на сторінках першоджерел, зберігаючи оригінальну високу якість контенту. Це стосується випадків, коли друзі або спільноти за інтересами хочуть споживати контент «тут і зараз», обмінюючись емоціями в реальному часі без технічних компромісів. Саме на створенні механізму синхронізації станів плеєра через браузерне розширення та серверний координатор зроблено акцент у цій розробці.

A.1 Підстави для розробки

Дипломна кваліфікаційна робота виконувалась у відповідності та на підставі Наказу № 139 від 07 квітня 2026 р. за Національним університетом «Запорізька політехніка», яким визначено тему роботи - «Розробка програмного забезпечення для синхронізації спільного перегляду медіа-контенту у браузерному середовищі».

A.2 Призначення розробки

Програма призначена для покращення організації синхронного споживання медіа-контенту групою віддалених користувачів. Система забезпечує створення віртуальних кімнат, автоматичну синхронізацію станів відтворення (відтворення, пауза, перемотування), управління правами доступу учасників сесії та візуальну індикацію статусу синхронізації безпосередньо в інтерфейсі веб-браузера.

A.3 Основні вимоги до програми

A.3.1 Вимоги до функціональних характеристик

Функціональні характеристики програми синхронізації медіа-контенту повинні включати:

- генерація унікального анонімного ідентифікатора користувача при першому запуску;
- створення віртуальних кімнат для спільного перегляду з генерацією унікального тризначного коду доступу;
- приєднання до існуючої сесії за кодом кімнати;
- автоматичне встановлення WebSocket-з'єднання з сервером координації;
- вибір об'єкта синхронізації (тег <video>) на будь-якій веб-сторінці через контекстне меню браузера;
- візуальне виділення активного відео-плеєра (рамка) та відображення сповіщень про статус синхронізації;
- перехоплення подій відтворення (play, pause) та зміна часової мітки (seeked) на клієнтській стороні;
- трансляція стану плеєра іншим учасникам кімнати в режимі реального часу;
- отримання актуального стану відео при вході в існуючу кімнату;

- керування правами доступу через режим дозвілу або заборони учасникам керувати плеєром;
- відображення списку активних учасників сесії та їхніх ролей у Рорир-вікні;
- можливість ручного відключення відео від синхронізації без виходу з кімнати;
- збереження метаданих кімнат та налаштувань у персистентному сховищі (PostgreSQL);
- обробка станів відтворення в оперативній пам'яті сервера для мінімізації затримок.

A.3.2 Умови експлуатації

Обов'язковою умовою експлуатації системи є наявність серверної частини (Node.js), розгорнутої в мережі Інтернет або локальній мережі, та встановленого браузерного розширення у користувачів (на базі двигуна Chromium). В обов'язки адміністратора сервера входить моніторинг активних сесій, підтримка бази даних PostgreSQL та контроль навантаження на WebSocket-шлюз.

A.3.3 Вимоги до складу та параметрів технічних засобів

Технічне забезпечення сервера повинно включати мінімально:

- процесор: 1 vCPU з частотою від 2.0 ГГц (наприклад, Intel Xeon або аналогічні хмарні рішення);
- жорсткий диск: 20 Гб (для логів та бази даних метаданих кімнат; оскільки медіа-контент не зберігається на сервері, вимоги до простору є мінімальними);
- оперативна пам'ять: 1 Гб (основна частина пам'яті використовується для In-memory Map активних сесій).

На стороні клієнта: ПК або ноутбук з ОС Windows/Linux/macOS та встановленим браузером Chrome (версія 100+) з підтримкою Manifest V3.

A.4 Порядок контролю та приймання

Починаючи з дати видачі завдання 7 квітня, протягом наступних 8 тижнів має бути забезпечено виконання роботи за календарним планом. План визначає етапи: проектування архітектури, розробка серверної частини на Node.js, реалізація розширення (Service Worker, Content Script), тестування синхронізації під навантаженням. Після забезпечення очікуваних результатів формується пояснювальна записка. Остаточним кроком приймання роботи є її публічний захист.

ДОДАТОК Б
Текст програми

Б.1 Текст файла content.js

```

let activeVideo = null;
let isRemoteUpdate = false;
let isInitializing = false;
let lastRightClickedVideo = null;
let roomSettings = null;
let isOwner = false;

function safeSendMessage(message) {
  if (!chrome.runtime?.id) {
    return;
  }
  try {
    chrome.runtime.sendMessage(message).catch((err) => {
      if (err.message.includes('Extension context
invalidated')) return;
      console.error('SyncPlayer Send Error:', err);
    });
  } catch (e) {
    console.log('Context lost.');
```

```

        document.body.appendChild(div);
        setTimeout(() => div.remove(), 3000);
    }

    function detachListeners() {
        if (activeVideo) {
            activeVideo.style.border = 'none';
            showToast('Відео відключено від синхронізації ');
        }

        if (activeVideo) {

            activeVideo = null;

            safeSendMessage({ action: 'video-detached-manual' });
        }

        isInitializing = false;
    }

    document.addEventListener(
        'contextmenu',
        (event) => {

            lastRightClickedVideo =
                event.target.tagName === 'VIDEO'
                ? event.target
                : document.querySelector('video');
        },
        true,
    );

    function attachListeners(video) {
        if (!video) return;

        if (activeVideo === video) {
            showToast('Відео вже синхронізовано! ');
            return;
        }

        if (activeVideo) activeVideo.style.border = 'none';

        activeVideo = video;

        activeVideo.style.border = '4px solid #4CAF50';

        showToast('Відео знайдено!');

        safeSendMessage({

```

```

        action: 'video-attached-request',
        url: window.location.href,
    });

    isInitializing = true;

    chrome.storage.local.get(['settings', 'isOwner'], (res)
=> {
        roomSettings = res.settings || { democracy: true };
        isOwner = res.isOwner || false;
    });

    const eventHandler = () => {

        if (isRemoteUpdate || isInitializing || !activeVideo)
return;
        if (roomSettings && !roomSettings.democracy &&
!isOwner) {

            return;
        }

        setTimeout(() => {
            if (isRemoteUpdate || isInitializing || !activeVideo)
return;

            const status = {
                playerStatus: activeVideo.paused ? 'PAUSE' :
'PLAYING',
                second: activeVideo.currentTime,
                timestamp: Date.now(),
            };
            safeSendMessage({ action: 'update-status', status:
status });
        }, 100);
    };

    activeVideo.addEventListener('play', eventHandler);
    activeVideo.addEventListener('pause', eventHandler);
    activeVideo.addEventListener('seeked', eventHandler);
}

function applyRemoteStatus(serverState) {
    if (!activeVideo) return;

    if (isInitializing) {

