МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

КАФЕДРА ЕЛЕКТРОНІКИ, ІНФОРМАЦІЙНИХ СИСТЕМ ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Кваліфікаційна робота

другий (магістерський)
(рівень вищої освіти)

на тему **Особливості використання WebAssembly (Wasm) для підвищення продуктивності та безпеки Web-застосунків**

Виконав: студент 2 курсу, групи 8.1213-іпз-2
спеціальності 121 Інженерія програмного забезпечення
(код і назва спеціальності)
освітньої програми Програмне забезпечення систем
(код і назва освітньої програми)
В. В. Булигін
(ініціали та прізвище)
Керівник доцент, к.т.н. Н. П. Полякова
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)
Рецензент директор ТОВ «Дісітел»
П.О. Лютий
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)

Запоріжжя
2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

Кафедра електроніки, інформаційних систем та програмного забезпечення
Рівень вищої освіти _____другий (магістерський)_____
Спеціальність ___121_Інженерія програмного забезпечення_____
(код та назва)
Освітня програма ___Інженерія програмного забезпечення
(код та назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри_____Т.В. Критська_
“_02_” ___вересня_ 2024_року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Булигіну Володимиру Володимировичу
(прізвище, ім’я, по батькові)

1.Тема роботи Особливості використання WebAssembly (Wasm) для підвищення продуктивності та безпеки Web-застосунків

керівник роботи Полякова Наталія Петрівна, доцент, к.т.н.
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом ЗНУ від 28.02.2024р. № 361-е

2. Строк подання студентом кваліфікаційної роботи 30.11.2024
3. Вихідні дані магістерської роботи
- комплект нормативних документів ;
- технічне завдання до роботи.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
- огляд та збір літератури стосовно теми кваліфікаційної роботи;
- огляд та аналіз існуючих рішень та аналогів;
- дослідження особливостей використання WebAssembly (WASM) для підвищення продуктивності та безпеки Web-застосунків;
- створення програмного продукту та його опис;
- розробка висновків та рекомендацій для розробників.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
слайдів презентації

6. Консультанти розділів магістерської роботи

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

7. Дата видачі завдання 02.09.2024

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської роботи	Строк виконання етапів магістерської роботи	Примітка
1	Аналіз предметної області	11.09.24	виконано
2	Формулювання основної задачі дипломної роботи та узгодження її з науковим керівником	12.09.24	виконано
3	Аналіз існуючих методів рішення	13.09-14.09.24	виконано
4	Дослідження сучасних засобів підвищення продуктивності та безпеки	15.09-16.09.24	виконано
5	Узгодження подальших дій з науковим керівником	20.09.24	виконано
6	Проектування архітектури веб застосунку	22.09-30.09.24	виконано
7	Програмна реалізація веб застосунку	01.10-20.10.24	виконано
8	Аналіз продуктивності та безпеки розробленого застосунку	21.10.24	виконано
9	Представлення отриманих результатів науковому керівнику та узгодження плану подальшого дослідження	24.10.24	виконано
10	Перевірка працездатності веб застосунку	29.10-01.11.24	виконано
11	Оформлення звіту	12.11-29.11.24	виконано
12	Оформлення презентації. Отримання рецензій від опонентів.	02.12-05.12.24	виконано

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

Керівник роботи _____ Полякова Н.П.
(підпис) (прізвище та ініціали)

Нормоконтроль пройдено

Нормоконтролер _____ Скрипник І.А.
(підпис) (прізвище та ініціал

АНОТАЦІЯ

Сторінок: 97

Рисунків: 14

Джерел: 27

Булигін В. В. Особливості використання WebAssembly (Wasm) для підвищення продуктивності та безпеки Web-застосунків: кваліфікаційна робота магістра спеціальності 121 «Інженерія програмного забезпечення» / наук. керівник Н. П. Полякова. Запоріжжя: ЗНУ, 2024. 97с.

Мета і завдання дослідження полягають у вивченні можливостей технології WebAssembly (WASM) для підвищення продуктивності та безпеки веб-застосунків, а також у розробці веб-додатка з використанням WASM, який демонструє практичні переваги цієї технології у вирішенні зазначених проблем.

У ході дослідження було здійснено системний аналіз проблем продуктивності та безпеки сучасних веб-застосунків, детально вивчено основні особливості технології WebAssembly, зокрема її інтеграцію з JavaScript та іншими веб-технологіями. Для верифікації отриманих теоретичних результатів було реалізовано веб-застосунок на основі NestJS, React і TypeScript із впровадженням WebAssembly у функціонально критичних модулях. Проведено порівняльний аналіз продуктивності та безпеки традиційного методу з підходом, що використовує WASM.

Ключові слова: WebAssembly, веб-застосунок, ізольоване виконання, обчислювальна ефективність, NestJS, React.

ABSTRACT

Pages: 97

Drawings: 14

Sources: 27

Bulyhin V. V. Features of Using WebAssembly (Wasm) to Improve Performance and Security of Web Applications: qualification work of the master of specialty 121 «Software Engineering» / scientific supervisor N. P. Poliakova. Zaporizhzhia: ZNU, 2024. 97p.

The purpose and objectives of the study are to explore the possibilities of WebAssembly technology (WASM) to improve the performance and security of web applications, as well as to develop a web application using WASM, which demonstrates the practical advantages of this technology in solving these problems.

In the course of the research, a systematic analysis of the performance and security problems of modern web applications was carried out, the main features of WebAssembly technology, in particular its integration with JavaScript and other web technologies, were studied in detail. To verify the obtained theoretical results, a web application based on NestJS, React and TypeScript with the implementation of WebAssembly in functionally critical modules was implemented. A comparative analysis of the performance and security of the traditional method with the approach using WASM is carried out.

Keywords: WebAssembly, web application, isolated execution, computational efficiency, NestJS, React.

ЗМІСТ

ВСТУП 8
1 ДОСЛІДЖЕННЯ ПРОБЛЕМИ ПРОДУКТИВНОСТІ ТА БЕЗПЕКИ У ВЕБ-ЗАСТОСУНКАХ 13
1.1 Огляд проблеми продуктивності та безпеки у веб-застосунках .. 13
1.2 Технології підвищення продуктивності веб-застосунків 14
1.2.1 Технологія Asm.js 14
1.2.2 Оптимізація через кешування 16
1.2.3 Паралельне виконання задач у веб-застосунках 18
1.3 Засоби та практики підвищення безпеки веб-застосунків 20
1.3.1 Content Security Policy (CSP) 20
1.3.2 Шифрування даних під час передачі та зберігання 21
1.3.3 Ізоляція коду за допомогою пісочниці (Sandboxing) 22
1.3.4 Обфускація коду (reverse engineering) 23
1.4 WebAssembly (WASM) як засіб підвищення продуктивності та безпеки веб-застосунків 25
1.5 Постановка завдання 27
1.6 Висновки з розділу 1 29
2 ДОСЛІДЖЕННЯ ПРОГРАМНИХ ЗАСОБІВ РЕАЛІЗАЦІЇ 30
2.1 Огляд екосистеми WebAssembly (Wasm) 30
2.2 Огляд мов програмування для компіляції у WebAssembly 34
2.3 Особливості інтеграції WebAssembly у веб-застосунках 38
2.4 Аналіз сучасних технологій для розробки веб-застосунків 41
2.4.1 Фреймворк створення серверної частини 41
2.4.2 Бібліотека розробки користувацької частини 42
2.4.3 Додаткові технології для розробки веб-застосунків 44
2.5 Висновки до розділу 2 46
3 РОЗРОБКА ЗАСТОСУНКУ З ВИКОРИСТАННЯМ ТЕХНОЛОГІЇ WEB-ASSEMBLY 47

3.1 Опис предметної області 47
3.2 Архітектура застосунку 48
3.3 Програмна реалізація застосунку 50
3.3.1 Реалізація бази даних 50
3.3.2 Реалізація серверної частини 53
3.3.3 Реалізація користувацького інтерфейсу 60
3.4 Функціонал застосунку 66
3.5 Реалізація Wasm-модулю 69
3.5.1 Налаштування компіляційного середовища Emscripten 69
3.5.2 Розробка функціонального модуля на основі C++ 71
3.5.3 Компіляція функціонального модуля до WASM 73
3.6 Інтеграція Wasm у застосунок 75
3.7 Висновки до розділу 3 81
4 ДОСЛІДЖЕННЯ РЕЗУЛЬТАТІВ ВПЛИВУ WEB-ASSEMBLY НА ПРОДУКТИВНІСТЬ ТА БЕЗПЕКУ ВЕБ-ЗАСТОСУНКУ 83
4.1 Аналіз продуктивності веб-застосунку з впровадженням WASM 83
4.2 Результати оцінки безпеки розробленого веб-застосунку 88
4.3 Оцінка впливу WebAssembly на ефективність роботи застосунку 90
4.4 Практичне значення отриманих результатів 91
4.5 Висновки з розділу 4 93
ВИСНОВКИ 94
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 95

ВСТУП

Розвиток Web-технологій та їх вплив на сучасний Web-ландшафт не лише формує наше технологічне середовище, але й має вагомий вплив на наше повсякденне життя. З появою нових інструментів та платформ, інтернет середовище стає все більш динамічним та багатогранним, відкриваючи нові горизонти для спілкування, торгівлі, розваг та освіти. Проте, зростаюча складність і об'єм функціоналу веб-додатків ставить під сумнів їхню продуктивність та безпеку. Від операційної системи до браузера, веб-додатки мають велику кількість проміжних ланок, де можуть виникати проблеми швидкодії та безпеки.