```

```

        console.log('Initial sync complete ');

        isInitializing = false;

        showToast('Синхронізовано ');

        safeSendMessage({ action: 'request-server-state' });
    }

    isRemoteUpdate = true;

    const delta = Math.abs(activeVideo.currentTime -
serverState.second);

    if (delta > 0.5) {
        activeVideo.currentTime = serverState.second;
    }

    if (serverState.playerStatus === 'PLAYING') {
        activeVideo.play().catch(() => {});
    } else {
        activeVideo.pause();
    }

    setTimeout(() => {
        isRemoteUpdate = false;
    }, 500);
}

chrome.runtime.onMessage.addListener((msg) => {
    if (msg.action === 'activate-sync-from-context') {
        if (lastRightClickedVideo)
attachListeners(lastRightClickedVideo);
    }

    if (msg.action === 'detach-video') {
        detachListeners();
    }

    if (msg.action === 'sync-failed-no-room') {
        if (activeVideo) {
            detachListeners();
            showToast('⚠ Помилка: Спочатку створіть або
приєднайтеся до кімнати.');
```

```

        if (msg.initiatorId !== res.myUserId ||
msg.initiatorId === 'server') {
            applyRemoteStatus(msg.status);
        }
    });
}

if (msg.action === 'room-settings-updated') {
    roomSettings = msg.settings;

    isOwner = msg.isOwner;
}
});

chrome.storage.onChanged.addListener((changes, areaName) =>
{

    if (areaName === 'local') {
        if (changes.settings) {
            roomSettings = changes.settings.newValue || {
democracy: true };
        }

        if (changes.isOwner) {
            isOwner = changes.isOwner.newValue || false;
        }
    }
});

```

Б.2 Текст файла manifest.json

```

{
    "manifest_version": 3,
    "name": "SyncPlayer",
    "version": "2.2",
    "description": "Synchronize video playback with friends
via Context Menu.",
    "permissions": ["activeTab", "scripting", "tabs",
"storage", "contextMenus"],
    "background": {
        "service_worker": "service_worker.js"
    },
    "action": {
        "default_popup": "popup.html"
    },
    "content_scripts": [
        {
            "matches": ["<all_urls>"],
            "js": ["content.js"],
            "all_frames": true
        }
    ]
}

```

```

    }
  ]
}

```

Б.3 Текст файлу popup.js

```

const screens = {
  menu: document.getElementById('screen-menu'),
  join: document.getElementById('screen-join'),
  room: document.getElementById('screen-room'),
};

function showScreen(name) {
  Object.values(screens).forEach((s) =>
s.classList.add('hidden'));
  screens[name].classList.remove('hidden');
}

function updateUI() {
  chrome.storage.local.get(
    ['roomId', 'roomName', 'connected', 'isOwner',
'settings', 'videoAttached'],
    (data) => {
      if (data.roomId) {
        showScreen('room');
        document.getElementById('display-room-
id').textContent = data.roomId;
        document.getElementById('display-room-
emoji').textContent =
          data.roomName || '⌚';

        const statusEl =
document.getElementById('connection-status');

        statusEl.textContent = data.connected ? 'Server:
OK' : 'Server: Off';

        statusEl.className = `status-badge ${
          data.connected ? 'status-connected' : 'status-
disconnected'
        }`;

        const videoEl = document.getElementById('video-
status');
        if (data.videoAttached) {
          videoEl.textContent = 'Video: Synced ';
          videoEl.className = 'status-badge status-
connected';
        } else {
          videoEl.textContent = 'Video: None ';

```

```

        videoEl.className = 'status-badge status-
disconnected';
    }

    const adminPanel = document.getElementById('admin-
controls');
    if (data.isOwner) {
        adminPanel.classList.remove('hidden');

        document.getElementById('chk-democracy').checked
=
        data.settings?.democracy || false;
    } else {
        adminPanel.classList.add('hidden');
    }

    const btnUnsync = document.getElementById('btn-
unsync-video');

    if (data.videoAttached) {
        btnUnsync.classList.remove('hidden');
    } else {
        btnUnsync.classList.add('hidden');
    }
    } else {
        showScreen('menu');
    }
    }
    );
}

    document.getElementById('btn-
create').addEventListener('click', () => {
        chrome.runtime.sendMessage({ action: 'create-room' });
    });

    document.getElementById('btn-join-
menu').addEventListener('click', () => {
        showScreen('join');
    });

    document.getElementById('btn-
back').addEventListener('click', () => {
        showScreen('menu');
    });

    document.getElementById('btn-join-
action').addEventListener('click', () => {

```

```

        const code = document.getElementById('room-code-
input').value;

        if (code.length === 3) {
            chrome.runtime.sendMessage({ action: 'join-room',
roomId: code });
        } else {
            alert('Введіть 3 цифри!');
        }
    });

    document.getElementById('btn-
leave').addEventListener('click', () => {
        chrome.runtime.sendMessage({ action: 'leave-room' });
    });

    document.getElementById('chk-
democracy').addEventListener('change', (e) => {
        chrome.storage.local.get('roomId', (res) => {
            if (res.roomId) {
                chrome.runtime.sendMessage({
                    action: 'update-settings',
                    roomId: res.roomId,
                    settings: { democracy: e.target.checked },
                });
            }
        });
    });

    document.getElementById('btn-unsync-
video').addEventListener('click', () => {
        if (
            confirm(
                'Ви впевнені, що хочете відключити відео?'
            )
        ) {
            chrome.runtime.sendMessage({ action: 'unsync-video-
manual' });
        }
    });

    chrome.storage.onChanged.addListener(() => {
        updateUI();
    });

    document.addEventListener('DOMContentLoaded', updateUI);

```

Б.4 Текст файла service_worker.js

```

const WS_URL = 'ws://localhost:3000';
let socket = null;
let keepAliveInterval;

function getUserId(cb) {
  chrome.storage.local.get('myUserId', (res) => {
    if (res.myUserId) cb(res.myUserId);
    else {
      const id = 'user_' + Math.floor(Math.random() *
100000);
      chrome.storage.local.set({ myUserId: id });
      cb(id);
    }
  });
}

function startKeepAlive() {
  if (keepAliveInterval) clearInterval(keepAliveInterval);
  keepAliveInterval = setInterval(
    () => chrome.runtime.getPlatformInfo(() => {}),
    20000,
  );
}

function stopKeepAlive() {
  clearInterval(keepAliveInterval);
}

function detachVideo(tabId) {
  chrome.storage.local.set({
    videoAttached: false,
    syncedTabId: null,
  });
  if (tabId) {
    chrome.tabs.sendMessage(tabId, { action: 'detach-video'
}).catch(() => {
  });
  }
}

```

```

function connect(roomId, actionType) {
  if (socket) socket.close();

  getUserId((userId) => {
    socket = new WebSocket(WS_URL);

    socket.onopen = () => {
      console.log('WS Connected');
      startKeepAlive();

      const msg = { action: actionType, userId: userId,
roomId: roomId };

      if (actionType === 'create-room') {
        chrome.storage.local.get(['syncedTabId'], (res) =>
{
          if (res.syncedTabId) {

            chrome.tabs.get(res.syncedTabId, (tab) => {
              if (tab) {
                msg.payload = { url: tab.url };
              }
              socket.send(JSON.stringify(msg));
            });
          } else {

            chrome.tabs.query({ active: true,
currentWindow: true }, (tabs) => {
              if (tabs[0]) msg.payload = { url: tabs[0].url
};

              socket.send(JSON.stringify(msg));
            });
          }
        });
      } else {
        socket.send(JSON.stringify(msg));
      }
    };

    socket.onmessage = (event) => {
      const data = JSON.parse(event.data);
      handleServerMessage(data);
    };

    socket.onclose = () => {
      console.log('WS Closed');
      chrome.storage.local.set({ connected: false });
      stopKeepAlive();
    };
  });
}

```

```

function handleServerMessage(data) {
  switch (data.action) {
    case 'joined':
      chrome.storage.local.set({
        roomId: data.roomId,
        roomName: data.roomName,
        connected: true,
        isOwner: data.isOwner,
        settings: data.settings,
        videoAttached: false,
      });

      if (data.videoUrl) {
        chrome.tabs.query({ active: true, currentWindow:
true }, (tabs) => {
          const currentTab = tabs[0];
          if (currentTab && currentTab.url !==
data.videoUrl) {

            chrome.scripting.executeScript({
              target: { tabId: currentTab.id },
              func: (url) => {
                if (confirm(`Room is watching: ${url}\nGo
there?`))
                  window.location.href = url;
              },
              args: [data.videoUrl],
            });
          }
        });
      }

      requestServerState();
      break;

    case 'sync-video':
      chrome.storage.local.get('syncedTabId', (res) => {
        const targetTabId = res.syncedTabId;

        if (targetTabId) {
          chrome.tabs
            .sendMessage(targetTabId, {
              action: 'apply-status',
              status: data.state,
              initiatorId: data.initiatorId,
            })
            .catch((err) => {

              if (err.message.includes('Could not establish
connection')) {
                chrome.storage.local.set({

```

```

        videoAttached: false,
        syncedTabId: null,
    });
    }
    });
    break;

case 'error':
    alert('SyncPlayer Error: ' + data.message);

    break;

case 'room-meta-update':
    chrome.storage.local.set({
        settings: data.settings,
        hosts: data.hosts,
        users: data.users,
        ownerId: data.ownerId,
    });

    break;
}
}

chrome.runtime.onMessage.addListener((req, sender,
sendResponse) => {

    if (req.action === 'create-room' || req.action === 'join-
room')
        connect(req.roomId, req.action);

    if (req.action === 'leave-room') {
        chrome.storage.local.get('syncedTabId', (res) => {
            if (socket) socket.close();

            detachVideo(res.syncedTabId);

            chrome.storage.local.set({ roomId: null, connected:
false });
        });
    }

    if (req.action === 'request-server-state') {
        requestServerState();
    }

    if (req.action === 'unsync-video-manual') {
        chrome.storage.local.get('syncedTabId', (res) => {

```

```

        detachVideo(res.syncedTabId);
    });
}

if (req.action === 'video-detached-manual') {
    chrome.storage.local.set({
        videoAttached: false,
        syncedTabId: null,
    });
}

if (req.action === 'update-settings') {
    chrome.storage.local.get(['myUserId'], (res) => {
        if (socket && socket.readyState === WebSocket.OPEN) {
            socket.send(
                JSON.stringify({
                    action: 'update-settings',
                    roomId: req.roomId,
                    userId: res.myUserId,
                    payload: { settings: req.settings },
                }),
            );
        }
    });
}

if (req.action === 'video-attached-request') {
    chrome.storage.local.get(['roomId', 'myUserId'], (res)
=> {
        if (!res.roomId) {
            chrome.tabs
                .sendMessage(sender.tab.id, { action: 'sync-
failed-no-room' })
                .catch(() => {});
            return;
        }

        chrome.storage.local.set({
            videoAttached: true,
            syncedTabId: sender.tab.id,
        });

        if (socket && socket.readyState === WebSocket.OPEN) {
            socket.send(
                JSON.stringify({
                    action: 'get-state',
                    roomId: res.roomId,
                    userId: res.myUserId,
                }),
            );
        }
    });
}

```

```

        );
    }
    });
}

    if (req.action === 'update-status') {
        chrome.storage.local.get(['roomId', 'myUserId'], (res)
=> {
            if (res.roomId && socket && socket.readyState ===
WebSocket.OPEN) {
                socket.send(
                    JSON.stringify({
                        action: 'video-update',
                        roomId: res.roomId,
                        userId: res.myUserId,
                        payload: { playerState: req.status },
                    }),
                );
            }
        });
    }
});

function requestServerState() {
    chrome.storage.local.get(['roomId', 'myUserId'], (res) =>
{
        if (res.roomId && socket && socket.readyState ===
WebSocket.OPEN) {
            console.log(`Sending GET-STATE request for Room
${res.roomId}`);

            socket.send(
                JSON.stringify({
                    action: 'get-state',
                    roomId: res.roomId,
                    userId: res.myUserId,
                }),
            );
        }
    });
}

chrome.runtime.onInstalled.addListener(() => {
    chrome.contextMenus.create({
        id: 'sync-video-action',
        title: '⚡ Синхронізувати це відео',
        contexts: ['all'],
    });
});

chrome.contextMenus.onClicked.addListener((info, tab) => {

```

```

    if (info.menuItemId === 'sync-video-action') {
      chrome.tabs
        .sendMessage(tab.id, { action: 'activate-sync-from-
context' })
        .catch(() => console.log('Please refresh page'));
    }
  });

```

Б.5 Текст файла db_repository.js

```

/**
 * Database Repository for Video Sync Server
 * Handles all database operations for rooms, users, and
settings
 */

const db = require('./config');

/**
 * Create a new room with default settings (transaction)
 * @param {string} id - Room ID (3-digit string)
 * @param {string} name - Room name (emoji string)
 * @param {string} ownerId - User ID of room owner
 * @returns {Promise<{id, name, ownerId, democracy,
autoSyncUrl}>}
 */
async function createRoom(id, name, ownerId) {
  const client = await db.getClient();
  try {
    await client.query('BEGIN');

    await client.query(
      'INSERT INTO rooms (id, name, owner_id) VALUES ($1,
$2, $3)',
      [id, name, ownerId],
    );

    await client.query(
      'INSERT INTO room_settings (room_id, democracy,
auto_sync_url) VALUES ($1, $2, $3)',
      [id, false, true],
    );

    await client.query('COMMIT');

    console.log(`[DB] Room created: ${id} (${name}) by
${ownerId}`);

```

```

        return { id, name, ownerId, democracy: false,
autoSyncUrl: true };
    } catch (err) {
        await client.query('ROLLBACK');

        console.error('[DB Error] Failed to create room:',
err.message);
        throw err;
    } finally {
        client.release();
    }
}

/**
 * Get room details with settings (JOIN)
 * @param {string} roomId - Room ID
 * @returns {Promise<{id, name, ownerId, createdAt,
democracy, autoSyncUrl} | null>}
 */
async function getRoomDetails(roomId) {
    const query = `
        SELECT
            r.id,
            r.name,
            r.owner_id as "ownerId",
            r.created_at as "createdAt",
            s.democracy,
            s.auto_sync_url as "autoSyncUrl"
        FROM rooms r
        LEFT JOIN room_settings s ON r.id = s.room_id
        WHERE r.id = $1
    `;

    try {
        const result = await db.query(query, [roomId]);
        return result.rows[0] || null;
    } catch (err) {
        console.error('[DB Error] Failed to get room details:',
err.message);
        throw err;
    }
}

/**
 * Get all rooms
 * @returns {Promise<Array>}
 */
async function getAllRooms() {
    const query = `
        SELECT
            r.id,