З розвитком нашої залежності від веб-технологій зростає і вимога до ефективних та безпечних веб-додатків. Користувачі та підприємства очікують від веб-застосунків миттєвої реакції та надійної роботи, що ставить під сумнів продуктивність та безпеку наявних рішень. Паралельно з цим, загрози кібербезпеки постійно наростають, вимагаючи ефективних заходів захисту від вразливостей та атак.

Такий попит сприяє появі нових технологій і стандартів, спрямованих на розв'язання цих проблем. Одним з таких інноваційних рішень є WebAssembly (WASM), що відкриває нові можливості для оптимізації продуктивності та підвищення безпеки веб-додатків. Шляхом надання можливості виконання коду майже на рівні машинного коду безпосередньо у веб-браузері, WASM відкриває нові можливості для розробників та користувачів.

Спрощуючи процес розробки веб-додатків та полегшуючи їхнє розгортання, WASM дозволяє розробникам створювати більш швидкі, потужні та інноваційні застосунки. Завдяки вбудованій підтримці багатьох мов програмування, включаючи C++, Rust, і Go, WASM дозволяє використовувати найкращі інструменти для кожного проекту, забезпечуючи максимальну продуктивність.

Отже, дослідження особливостей використання WebAssembly для підвищення продуктивності та безпеки веб-додатків є актуальною темою, яка вимагає ретельного дослідження та вивчення. Розуміння можливостей та обмежень WASM дозволить сформувати рекомендації як використовувати цю технологію для створення швидких, надійних та безпечних веб-додатків.

Мета і завдання дослідження

Метою цієї роботи є вивчення та аналіз можливостей використання технології WebAssembly (WASM) з метою підвищення продуктивності та безпеки веб-застосунків. Зокрема, завданням дослідження є аналіз проблем, пов'язаних з продуктивністю та безпекою веб-додатків, вивчення основних можливостей та переваг WASM, дослідження практичних прикладів використання WASM у реальних проєктах, а також формулювання висновків та рекомендацій щодо використання WASM для оптимізації веб-додатків.

Об’єкт дослідження

Об’єктом дослідження є веб-застосунки, зокрема їх продуктивність та безпека.

Предмет дослідження

Предметом дослідження є особливості застосування WebAssembly (Wasm) та його вплив на покращення продуктивності та безпеки веб-застосунків.

Методи дослідження

Для досягнення поставленої мети використовуються наступні методи дослідження:

1. Аналіз літературних джерел та публікацій що стосуються WebAssembly та його впливу на продуктивність та безпеку web-застосунків.

2. Аналіз технологій та бібліотек для впровадження WASM у web-застосунки.
3. Проведення експериментальних випробувань та тестувань для оцінки продуктивності та безпеки WebAssembly у різних умовах використання.
4. Аналіз даних шляхом розробки аналітичних підходів, які враховують різноманітні аспекти, такі як розмір і склад програмного коду, типи операцій та умови виконання, для оцінки впливу використання WebAssembly на ефективність та безпеку веб-застосунків.
5. Застосування методів аналізу профілювання для виявлення вузьких місць продуктивності веб-застосунків і вдосконалення їх ефективності.
6. Розробка та демонстрація веб-застосунку для ілюстрації переваг та можливих обмежень використання WASM.

Наукова новизна одержаних результатів

Наукова новизна одержаних результатів дослідження полягає у тому, що вперше в рамках даної роботи детально проаналізовано потенціал використання технології WebAssembly (WASM) з метою підвищення продуктивності та забезпечення безпеки веб-застосунків. Розкрито інноваційні підходи та методи, спрямовані на оптимізацію використання WASM у веб-розробці, що в результаті дозволило досягти значного прискорення виконання коду та підвищити рівень захисту від потенційних загроз.

Практичне значення одержаних результатів

Отримані результати не лише розширюють теоретичний аспект веб-розробки, але й мають практичне застосування. Зроблені висновки можуть служити керівництвом для розробників під час вибору стратегій оптимізації веб-додатків та забезпечення їхньої безпеки. Вони відкривають нові можливості для створення ефективних та надійних веб-додатків, що можуть забезпечити користувачам безпеку та високу продуктивність. Ці результати

також є цінним внеском у веб-індустрію та можуть слугувати основою для подальших досліджень у цьому напрямку.

Апробація одержаних результатів

Результати дослідження були представлені на XVII науково-практичній конференції студентів, аспірантів, докторантів і молодих вчених Запорізького національного університету «Молода наука-2024» [26], а також на I Міжнародній науково-практичній конференції «Інженерні інновації та розбудова національної економіки» [27].

Глосарій

WebAssembly (або WASM) — відкритий web-стандарт байт-коду, який дозволяє виконувати високопродуктивний байт-код у веб-браузері.

Байт-код (англ. Bytecode) — інструкції, що представлені у вигляді байтів, які можуть бути виконані віртуальною машиною, у контексті WASM, байт-код - це низькорівнева мова інструкцій, яка використовується для виконання коду у веб-середовищі.

Продуктивність (англ. Performance) — міра ефективності виконання програмного забезпечення, що враховує швидкість обробки даних, час виконання інструкцій, а також обсяг використання обчислювальних ресурсів.

Безпека (англ. Security) — характеристика програмного забезпечення, яка гарантує захист даних та системи від потенційних загроз, таких як несанкціонований доступ або зловмисні дії.

Віртуальна машина (англ. Virtual Machine) — програмний або апаратний засіб, яке інтерпретує та виконує байт-код, створюючи середовище для виконання програм незалежно від апаратної архітектури.

Ізоляція пам'яті (англ. Memory Isolation) — підхід, що забезпечує захист пам'яті одного процесу від несанкціонованого доступу з боку іншого процесу.

LLVM — це модульна інфраструктура для розробки компіляторів, яка забезпечує набір інструментів і бібліотек для аналізу, оптимізації та генерації коду для різних мов програмування і процесорних архітектур.

Emscripten — інструментарій для компіляції програмного коду, написаного мовами програмування, такими як C і C++, у формат WebAssembly, а також для створення JavaScript-обгорток для взаємодії з цим кодом.

FFmpeg — це інструмент для роботи з мультимедіа, що забезпечує кодування, декодування, конвертацію, обробку та потокову передачу аудіо- і відеофайлів у різних форматах.

Node.js — відкрите середовище виконання JavaScript, яке ґрунтується на двигуні V8, розробленому Google для веб-браузера Chrome та інших, та дозволяє виконувати JavaScript-код на стороні сервера.

TypeScript — мова програмування, розроблена компанією Microsoft, яка надбудовує JavaScript і дозволяє розробникам створювати більш надійні програми, зручні для збереження та розширення, завдяки можливості явно вказувати типи даних.

Nest.js (або NestJS) — фреймворк для створення масштабованих та ефективних серверних додатків на Node.js, що базується на концепції модулів та дозволяє організувати код в окремі, автономні блоки, які можуть бути повторно використані в інших додатках.

React.js (або React) — бібліотека JavaScript, розроблена компанією Facebook, що дозволяє розробникам створювати високоякісні, швидкі та масштабовані веб-додатки, використовуючи компонентний підхід для створення користувацьких інтерфейсів.