```

```

        r.name,
        r.owner_id as "ownerId",
        r.created_at as "createdAt",
        s.democracy,
        s.auto_sync_url as "autoSyncUrl",
        COUNT(u.id) as "userCount"
    FROM rooms r
    LEFT JOIN room_settings s ON r.id = s.room_id
    LEFT JOIN users u ON r.id = u.room_id
    GROUP BY r.id, r.name, r.owner_id, r.created_at,
s.democracy, s.auto_sync_url
    ORDER BY r.created_at DESC
`;

    try {
        const result = await db.query(query);

        return result.rows;
    } catch (err) {
        console.error('[DB Error] Failed to get all rooms:',
err.message);
        throw err;
    }
}

/**
 * Delete a room (cascade deletes users and settings)
 * @param {string} roomId - Room ID
 * @returns {Promise<boolean>} True if deleted, false if
not found
 */
async function removeRoom(roomId) {
    try {
        const result = await db.query('DELETE FROM rooms WHERE
id = $1', [roomId]);

        const deleted = result.rowCount > 0;

        if (deleted) {
            console.log(`[DB] Room deleted: ${roomId}`);
        }
        return deleted;
    } catch (err) {

        console.error('[DB Error] Failed to delete room:',
err.message);

        throw err;
    }
}

```

```

/**
 * Update room settings
 * @param {string} roomId - Room ID
 * @param {object} settings - Settings object {democracy,
autoSyncUrl}
 * @returns {Promise<{democracy, autoSyncUrl}>}
 */
async function updateRoomSettings(roomId, settings) {
  const { democracy, autoSyncUrl } = settings;

  let query = 'UPDATE room_settings SET ';
  const params = [];
  const updates = [];
  let paramCount = 1;

  if (democracy !== undefined) {
    updates.push(`democracy = ${`${paramCount}`} `);

    params.push(democracy);

    paramCount++;
  }

  if (autoSyncUrl !== undefined) {
    updates.push(`auto_sync_url = ${`${paramCount}`} `);

    params.push(autoSyncUrl);

    paramCount++;
  }

  if (updates.length === 0) {
    return null;
  }

  query += updates.join(', ');

  query += ` WHERE room_id = ${`${paramCount}`} `;

  params.push(roomId);

  try {
    const result = await db.query(query, params);
    if (result.rowCount > 0) {

      const updated = await getRoomDetails(roomId);

      console.log(`[DB] Room settings updated: ${roomId}`);
      return {
        democracy: updated.democracy,

```

```

        autoSyncUrl: updated.autoSyncUrl,
    };
    }
    return null;
  } catch (err) {
    console.error('[DB Error] Failed to update room
settings:', err.message);
    throw err;
  }
}

```

```

/**
 * Add or update user in room
 * @param {string} userId - User ID
 * @param {string} roomId - Room ID
 * @param {string} role - User role (OWNER or PARTICIPANT)
 * @returns {Promise<{id, role, roomId}>}
 */
async function addUserToRoom(userId, roomId, role =
'PARTICIPANT') {
  try {
    await db.query(
      `INSERT INTO users (id, room_id, role)
      VALUES ($1, $2, $3)
      ON CONFLICT (id) DO UPDATE
      SET room_id = $2, role = $3, last_active =
CURRENT_TIMESTAMP`,
      [userId, roomId, role],
    );

    console.log(`[DB] User added to room: ${userId} →
${roomId} (${role})`);
    return { id: userId, role, roomId };
  } catch (err) {
    console.error('[DB Error] Failed to add user to room:',
err.message);
    throw err;
  }
}

```

```

/**
 * Remove user from room (sets room_id to NULL)
 * @param {string} userId - User ID
 * @param {string} roomId - Room ID
 * @returns {Promise<boolean>} True if user was in the room
 */
async function removeUserFromRoom(userId, roomId) {
  try {
    const result = await db.query(

```

```

        'UPDATE users SET room_id = NULL WHERE id = $1 AND
room_id = $2',
        [userId, roomId],
    );

    if (result.rowCount > 0) {
        console.log(`[DB] User removed from room: ${userId} ←
${roomId}`);
    }

    return result.rowCount > 0;
} catch (err) {
    console.error('[DB Error] Failed to remove user from
room:', err.message);
    throw err;
}
}

/**
 * Get all users in a room
 * @param {string} roomId - Room ID
 * @returns {Promise<Array>}
 */
async function getUsersInRoom(roomId) {
    try {
        const result = await db.query(
            'SELECT id, role, last_active FROM users WHERE
room_id = $1 ORDER BY last_active DESC',
            [roomId],
        );
        return result.rows;
    } catch (err) {
        console.error('[DB Error] Failed to get users in
room:', err.message);
        throw err;
    }
}

/**
 * Update user's last_active timestamp
 * @param {string} userId - User ID
 * @returns {Promise<boolean>}
 */
async function updateUserLastActive(userId) {
    try {
        const result = await db.query(
            'UPDATE users SET last_active = CURRENT_TIMESTAMP
WHERE id = $1',
            [userId],
        );
        return result.rowCount > 0;
    } catch (err) {

```

```

        console.error('[DB Error] Failed to update user
last_active:', err.message);
        throw err;
    }
}

/**
 * Get user by ID
 * @param {string} userId - User ID
 * @returns {Promise<{id, role, roomId, lastActive} |
null>}
 */
async function getUser(userId) {
    try {
        const result = await db.query(
            `SELECT
            id,
            role,
            room_id as "roomId",
            last_active as "lastActive"
            FROM users
            WHERE id = $1`,
            [userId],
        );

        return result.rows[0] || null;
    } catch (err) {
        console.error('[DB Error] Failed to get user:',
err.message);
        throw err;
    }
}

async function deleteUser(userId) {
    try {
        const result = await db.query('DELETE FROM users WHERE
id = $1', [userId]);

        if (result.rowCount > 0) {
            console.log(`[DB] User deleted: ${userId}`);
        }

        return result.rowCount > 0;
    } catch (err) {
        console.error('[DB Error] Failed to delete user:',
err.message);

        throw err;
    }
}

```

```

module.exports = {
  createRoom,
  getRoomDetails,
  getAllRooms,
  removeRoom,

  updateRoomSettings,

  addUserToRoom,
  removeUserFromRoom,
  getUsersInRoom,
  updateUserLastActive,
  getUser,
  deleteUser,
};

```

Б.6 Текст файлу index.js

```

const WebSocket = require('ws');
const http = require('http');

const server = http.createServer();
const wss = new WebSocket.Server({ server });

const rooms = new Map();

const EMOJIS = ['', '', '', '', '', '', '', '', '', ''];

function generateRoomId() {
  let id;
  do {
    id = Math.floor(Math.random() * 900 + 100).toString();
  } while (rooms.has(id));
  return id;
}

function getRandomEmojis() {
  return Array(3)
    .fill()
    .map(() => EMOJIS[Math.floor(Math.random() *
EMOJIS.length)])
    .join('');
}

wss.on('connection', (ws) => {
  let currentRoomId = null;
  let currentUserId = null;

  ws.on('message', (message) => {

```

```

let data;

try {
  data = JSON.parse(message);
} catch (e) {
  console.error('Invalid JSON received:', message);
  return;
}

const { action, userId, roomId, payload } = data;
currentUserId = userId;

console.log(`Received action: ${action} `);

switch (action) {
  case 'create-room': {
    const newRoomId = generateRoomId();
    const name = getRandomEmojis();

    rooms.set(newRoomId, {
      id: newRoomId,
      name: name,
      ownerId: userId,
      hosts: [userId],
      settings: { democracy: true, autoSyncUrl: false
},
      videoUrl: payload?.url || null,
      playerState: {
        playerStatus: 'INIT',
        second: 0,
        timestamp: Date.now(),
      },
      users: new Map(),
    });
    handleJoinRoom(newRoomId, userId, ws);
    break;
  }

  case 'join-room': {
    handleJoinRoom(roomId, userId, ws);
    break;
  }

  case 'leave-room': {
    handleLeaveRoom(currentRoomId, userId);
    break;
  }

  case 'get-state': {
    const room = rooms.get(roomId);
    if (room) {

```

```

        ws.send(
            JSON.stringify({
                action: 'sync-video',
                state: room.playerState,
                videoUrl: room.videoUrl,
                initiatorId: 'server',
            }),
        );
    }
    break;
}

case 'video-update': {

    const room = rooms.get(roomId);
    if (!room) return;

    const isAllowed =
room.settings.democracy    ||
room.hosts.includes(userId);

    if (isAllowed) {
        if (payload.playerState) {
            room.playerState = payload.playerState;
        }
        if (payload.videoUrl && payload.videoUrl !==
room.videoUrl) {
            room.videoUrl = payload.videoUrl;
            room.playerState = {
                playerStatus: 'PAUSE',
                second: 0,
                timestamp: Date.now(),
            };
        }

        broadcastToRoom(
            roomId,
            {
                action: 'sync-video',
                state: room.playerState,
                videoUrl: room.videoUrl,
                initiatorId: userId,
            },
            ws,
        );
    }
    break;
}

case 'update-settings': {

    const room = rooms.get(roomId);

```

```

        if (!room) {
            ws.send(
                JSON.stringify({ action: 'error', message:
'Room not found' })),
            );
            return;
        }

        if (room.ownerId !== userId) {
            ws.send(
                JSON.stringify({
                    action: 'error',
                    message: 'Only room owner can change
settings',
                })),
            );
            return;
        }

        if (payload.settings) {
            room.settings = { ...room.settings,
...payload.settings };
        }

        broadcastRoomState(roomId);
        break;
    }
}
console.log(`rooms: ${[...rooms.values()].length}`);
});

ws.on('close', () => {
    if (currentRoomId && currentUserId) {
        handleLeaveRoom(currentRoomId, currentUserId);
    }
});

function handleJoinRoom(roomId, userId, socket) {
    const room = rooms.get(roomId);

    if (!room) {
        socket.send(
            JSON.stringify({ action: 'error', message: 'Room
not found' })),
        );
        return;
    }

    currentRoomId = roomId;

    room.users.set(userId, socket);

```

```

socket.send(
  JSON.stringify({
    action: 'joined',
    roomId: roomId,
    roomName: room.name,
    isOwner: room.ownerId === userId,
    settings: room.settings,
    videoUrl: room.videoUrl,
  }),
);

broadcastRoomState(roomId);
}

function handleLeaveRoom(roomId, userId) {
  const room = rooms.get(roomId);
  if (!room) return;

  room.users.delete(userId);

  if (room.users.size === 0) {
    rooms.delete(roomId);
    console.log(`Room ${roomId} deleted (empty).`);
  } else {

    if (room.ownerId === userId) {
      const nextUser = room.users.keys().next().value;

      room.ownerId = nextUser;

      if (!room.hosts.includes(nextUser))
room.hosts.push(nextUser);
    }

    broadcastRoomState(roomId);
  }
}

function broadcastRoomState(roomId) {
  const room = rooms.get(roomId);

  if (!room) return;

  const userList = Array.from(room.users.keys());

  const msg = JSON.stringify({
    action: 'room-meta-update',
    users: userList,
    ownerId: room.ownerId,
    hosts: room.hosts,
    settings: room.settings,
  });

```

```

    });

    room.users.forEach((client) => {
        if (client.readyState === WebSocket.OPEN)
client.send(msg);
    });
}

function broadcastToRoom(roomId, data, excludeSocket) {
    const room = rooms.get(roomId);
    if (!room) return;

    const msg = JSON.stringify(data);
    room.users.forEach((client) => {
        if (client !== excludeSocket && client.readyState ===
WebSocket.OPEN) {
            client.send(msg);
        }
    });
}

server.listen(3000, () => {
    console.log('Server started on port 3000');
});