PostgreSQL (або Postgres) — потужна, об'єктно-реляційна система управління базами даних з відкритим вихідним кодом, яка надає широкий набір функцій для зберігання та обробки структурованих даних, забезпечуючи надійність, масштабованість і гнучкість в роботі з даними.

1 ДОСЛІДЖЕННЯ ПРОБЛЕМИ ПРОДУКТИВНОСТІ ТА БЕЗПЕКИ У ВЕБ-ЗАСТОСУНКАХ

1.1 Огляд проблеми продуктивності та безпеки у веб-застосунках

Продуктивність і безпека веб-застосунків є одними з найважливіших аспектів сучасної веб-розробки. Еволюція веб-технологій сприяла значному прогресу у створенні інтерактивних, функціональних та швидких веб-застосунків. Однак разом із цим постають і значні виклики, пов’язані з обмеженнями продуктивності та загрозами безпеки.

Науковці та автори дослідження [1] що зі збільшенням складності веб-додатків суттєво зростає навантаження як на браузери, так і на сервери, особливо у випадках обробки великих обсягів даних або виконання ресурсомістких завдань. Це зумовлено не лише кількістю оброблюваних даних, але й необхідністю забезпечення високої інтерактивності, яка стає ключовою вимогою сучасних користувачів.

Традиційні веб-технології, такі як JavaScript, хоча й зазнали значної еволюції, мають обмеження у продуктивності, зумовлені інтерпретованою природою мови та особливостями її виконання у браузері [2]. Наприклад, при виконанні складних обчислювальних задач або обробці великомасштабних даних, JavaScript може демонструвати затримки, що знижують ефективність роботи додатків. Як зазначає Хьюго Ді Франческо у своїй праці [3], оптимізація продуктивності веб-додатків вимагає не лише вдосконалення алгоритмів, але й впровадження нових архітектурних рішень, здатних ефективно використовувати апаратні ресурси.

Однак, разом із зростанням вимог до продуктивності, набуває особливого значення і питання безпеки веб-додатків. Захист від різноманітних загроз кібербезпеки, таких як атаки на захист, вразливості програмного забезпечення та крадіжки даних, є невід'ємною складовою розробки та експлуатації веб-додатків[4]. Недоліки в забезпеченні безпеки можуть

призвести до серйозних наслідків, включаючи втрату конфіденційності та цілісності даних користувачів, а також негативний вплив на репутацію та довіру до веб-додатків.

Зокрема, впровадження механізму "пісочниці" (sandboxing) є одним із найефективніших засобів захисту. Пісочниця ізолює виконуваний код у межах обмеженого середовища, що запобігає доступу до критично важливих системних ресурсів або втручанню у роботу інших частин додатка. Згідно з дослідженням Срівастави та Шарми [5], пісочниці значно знижують ризик виконання шкідливого коду або експлуатації вразливостей в додатках, мінімізуючи таким чином можливість несанкціонованого доступу до даних. Використання пісочниць є особливо актуальним для веб-браузерів, де цей підхід захищає користувачів від шкідливих скриптів та вразливостей у плагінах.

Таким чином, сучасні виклики продуктивності та безпеки у веб-додатках є багатогранними та потребують комплексного підходу до їх вирішення. Зокрема, важливими є пошук нових технологій, таких як WebAssembly, що дозволяють оптимізувати продуктивність обчислень, а також впровадження ефективних засобів захисту, які забезпечують баланс між швидкодією та безпекою [6]. Крім того, вирішення цих завдань вимагає тісної інтеграції інноваційних підходів із сучасними інструментами розробки, що дозволяє створювати надійні, масштабовані та конкурентоспроможні веб-рішення.

1.2 Технології підвищення продуктивності веб-застосунків

1.2.1 Технологія Asm.js

Asm.js (або AsmJs) — це строго типізована підмножина мови JavaScript, яка використовується для створення високопродуктивних веб-застосунків. Розроблена компанією Mozilla, asm.js має на меті покращити продуктивність веб-застосунків, шляхом використання оптимізацій для виконання складних операцій у веб-браузері.

Як оптимізаційний шар, ця технологія перетворює вихідний код на JavaScript у форму, яка може бути ефективно виконана браузером навіть за відсутності прямого використання специфікацій Asm.js у вихідному коді. Це досягається завдяки використанню строго типізованої підмножини JavaScript, що забезпечує передбачувану обробку і полегшує компіляцію в машинний код для браузера (див. Рис 1).

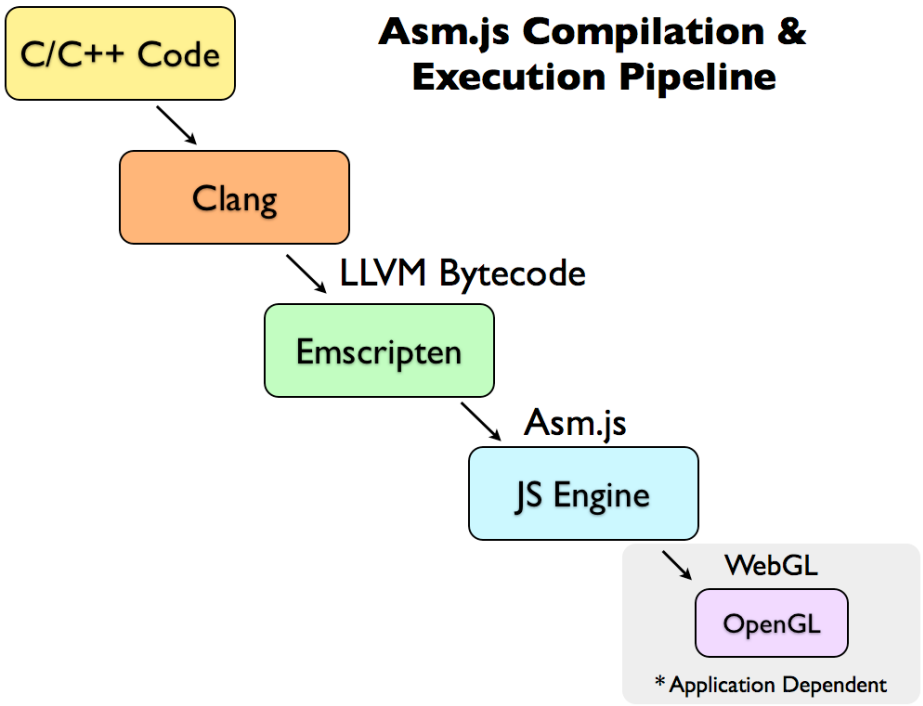


Рисунок 1 — Приклад процесу компіляції та виклику asm.js

Процес компіляції починається з вихідного коду, написаного на таких мовах, як C або C++. Цей код компілюється за допомогою Clang, популярного компілятора, що є частиною інфраструктури LLVM. Після компіляції C/C++ коду він перетворюється у LLVM Bytecode, проміжний формат, що використовується LLVM для подальшої оптимізації та обробки.

Далі процес передається на Emscripten, спеціальний компілятор, який перетворює цей байт-код у JavaScript, точніше — у Asm.js. Asm.js, будучи строго типізованою підмножиною JavaScript, дозволяє браузеру оптимізувати виконання цього коду завдяки більш простій та передбачуваній структурі.

Після цього JS Engine браузера (наприклад, V8 у Chrome чи SpiderMonkey у Firefox) інтерпретує або компілює цей код безпосередньо у машинний код, оптимізуючи виконання. Важливою частиною цього процесу є підтримка графіки, яка може використовувати WebGL або OpenGL для відображення 3D-графіки та складних візуалізацій у браузері. Для таких застосунків, як ігри або графічні симуляції, інтеграція з WebGL є важливою частиною цього процесу, забезпечуючи високопродуктивне відтворення контенту.

Таким чином, процес компіляції з C/C++ у Asm.js через Clang та Emscripten дозволяє веб-застосункам досягати набагато кращих показників продуктивності, зокрема для обчислювально інтенсивних завдань.Ще однією перевагою Asm.js є легкість відлагодження. Оскільки Asm.js використовує JavaScript синтаксис, відлагодження коду може бути плюсом для тих, хто вже знайомий з інструментами відлагодження JavaScript. Для невеликих проєктів де продуктивність не є критичною, Asm.js може бути швидшим способом досягнення певного рівня оптимізації без необхідності вивчення нового інструментарію.