```

ДОДАТОК В
Слайди презентації

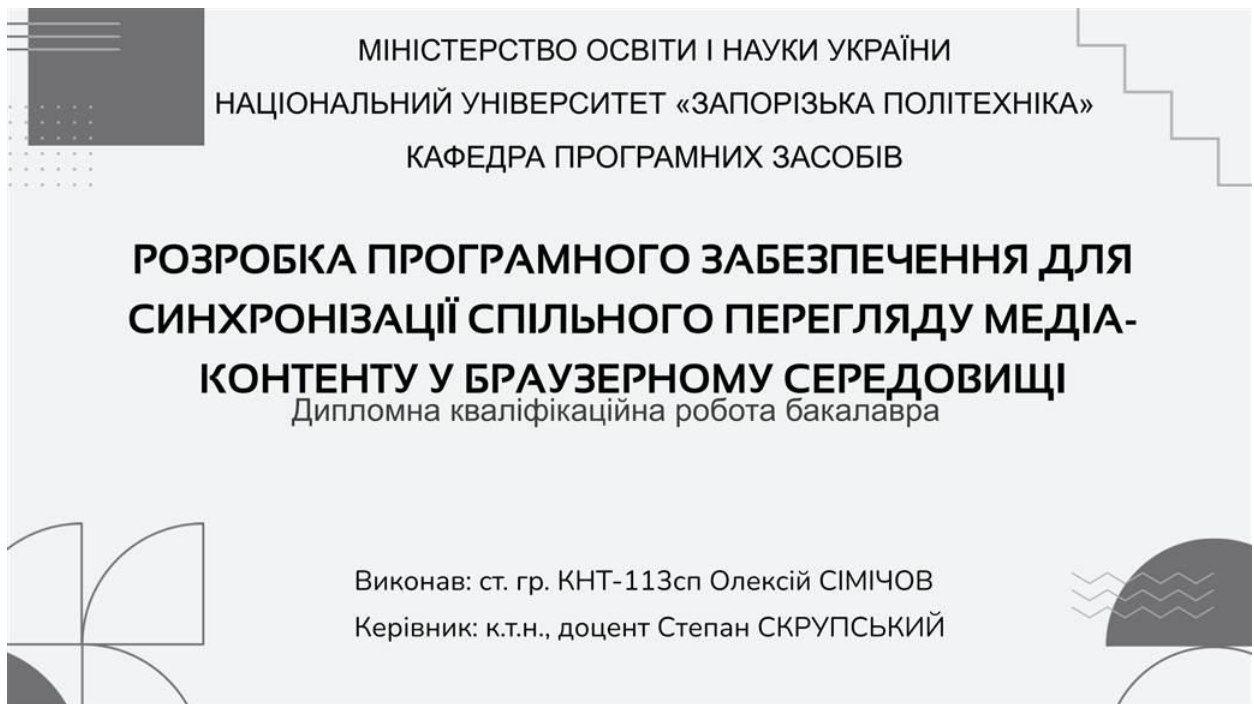


Рисунок В.1 - Слайд 1

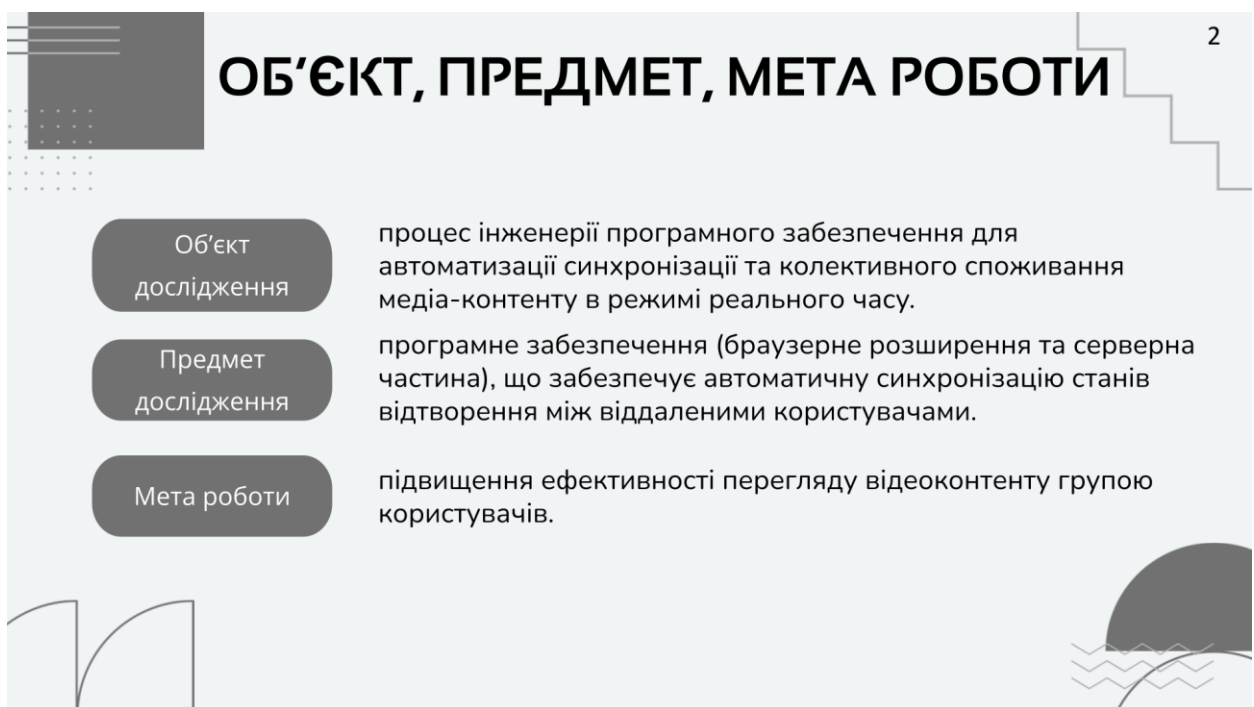


Рисунок В.2 - Слайд 2

ПОСТАНОВА ЗАВДАННЯ

- Проаналізувати програмне забезпечення для організації спільного споживання медіа-контенту в режимі реального часу;
- Визначити методи та засоби проектування програмного забезпечення;
- Сформулювати вимоги до системи спільного споживання медіа-контенту;
- Виконати реалізацію програмного забезпечення для керування заходами та взаємодії користувачів.

Рисунок В.3 - Слайд 3

ПОРІВНЯННЯ ЗАСОБІВ ОРГАНІЗАЦІЇ СПІЛЬНОГО ПЕРЕГЛЯДУ МЕДІА-КОНТЕНТУ

Критерій	Discord (Screen Share)	Teleparty	Watch2Gether
Метод синхронізації	Потокова трансляція відеосигналу	Синхронізація станів плеєра (метадані)	Вбудований програвач / API
Якість зображення	Знижена (через стиснення потоку)	Максимальна (першоджерело)	Залежить від API джерела
Універсальність (підтримка сайтів)	Висока (будь-яке вікно)	Низька (лише популярні стрімінги)	Середня (лише підтримувані сервіси)
Навантаження на систему та мережу	Високе (кодування відео)	Низьке	Середнє
Середовище перегляду	Окремий застосунок	Сайт стрімінгу	Сторонній вебсайт
Вимоги до учасників	Наявність застосунку	Однакові підписки	Відсутні