Ураховуючи ці фактори, Asm.js може бути привабливим варіантом для деяких проектів, де зручність розробки та сумісність з існуючою JavaScript базою є ключовими факторами.

1.2.2 Оптимізація через кешування

Кешування є одним із найбільш ефективних і широко використовуваних методів оптимізації продуктивності веб-додатків. Воно дозволяє зменшити частоту звернень до серверів, зберігаючи часто використовувані ресурси або дані в тимчасових сховищах, що скорочує час доступу до них при повторних запитах [7]. У результаті цього знижується затримка завантаження веб-сторінок, що сприяє підвищенню загальної швидкодії та ефективності веб-додатків.

Одним з основних завдань кешування є мінімізація обробки і передачі даних між клієнтом і сервером. Це досягається через тимчасове збереження даних у різних точках мережі або локально на пристрої користувача, що дозволяє повторно використовувати ці дані без необхідності здійснювати запити до віддалених серверів або виконувати ресурсомісткі обчислення.

Основні типи кешування, що використовуються для покращення продуктивності веб-додатків, включають:

1. Кешування в браузері: збереження локальних копій файлів на пристрої користувача для швидкого доступу при повторних відвідуваннях.
2. Кешування на рівні мережі: використання інфраструктури CDN для розміщення ресурсів на географічно розподілених серверах, що мінімізує час передачі даних.
3. Серверне кешування: збереження результатів обчислень або запитів на сервері з метою зменшення кількості повторних запитів до баз даних або інших серверних ресурсів.
4. HTTP-заголовки для контролю кешування: використання заголовків, таких як Cache-Control і ETag, для керування політиками кешування та забезпечення актуальності даних.

Окрім зменшення затримок при завантаженні сторінок, кешування також дозволяє знизити навантаження на сервери, що забезпечує більш стабільну роботу додатків під час пікових навантажень. Це критично важливо для веб-додатків, які обслуговують велику кількість користувачів або мають складні динамічні компоненти.

Ефективно налаштовані механізми кешування забезпечують не тільки прискорення завантаження контенту, але й сприяють більш раціональному використанню мережевих і обчислювальних ресурсів. Проте, важливо враховувати потенційні проблеми, пов’язані з кешуванням, такі як ризик збереження застарілих даних або складність керування оновленням кешу. Тому належна конфігурація цих механізмів є ключовим аспектом при розробці продуктивних та масштабованих веб-додатків.

1.2.3 Паралельне виконання задач у веб-застосунках

Йоган Джерф Карльторп та Ерік Скоглунд зазначають у своїй роботі [8], що паралельне виконання завдань (багатопоточність) є ключовим механізмом для підвищення продуктивності та оптимізації обробки даних у веб-додатках. Цей підхід передбачає одночасну роботу кількох потоків або процесів, що дозволяє розподіляти навантаження між різними ядрами процесора. Така організація дає змогу ефективно виконувати складні обчислювальні операції, як-от обробка великих масивів даних, рендеринг графіки або асинхронна взаємодія з сервером, без блокування основного потоку.

Хоча JavaScript працює в одному потоці, сучасна веб-екосистема реалізує багатопоточність через Web Workers і асинхронне програмування. Web Workers дозволяють виконувати важкі обчислення у фоновому режимі, зберігаючи стабільність і плавність роботи основного інтерфейсу. Це особливо важливо для ресурсоємних операцій, таких як обробка великих обсягів даних чи складних графічних обчислень.

На рисунку 2 ілюструється принцип взаємодії між основним потоком JavaScript та Web Worker:

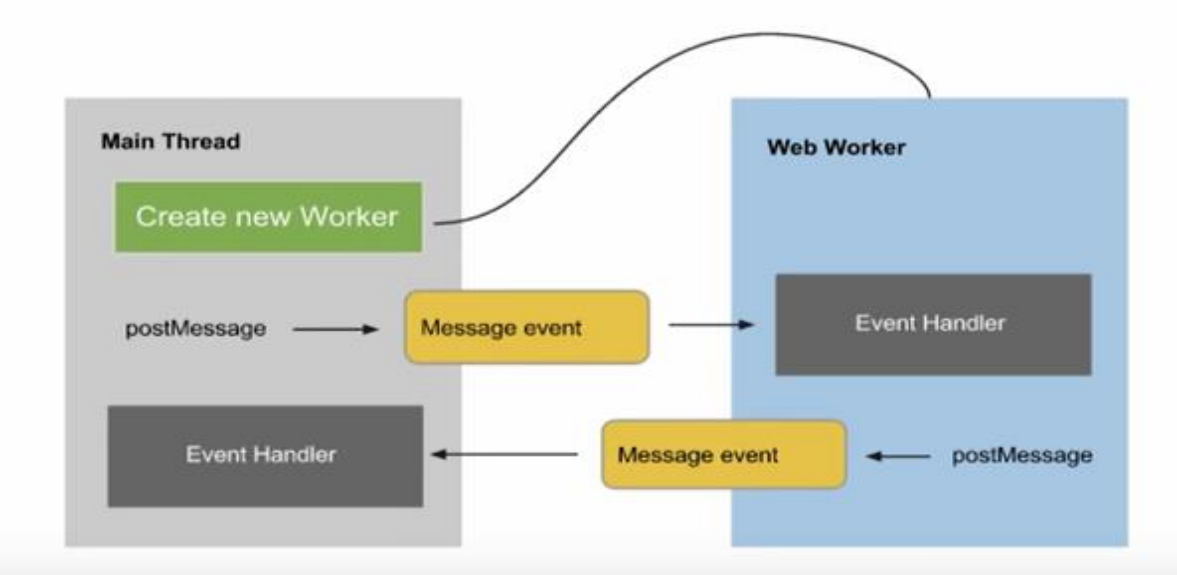


Рисунок 2 — Принцип роботи WebWorker

Як видно, основний потік створює Web Worker і передає йому дані через метод postMessage. Web Worker отримує ці дані, обробляє їх і надсилає результат назад у головний потік, також використовуючи postMessage. Цей підхід забезпечує безперервність роботи додатка без блокування основного потоку.

Основні переваги паралельного виконання задач у веб-додатках включають:

1. Підвищення продуктивності: одночасне виконання кількох задач дозволяє суттєво скоротити час їх обробки, що є важливим для додатків з високим навантаженням.
2. Плавність роботи інтерфейсу: основний потік, відповідальний за роботу з інтерфейсом користувача, залишається вільним для реагування на дії користувача, поки інші задачі обробляються у фоновому режимі.
3. Ефективність використання ресурсів: багатоядерні процесори сучасних пристроїв дозволяють розподіляти навантаження між кількома ядрами, що підвищує загальну ефективність додатків.

Однак, використання багатопоточності також несе певні виклики. Серед них:

1. Складність управління: розробка паралельних алгоритмів вимагає ретельного планування, особливо щодо синхронізації потоків і управління ресурсами, щоб уникнути конфліктів та втрати даних.
2. Витрати на обробку: створення та управління потоками споживає додаткові ресурси системи, тому потрібно збалансувати кількість потоків із доступними ресурсами для досягнення оптимальної продуктивності.

Таким чином, паралельне виконання задач є потужним засобом для підвищення продуктивності сучасних веб-додатків, який забезпечує швидкість і ефективність обробки даних, зберігаючи при цьому високу чутливість інтерфейсу до дій користувача.

1.3 Засоби та практики підвищення безпеки веб-застосунків

Забезпечення безпеки веб-додатків є важливим аспектом сучасної розробки програмного забезпечення, оскільки веб-застосунки часто виступають об'єктом складних та цілеспрямованих атак, таких як міжсайтове скриптування (XSS), SQL-ін'єкції та компрометація цілісності даних. У відповідь на ці загрози було розроблено широкий спектр технологій, які спрямовані на мінімізацію ризиків, підвищення стійкості до атак та захист конфіденційності користувачів. Нижче розглянуто ключові технології, що є ефективними засобами для посилення безпеки веб-додатків.

1.3.1 Content Security Policy (CSP)

Content Security Policy (CSP) — це механізм безпеки, що дозволяє розробникам обмежити ресурси, які можуть бути завантажені та виконані веб-сторінкою. CSP запобігає атакам, таким як міжсайтовий скриптинг (XSS), забороняючи виконання неперевіреного коду або підключення сторонніх скриптів. Ця технологія визначає набір політик, які контролюють, які типи контенту можуть бути завантажені браузером, включаючи скрипти, стилі, шрифти та інші ресурси [9].

Важливою частиною впровадження CSP є можливість створення списку дозволених доменів, з яких дозволено завантажувати ресурси. Таким чином, навіть якщо зловмисник вбудує шкідливий код у веб-додаток, він не зможе бути виконаний, оскільки не буде відповідати політикам CSP. Це значно підвищує безпеку веб-додатків, знижуючи ризик виконання зловмисного коду.

Політики CSP зазвичай включають:

1. Обмеження на завантаження JavaScript, стилів (CSS), шрифтів та інших ресурсів лише з надійних джерел.
2. Використання механізму nonce або хешування для дозволу виконання лише конкретного скрипта.

3. Блокування використання вбудованого JavaScript та стилів, що значно знижує ризик експлуатації вразливостей через XSS.

CSP також дозволяє відслідковувати та фіксувати порушення політик шляхом відправлення звітів на сервер розробника, що дозволяє оперативно реагувати на потенційні загрози. Таким чином, ця технологія не тільки запобігає можливості виконання небезпечних сценаріїв, але й допомагає у виявленні та усуненні вразливостей у додатку.

1.3.2 Шифрування даних під час передачі та зберігання

Шифрування даних є одним із ключових інструментів для забезпечення безпеки інформації як під час її передачі через мережі, так і при зберіганні на фізичних або віртуальних носіях. Його застосування дозволяє гарантувати конфіденційність і цілісність даних, мінімізуючи ризики несанкціонованого доступу або модифікації. Однак впровадження шифрування також пов'язане з певними викликами, які потрібно враховувати для забезпечення належного рівня безпеки [10].

Передача даних через мережу, зокрема Інтернет, супроводжується ризиками, такими як атаки перехоплення трафіку, зокрема "людина посередині" (man-in-the-middle). Для забезпечення захищеного з'єднання між клієнтами і серверами використовується протокол TLS (Transport Layer Security), який шифрує дані під час їхньої передачі, запобігаючи несанкціонованому доступу та модифікації. Цей протокол лежить в основі HTTPS, що є стандартним засобом шифрування веб-трафіку.

Додатково для захисту даних у публічних мережах використовується технологія VPN (Virtual Private Network), яка створює захищений канал передачі інформації, гарантуючи її конфіденційність і захист навіть за умов небезпечного середовища. У разі потреби в максимальній конфіденційності застосовується шифрування "від кінця до кінця" (E2EE), при якому дані шифруються на стороні відправника і можуть бути розшифровані лише отримувачем, виключаючи можливість доступу навіть на проміжних вузлах.

Захист збережених даних також вимагає застосування сучасних підходів до шифрування. Бази даних можуть шифруватися як повністю, так і частково, наприклад, для конфіденційної інформації (паролів або фінансових даних), використовуючи механізми на зразок Transparent Data Encryption (TDE). Додатково шифрування файлових систем дозволяє захищати дані на фізичних носіях, унеможливлюючи доступ до них у разі несанкціонованого фізичного доступу до пристроїв. Усі ці заходи мають ефективно доповнюватися системами керування криптографічними ключами, адже ненадійне їх зберігання може скомпрометувати систему.

Шифрування даних є ключовим для забезпечення конфіденційності та цілісності інформації під час її передачі чи зберігання. Водночас воно створює виклики, такі як збільшене навантаження на обчислювальні ресурси та складнощі в управлінні ключами, що потребують ретельного планування під час його впровадження.

1.3.3 Ізоляція коду за допомогою пісочниці (Sandboxing)

Sandboxing — це технологія ізоляції програмного забезпечення, яка забезпечує виконання коду в обмеженому та контрольованому середовищі з мінімальними привілеями доступу до системних ресурсів. Ця технологія спрямована на захист основної інфраструктури додатка від потенційно небезпечного або ненадійного коду шляхом запровадження суворих обмежень на операції, які можуть виконуватись цим кодом [11].

Ізольоване середовище виконання коду є ефективним методом підвищення безпеки. Воно обмежує доступ до критичних компонентів системи, запобігаючи експлуатації вразливостей та поширенню загроз. Це досягається через ізоляцію процесів, обмеження привілеїв і моніторинг виконання.

Ізоляція дозволяє виконувати кожен процес у відокремленому середовищі, унеможливлюючи несанкціонований доступ до даних. Обмежені права забезпечують виконання коду лише в межах необхідних привілеїв,

знижуючи ризики доступу до критичних ресурсів. Моніторинг допомагає виявляти та блокувати потенційно небезпечні дії, забезпечуючи дотримання політик безпеки.

Цей підхід активно використовується у веб-браузерах і веб-додатках. Наприклад, ізоляція вкладок у браузерах через незалежні процеси запобігає поширенню шкідливого контенту, забезпечуючи безпечну роботу навіть за умов компрометації окремих елементів.

Пісочниці забезпечують мінімізацію ризиків шкідливого коду шляхом ізоляції, яка обмежує його вплив лише межами ізольованого середовища. Вони також гарантують безпечне виконання ненадійних плагінів та скриптів, знижуючи ризик, пов’язаний із вразливими компонентами. Гнучкість у налаштуванні прав доступу дозволяє розробникам обмежувати ресурси, доступні для виконуваного коду, відповідно до вимог безпеки.

Основними недоліками є додаткові витрати на ресурси, оскільки ізольовані процеси вимагають більше обчислювальної потужності, що може впливати на продуктивність. Крім того, обмеження доступу до системних ресурсів може зменшувати функціональність, що потребує ретельного налаштування середовища для підтримки необхідних можливостей.

Sandboxing є ефективною технологією для підвищення безпеки веб-додатків шляхом ізоляції виконання потенційно небезпечного коду. Завдяки обмеженню привілеїв та контролю за виконанням операцій, пісочниці значно знижують ризики компрометації системи та забезпечують захист від небажаних впливів шкідливого програмного забезпечення.

1.3.4 Обфускація коду (reverse engineering)

Обфускація коду (або *code obfuscation*) — це процес навмисної модифікації вихідного коду програми з метою ускладнення його розуміння для людини або автоматичних засобів аналізу. Основною метою обфускації є захист коду від зворотньої розробки, тобто спроб відтворити або відновити

вихідний код за допомогою аналізу виконуваного файлу або компільованого бінарного коду [12].

Зворотна розробка є одним із головних ризиків безпеки веб-додатків, особливо у випадках, коли програма містить конфіденційну логіку, алгоритми або дані. Зловмисники можуть спробувати дослідити та модифікувати код для компрометації безпеки додатка, що може призвести до крадіжки інтелектуальної власності, отримання несанкціонованого доступу або експлуатації вразливостей.

Щоб ускладнити процес зворотної розробки, використовуються різноманітні методи обфускації, серед яких найбільш поширеними є:

1. Заміна імен: Заміна логічних назв змінних, функцій та класів на неінформативні набори символів ускладнює аналіз коду.
2. Реорганізація структури: Використання складних структур, таких як вкладені цикли або умовні оператори, знижує зрозумілість коду, не змінюючи його функціональності.
3. Динамічне генерування: Генерація частин коду під час виконання програми робить зворотний аналіз складнішим.
4. Шифрування коду: Захист шляхом шифрування частин програми, які розшифровуються лише під час виконання, запобігає прямому доступу до вихідного коду.

Переваги обфускації коду:

1. Захист інтелектуальної власності: Обфускація запобігає розкриттю важливих алгоритмів, комерційної логіки або бізнес-процесів, які можуть бути використані конкурентами або зловмисниками.
2. Зменшення ризиків експлуатації вразливостей: Ускладнений для аналізу код знижує ймовірність того, що зловмисники знайдуть у ньому вразливості, які можуть бути використані для атак.
3. Посилення захисту комерційних продуктів: Обфускація коду дозволяє захистити програмні продукти від нелегального копіювання або модифікації.

Серед недоліків обфускації коду можна виділити:

1. Ускладнення налагодження та підтримки: Через модифікації та ускладнення коду його стає важче налагоджувати або розширювати. Це може створювати труднощі для команди розробників, особливо під час підтримки великомасштабних проектів.

Обфускація коду є важливим інструментом захисту веб-додатків від зворотньої розробки та компрометації безпеки. Завдяки різним методам ускладнення коду, розробники можуть ефективно захистити свої програми від несанкціонованого аналізу, крадіжки інтелектуальної власності або експлуатації вразливостей. Однак цей процес має бути збалансованим з урахуванням потенційних втрат продуктивності та складнощів у підтримці коду.