Рисунок В.4 - Слайд 4

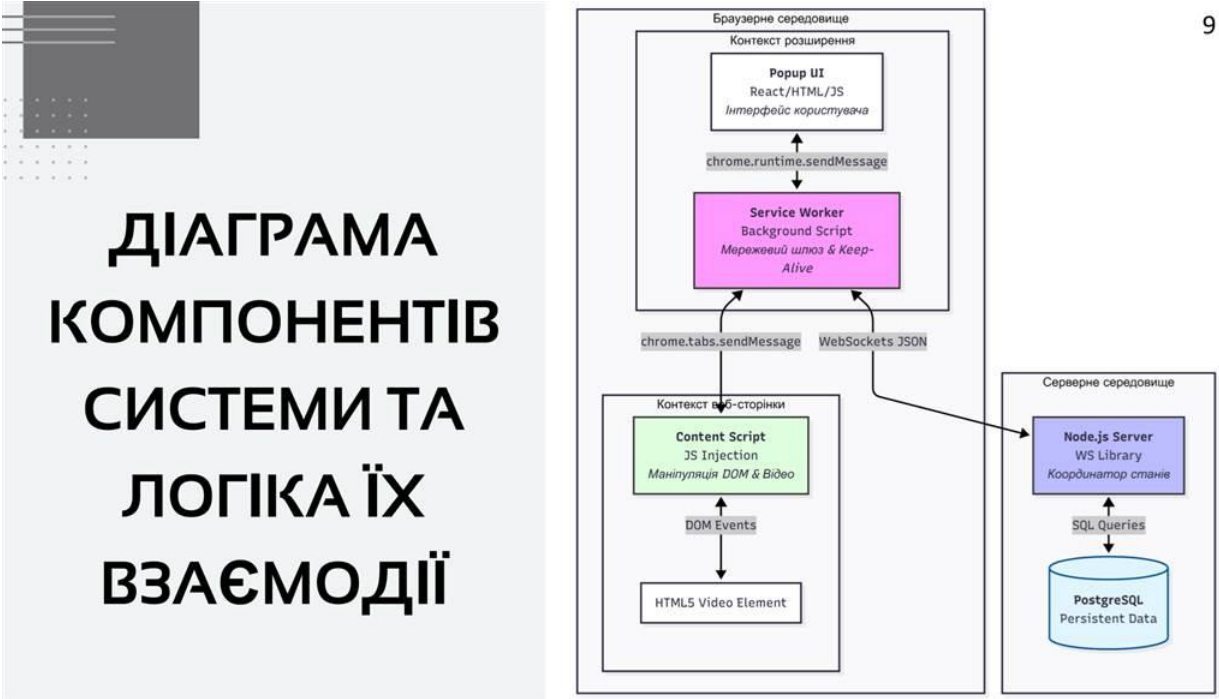


Рисунок В.9 - Слайд 9

ER-МОДЕЛЬ БД

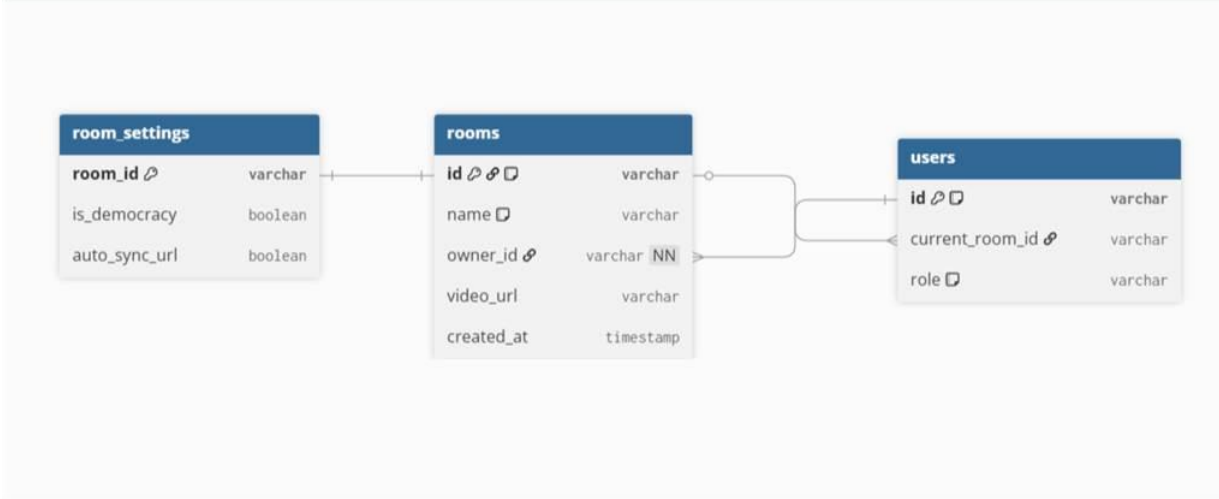


Рисунок В.10 - Слайд 10

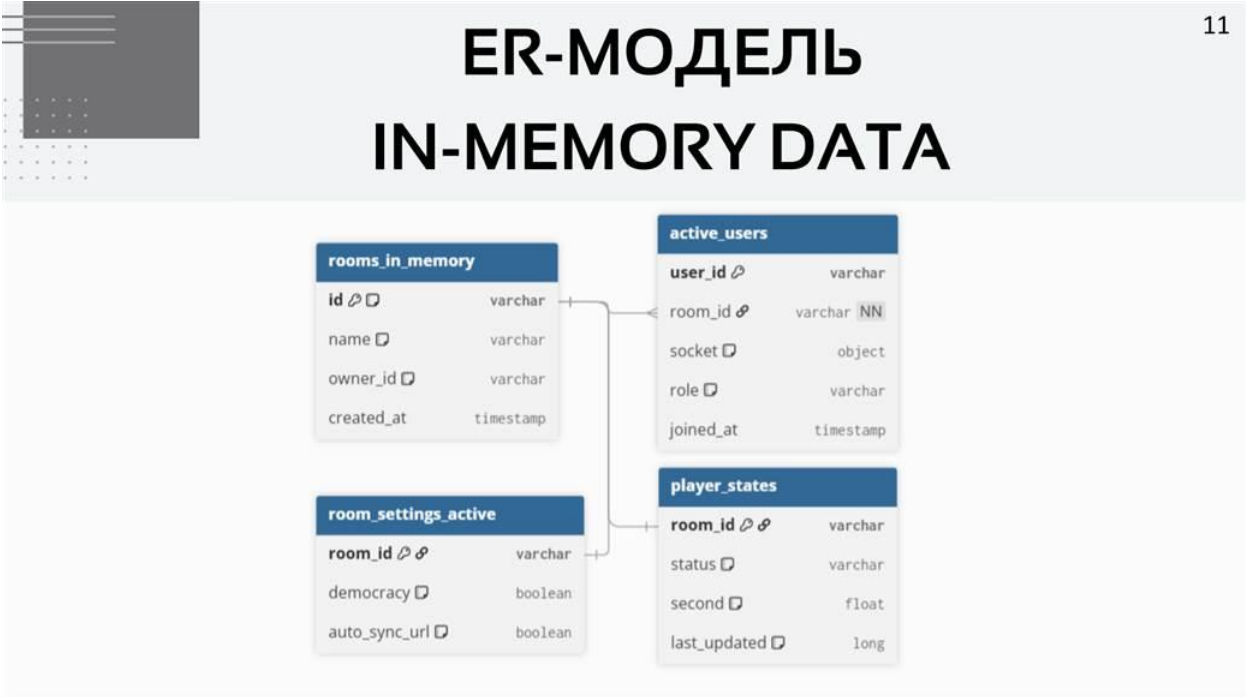


Рисунок В.11 - Слайд 11

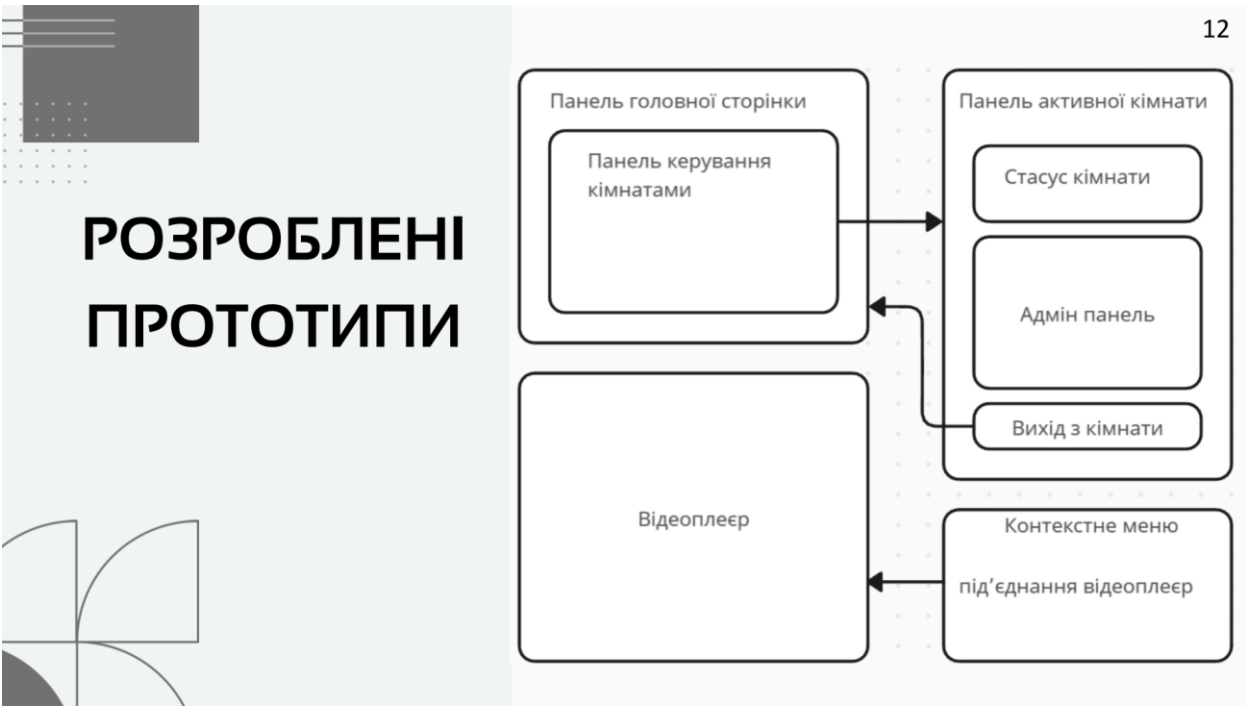


Рисунок В.12 - Слайд 12

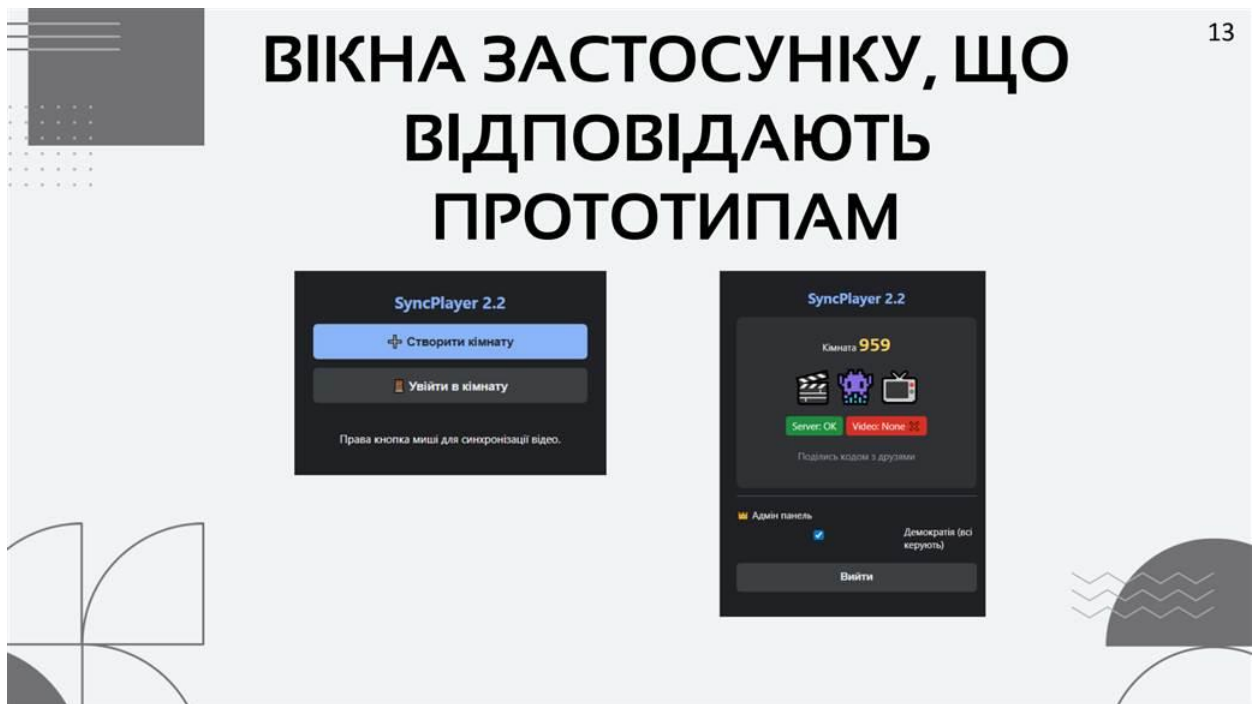


Рисунок В.13 - Слайд 13

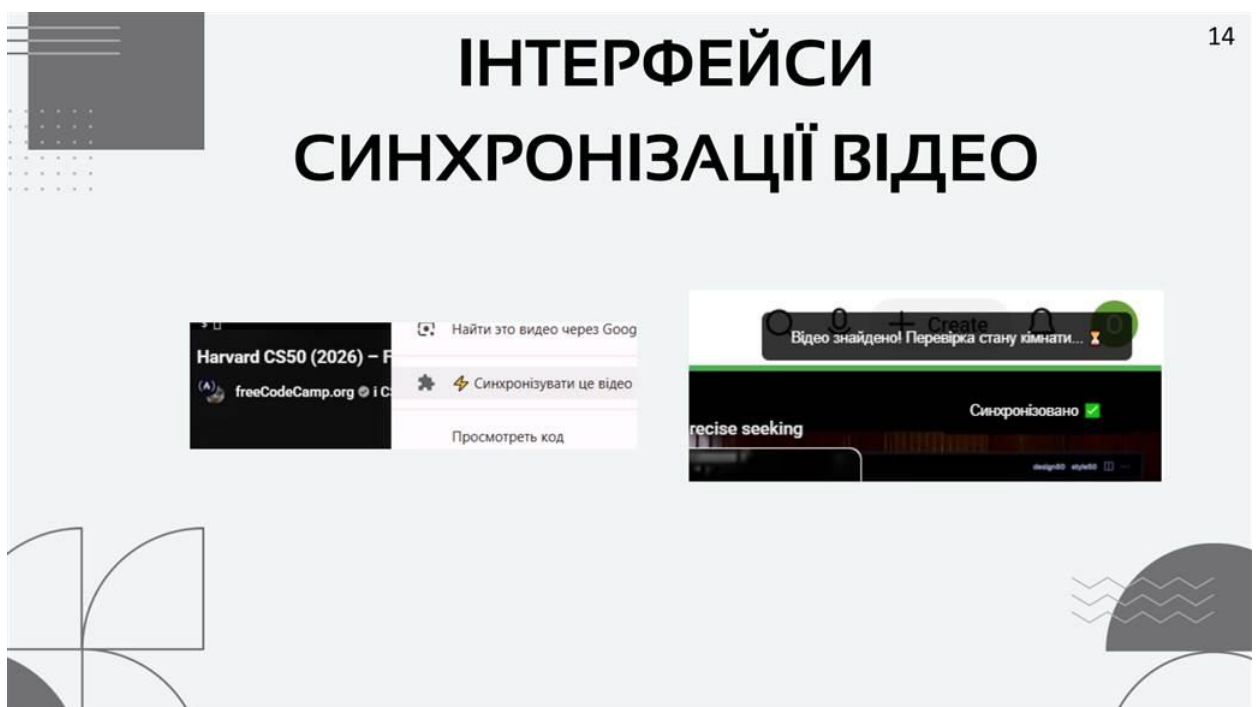


Рисунок В.14 - Слайд 14

ДЕМОНСТРАЦІЯ СИНХРОНІЗАЦІЇ



Рисунок В.15 - Слайд 15

ВИСНОВКИ

- Було проаналізовано існуюче програмне забезпечення для організації спільного перегляду медіа-контенту в реальному часі. Виявлено суттєві недоліки аналогів, зокрема деградацію якості відео через стиснення та обмеженість закритими екосистемами.
- Було визначено та обґрунтовано використання методів і засобів проєктування програмного забезпечення. Весь обраний набір інструментів, включаючи Node.js, WebSockets та стандарт Manifest V3, є сучасним, що гарантує мінімальну затримку синхронізації та можливість масштабування системи.
- Було сформульовано вимоги до системи, що стали основою для розробки архітектури та структури бази даних. До цих вимог було включено: створення віртуальних кімнат, миттєву синхронізацію станів плеєра, розмежування прав доступу (адміністрування) та підтримку будь-яких HTML5-відео.
- Зрештою, було розроблено програмне забезпечення для синхронізації споживання контенту та взаємодії користувачів. Система відповідає всім поставленим задачам, забезпечує збереження оригінальної якості медіа-потoku та має зручний інтерфейс, інтегрований безпосередньо у браузерне середовище.

Рисунок В.16 - Слайд 16