1.4 WebAssembly (WASM) як засіб підвищення продуктивності та безпеки веб-застосунків

З ростом вимог та переходом інфраструктури до веб-додатків виникла необхідність у технологіях, які здатні забезпечити високу продуктивність та безпеку при виконанні складних обчислювальних задач у браузері. Традиційні методи, такі як використання JavaScript, хоча і залишаються основою веб-розробки, мають свої обмеження, особливо у сфері обчислювальної ефективності та безпеки [13]. Для вирішення цих проблем було розроблено новий підхід — двійковий формат виконання, що дозволяє виконувати код з високою швидкістю та в ізольованому середовищі. Цей підхід отримав назву WebAssembly (WASM) і став важливим інструментом для підвищення продуктивності та забезпечення безпеки веб-додатків.

WebAssembly (WASM) представляє собою двійковий формат інструкцій, що дозволяє виконувати код майже на нативному рівні продуктивності [14]. Використовуючи компіляцію з мов програмування високого рівня (C, C++, Rust та інші), WASM дає змогу досягти значного

приросту швидкості виконання у порівнянні з JavaScript, що є інтерпретованою мовою. Це робить WASM ефективним для додатків, які потребують значних обчислювальних ресурсів, зокрема для задач, пов’язаних з обробкою мультимедійних даних, моделюванням фізичних процесів, 3D-рендерингом, а також у контексті високопродуктивних ігор та наукових обчислень.

З точки зору безпеки, WASM забезпечує ізольоване середовище виконання (sandbox), що обмежує можливість доступу до системних ресурсів користувача, таких як файлові системи та оперативна пам'ять [15]. Це значно знижує ризик експлуатації вразливостей і захищає від потенційно шкідливого коду. Модель WASM передбачає мінімізацію прав доступу, надаючи лише необхідні привілеї для виконання задач, що додає додатковий рівень захисту для веб-додатків.

Важливою особливістю WebAssembly з точки зору безпеки є те, що формат не передбачає прямого виконання вихідного коду, а замість цього використовує компіляцію у двійковий формат, який інтерпретується браузером. Це ускладнює процес зворотної інженерії, оскільки бінарний код важче проаналізувати або модифікувати, що додає додатковий рівень захисту від втручання зловмисників.

Ця технологія має декілька значних переваг:

1. Висока продуктивність: Завдяки компіляції до машинного коду, Wasm забезпечує швидкість виконання, близьку до нативної, що є критично важливим для обчислювально інтенсивних задач.
2. Ізоляція виконання (sandboxing): Виконання коду в ізольованому середовищі мінімізує можливість компрометації системи користувача, обмежуючи доступ до системних ресурсів.
3. Модель обмежених прав доступу: WASM забезпечує чітке розмежування прав доступу, надаючи кодові лише мінімально необхідні дозволи, що знижує ризик ескалації привілеїв і кібератак.

4. Сумісність із веб-технологіями: WASM інтегрується з JavaScript, що дозволяє використовувати його для важких обчислень, а JavaScript — для роботи з інтерфейсом користувача.
5. Захист від зворотної інженерії: Компіляція в двійковий формат ускладнює аналіз та модифікацію коду, що захищає від зловмисників.

Попри значні переваги, WASM має також низку обмежень, які слід враховувати при його використанні у реальних проєктах:

1. Обмежений доступ до системних API: WASM функціонує в обмеженому середовищі і не має прямого доступу до деяких системних ресурсів, що ускладнює його використання для завдань, що вимагають тісної інтеграції з операційною системою.
2. Складність розробки: Використання цієї технології вимагає знань низькорівневих мов, таких як C або Rust, що створює додаткові труднощі для менш досвідчених розробників.
3. Збільшення розміру бінарних файлів: Компіляція у бінарний формат може збільшувати розмір додатку, що впливає на швидкість завантаження за умов слабкого інтернет-з’єднання.

Загалом, WebAssembly є перспективною технологією, яка здатна суттєво розширити можливості веб-розробки, особливо у сценаріях, що вимагають високої продуктивності та безпеки. Подальший розвиток інфраструктури та розширення функціональності WASM, зокрема за рахунок таких ініціатив, як WebAssembly System Interface (WASI), можуть зробити цю технологію універсальним інструментом не лише для клієнтської, але й для серверної частини веб-додатків.

1.5 Постановка завдання

Метою кваліфікаційної роботи є вивчення особливостей використання технології WebAssembly (WASM) для підвищення продуктивності та безпеки веб-додатків. Зокрема, в рамках дослідження планується визначити

можливості та обмеження використання WASM у контексті веб-розробки, а також розробити рекомендації щодо оптимального використання цієї технології.

Для досягнення покладеної мети необхідно вирішити декілька наступних завдань:

1. Ретельний аналіз існуючих методів та технологій, що використовуються для підвищення продуктивності та забезпечення безпеки веб-додатків. Це включає вивчення літературних джерел, аналіз практичних реалізацій та огляд інструментів, які використовуються у сфері веб-розробки.
2. Вивчення технології WebAssembly (WASM) у контексті її можливостей та обмежень. Під час дослідження буде звернута увага на те, як WASM може бути використаний для оптимізації продуктивності та підвищення рівня безпеки веб-додатків.
3. Розробити методи та стратегії оптимізації використання WASM з метою досягнення найвищої продуктивності та безпеки веб-додатків..
4. Проведення експериментальних досліджень для оцінки впливу використання WASM на продуктивність та безпеку веб-додатків. Це включає збір та аналіз даних про швидкодію та безпеку додатку з використанням WASM порівняно з аналогічними рішеннями без WASM.
5. Проведення тестування системи з використанням для підтвердження ефективності та коректності її роботи.
6. Висунення висновків та формулювання рекомендацій щодо використання технології WebAssembly (WASM) для підвищення продуктивності та безпеки веб-додатків. Висновки будуть засновані на отриманих результатів експериментів та аналізі літературних джерел.

1.6 Висновки з розділу 1

1. Проведено аналіз та оглянуті головні проблеми, що виникають у процесі створення веб-додатків з врахуванням продуктивності та безпеки. Серед них варто відзначити складність масштабування для оптимального управління ресурсами та кодом при зростанні обсягу даних та користувачів. Також необхідно приділити особливу увагу захисту конфіденційної інформації користувачів, оскільки це критично важливий аспект для веб-додатків. Забезпечення високої продуктивності та ефективності роботи додатків є ключовим завданням для задоволення потреб користувачів та покращення їхнього досвіду використання.
2. Проаналізовано різноманітні технології та практики для підвищення продуктивності та безпеки. Були визначені переваги та недоліки кожної з цих технологій, а також їх відмінності та особливості застосування в контексті веб-розробки.
3. Проведено ретельний огляд стандарту WASM та визначено особливості роботи цього інструменту. Також проведено аналіз цієї технології та визначено як саме вона може підвищити продуктивність та безпеку веб-додатка.
4. Визначена постановка завдання дослідження, яке включає аналіз існуючих методів для підвищення продуктивності та безпеки, вибір технологій, розробку методів та стратегій оптимізації, реалізацію функціональності та тестування системи, апробацію результатів.

2 ДОСЛІДЖЕННЯ ПРОГРАМНИХ ЗАСОБІВ РЕАЛІЗАЦІЇ

2.1 Огляд екосистеми WebAssembly (Wasm)

Стандартизація WebAssembly у 2017 році стала результатом співпраці компаній Google, Mozilla, Microsoft та Apple, які прагнули подолати обмеження JavaScript, особливо у сфері обчислювально інтенсивних завдань. Ця технологія націлена на виконання низькорівневого байт-коду, який є оптимізованим для різних апаратних платформ і водночас забезпечує високу портативність і безпеку [16].

WebAssembly базується на концепції, що код, написаний на мовах програмування високого рівня (C, C++, Rust), компілюється у проміжний байт-код. Цей байт-код є компактним, добре оптимізованим і може швидко інтерпретуватися або виконуватися в браузері за допомогою відповідного середовища виконання. Завдяки цьому вдається уникнути тривалих процесів компіляції під кожну платформу окремо.

В основі роботи WebAssembly лежить процес, що складається з кількох етапів:

1. Написання програми на підтримуваній мові: Наприклад, на C++ чи Rust розробник створює код програми, що виконує необхідні обчислення або операції.
2. Крос-компіляція у байт-код WebAssembly: Код програми компілюється у wasm файл — компактний і оптимізований байт-код, що може бути виконаний у браузері.
3. Інтеграція з JavaScript: Для взаємодії між WebAssembly та іншими частинами веб-застосунку використовується JavaScript API. Це дозволяє викликати функції WebAssembly з JavaScript і передавати дані між ними.
4. Виконання у середовищі браузера: Браузер завантажує .wasm файл і передає його у модуль виконання, який інтерпретує чи компілює код у машинні інструкції платформи.

На рисунку 3 наведено схему роботи WebAssembly у типовому веб-застосунку.

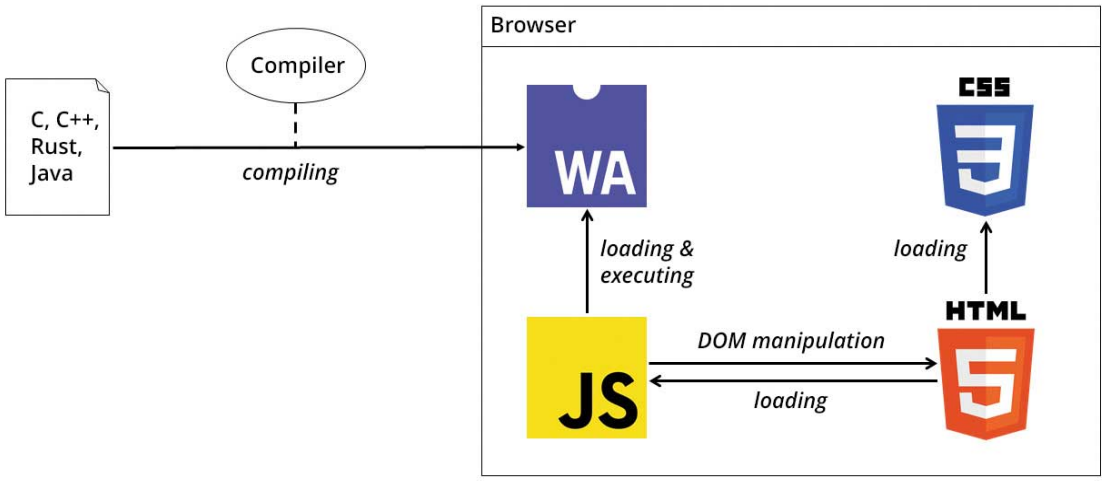


Рисунок 3 — Схема роботи WebAssembly

Архітектурна реалізація WebAssembly характеризується комплексним підходом до організації виконання низькорівневого коду у веб-середовищі. Фундаментальною основою технології є бінарний формат представлення програмного коду, що забезпечує компактність та ефективність його передачі через мережу. Бінарний формат оптимізований для швидкого декодування та валідації, що є критичним фактором у контексті веб-застосунків.

Визначальною характеристикою WebAssembly є інтеграція строгої статичної системи типізації, яка підтримує базові числові типи даних: i32, i64 для цілочисельних значень та f32, f64 для чисел з плаваючою комою. Така типізація гарантує безпеку та передбачуваність виконання програмного коду, що особливо важливо для високопродуктивних обчислень.

Як зазначають Войдан Кьорвезироски та Соня Филипоска [17] суттєвою перевагою технології є реалізація концепції лінійної пам'яті, яка представляє собою континуальний масив байтів з можливістю динамічного розширення. Цей підхід забезпечує ефективне управління пам'яттю, характерне для мов

програмування низького рівня, зберігаючи при цьому безпеку веб-платформи (див. Рис. 4).

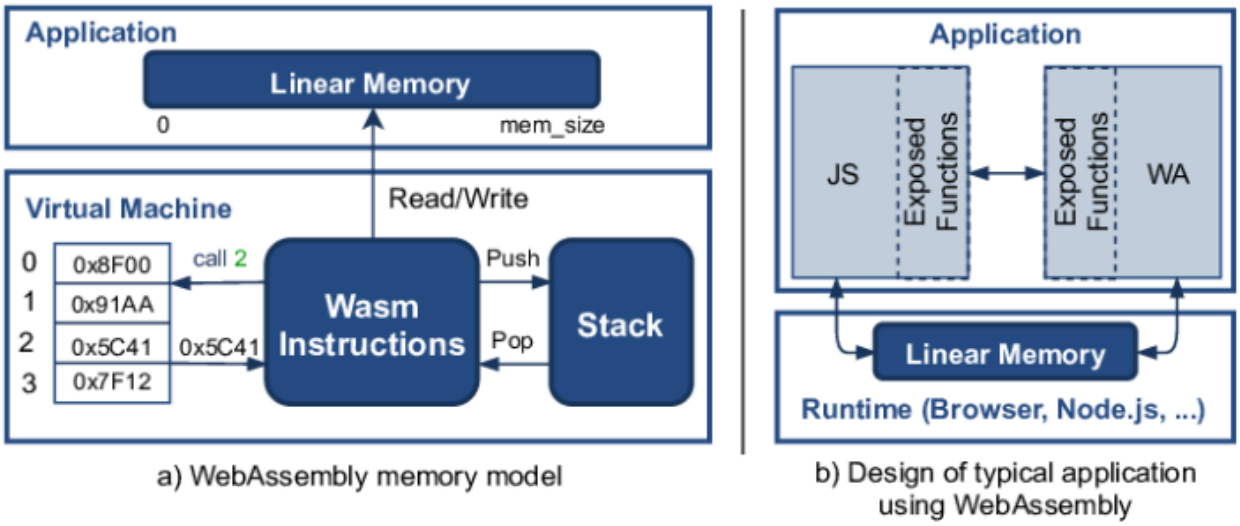


Рисунок 4 — Модель пам'яті WebAssembly та дизайн взаємодії з JavaScript

Модульна архітектура WebAssembly реалізує принцип інкапсуляції через механізм ізольованих модулів, кожен з яких містить власний простір імен, таблиці функцій та сегмент лінійної пам'яті. Така організація сприяє ефективному управлінню ресурсами та забезпечує можливість незалежної компіляції окремих компонентів системи.

Механізм виконання WebAssembly базується на концепції потокової компіляції, що дозволяє розпочинати процес компіляції до повного завантаження модуля. Це суттєво оптимізує час від початку завантаження до виконання програмного коду. Детермінований час виконання операцій досягається завдяки простій моделі пам'яті та відсутності автоматичного збирача сміття, що є критичним для систем реального часу.

Безпека виконання забезпечується комплексом механізмів, включаючи ізольоване середовище виконання, статичну валідацію коду та контрольований доступ до пам'яті. Ці механізми гарантують захист хост-системи від потенційно небезпечних операцій та несанкціонованого доступу до системних ресурсів.

Інтеграція з JavaScript реалізована через спеціалізований API, який забезпечує ефективний двосторонній обмін даними між WebAssembly модулями та JavaScript кодом. Модулі WebAssembly дозволяють викликати свої функції з JavaScript, а також використовувати доступні функції JavaScript для розширення своїх можливостей. Простий обмін даними, такими як числа або логічні значення, виконується через стек викликів із мінімальними витратами.

Для передачі складних структур, таких як рядки чи масиви, у WebAssembly використовується Linear Memory — спільна область пам’яті з базовим розміром 64 КБ, яка може динамічно розширюватися за потреби. Взаємодія між функціями здійснюється через Table Objects, що забезпечують динамічні виклики між WebAssembly і JavaScript. Така інтеграція поєднує продуктивність WebAssembly із гнучкістю JavaScript, роблячи його ефективним інструментом для створення високопродуктивних веб-застосунків.

Впровадження описаної архітектури WebAssembly відкриває широкі перспективи для реалізації високопродуктивних веб-застосунків різного призначення. Аналіз сучасних тенденцій використання технології дозволяє виділити ключові сфери її практичного застосування:

1. Графіка та ігрові застосунки: Використовується для створення 3D-графіки, ігор та візуалізації у веб-середовищі завдяки високій швидкості обробки.
2. Обробка мультимедіа: Застосовується для роботи з аудіо, відео та зображеннями, включаючи складні операції, такі як стиснення, декодування та редагування.
3. Наукові обчислення: Використовується для виконання складних математичних розрахунків і моделювання у веб-додатках.
4. Бізнес-додатки зі складною логікою: Застосовується для побудови складних інтерфейсів і забезпечення інтерактивності у корпоративних застосунках.

5. Інструменти розробників: WebAssembly використовується для створення компіляторів, емуляторів та інших інструментів, які працюють безпосередньо у браузері.

У підсумку, технологія WebAssembly являє собою значний прорив у сфері веб-розробки, пропонуючи революційний підхід до створення високопродуктивних і безпечних веб-застосунків. Її впровадження дозволяє ефективно вирішувати складні завдання безпосередньо у браузері, усуваючи обмеження, притаманні традиційним технологіям веб-програмування, таким як JavaScript. WebAssembly забезпечує виконання низькорівневого байт-коду, оптимізованого для різних платформ, що дає змогу досягати рівня продуктивності, порівнянного з рідними додатками, зберігаючи при цьому портативність і платформонезалежність.

2.2 Огляд мов програмування для компіляції у WebAssembly

Написання коду у бінарному форматі WebAssembly є надзвичайно складним завданням, що значно обмежує його практичне застосування у розробці програмного забезпечення. Існує можливість створення коду WebAssembly безпосередньо у текстовому форматі WAT (WebAssembly Text Format), що слугує читабельною альтернативою бінарному формату (див. Рис. 5). Однак цей підхід є недоцільним для розробки масштабних проєктів через низький рівень абстракції та значну складність написання коду. WAT використовується здебільшого для налагодження та аналізу, оскільки він надає текстове уявлення бінарного коду, яке є зрозумілішим для розробників. Незважаючи на зручність у невеликих завданнях, цей формат не забезпечує високорівневих конструкцій та автоматизації.

```
(module
  (type $t0 (func (param i64) (result i64)))
  (func $fac (export "fac") (type $t0) (param $p0 i64) (result i64)
    (if $I0 (result i64)
      (i64.lt_s
        (local.get $p0)
        (i64.const 1))
      (then
        (i64.const 1))
      (else
        (i64.mul
          (local.get $p0)
          (call $fac
            (i64.sub
              (local.get $p0)
              (i64.const 1)))))))))
```

Рисунок 5 — WAT формат коду WASM

Натомість WebAssembly призначений як ціль компіляції для різноманітних мов програмування, таких як C++, C#, Go та інших. Ця можливість забезпечує розробникам широкий вибір інструментів, що дозволяє обирати оптимальну мову залежно від завдань додатку.

Особливості різних мов програмування впливають на продуктивність і поведінку коду в браузері, зокрема на управління пам’яттю, безпеку та підтримку паралельних обчислень. Це робить можливим адаптацію під специфічні вимоги застосунків, такі як обчислювальна складність або потреба у високій продуктивності.

C/C++

Одними з найпоширеніших мов для системного програмування є C та C++, які позиціонуються як низькорівневі мови програмування, що протягом десятиліть залишаються важливими інструментами в розробці системного та прикладного програмного забезпечення. Мова C, спроектована як загального призначення, має акцент на ефективності та можливості точного контролю апаратного забезпечення. У свою чергу, C++ розширює функціональність C, вводячи об’єктно-орієнтоване програмування, що значно спрощує створення складних програмних структур. Завдяки високій продуктивності та синтаксичній подібності ці мови часто обирають для розробки обчислювально

інтенсивних застосунків, таких як ігри, графічні системи, наукові симуляції та інструменти для обробки зображень.

Їхня ефективність і низький рівень абстракції роблять C і C++ ідеальними для компіляції у WebAssembly, де критично важлива швидкість виконання коду. Інструментом, який найчастіше використовується для цього, є компілятор Emscripten. Він дозволяє перекладати код із мінімальними змінами, забезпечуючи сумісність із браузерними API, такими як WebGL, і надає можливість переносити існуючі бібліотеки й програми у веб-середовище. Це особливо корисно для графічних застосунків і високопродуктивних обчислень.

Основними перевагами використання C та C++ для компіляції в WebAssembly є їхня висока продуктивність і доступ до апаратного забезпечення, що забезпечує створення оптимізованих програм із мінімальними затримками. Крім того, велика кількість доступних бібліотек, таких як OpenCV для обробки зображень чи Eigen для числових обчислень, дозволяє легко переносити складні функціональні можливості в браузер. Спільнота розробників із багаторічним досвідом і значним обсягом відкритих матеріалів забезпечує ефективне вирішення потенційних проблем.

Втім, ці мови мають і свої недоліки. Управління пам’яттю в C та C++ вимагає ручного втручання, що може призводити до помилок, таких як витоки пам’яті, і ускладнює розробку. До того ж низький рівень абстракції та складний синтаксис підвищують поріг входу для новачків, які звикли до мов високого рівня. Незважаючи на ці виклики, використання C та C++ у поєднанні з WebAssembly відкриває нові горизонти для продуктивних та ефективних веб-застосунків.

C#

На відміну від C і C++, які орієнтовані на низькорівневе програмування з акцентом на максимальну ефективність і повний контроль над управлінням пам’яттю, C# є мовою високого рівня, створеною для розробки додатків і сервісів із використанням сучасних програмних парадигм. Ця мова,

розроблена Microsoft у межах екосистеми .NET, поєднує простий і зрозумілий синтаксис із потужним інструментарієм та бібліотеками, що забезпечує ефективну реалізацію бізнес-додатків, веб-сервісів, мобільних застосунків і програмного забезпечення для Windows.

Запровадження підтримки WebAssembly у рамках проєкту Blazor значно розширило сферу застосування C# у веб-розробці. Завдяки можливості писати фронтенд і бекенд-код однією мовою, з’явилася можливість знизити складність створення великих проєктів і оптимізувати їхню підтримку. Використання єдиної екосистеми для серверних і клієнтських задач сприяє більш ефективному розробленню інтегрованих рішень.

C# вирізняється низкою переваг, зокрема сучасними можливостями платформи .NET, яка у поєднанні з Blazor забезпечує доступ до широкого спектра бібліотек і сервісів. Зручний синтаксис мови та інтеграція механізмів асинхронного програмування підвищують ефективність розробки веб-застосунків, а підтримка різних платформ дозволяє створювати кросплатформені рішення, сумісні з Windows, Linux і macOS.

Водночас, використання C# у контексті WebAssembly має певні обмеження. Продуктивність мови може бути нижчою порівняно з C або C++ через високий рівень абстракції та механізми автоматичного збирання сміття, які створюють додаткові накладні витрати. Також розмір скомпільованого Wasm-файлу, створеного на основі C#, може бути значним, що впливає на швидкість завантаження застосунків, особливо у разі низької швидкості мережі.

Golang

Go — це сучасна мова програмування, розроблена компанією Google, яка зосереджена на простоті, продуктивності та ефективній роботі з конкурентними задачами. Її основна мета — надати розробникам інструмент для створення високопродуктивних програм із мінімальною складністю коду. Лаконічний та інтуїтивно зрозумілий синтаксис мови полегшує її освоєння, що робить Go доступною навіть для початківців. Ключовою перевагою Go є

вбудована підтримка багатопотоковості, яка дозволяє ефективно працювати з паралельними задачами. Це робить мову ідеальною для розробки систем, що потребують високої продуктивності та масштабованості, таких як веб-сервери, мікросервіси та системи обробки великих обсягів даних. Швидкий компілятор Go забезпечує суттєве скорочення часу компіляції і високу швидкість виконання програмного коду.

Можливість компілювати Golang код у WebAssembly дозволяє виконувати його безпосередньо у браузері, що відкриває нові перспективи для розробки веб-застосунків із високою швидкістю обробки даних. Простота синтаксису та надійна підтримка багатопотоковості роблять цю мову зручним вибором для створення продуктивних клієнтських додатків. Водночас екосистема Go у контексті WebAssembly перебуває на етапі активного розвитку, оскільки підтримка цієї технології є порівняно новою. Через це можуть виникати певні обмеження, зокрема труднощі з інтеграцією інших браузерних API.

Таким чином, Go вирізняється простотою використання, потужними механізмами паралельного виконання та високою продуктивністю, що робить її перспективною для задач, пов’язаних із WebAssembly. Однак, поточний етап розвитку підтримки цієї технології потребує врахування обмежень екосистеми, що поки що перебуває у фазі становлення.

2.3 Особливості інтеграції WebAssembly у веб-застосунках

Інтеграція WebAssembly у веб-додатки сприяє реалізації продуктивного та масштабованого програмного забезпечення, дозволяючи ефективно вирішувати обчислювально-інтенсивні завдання завдяки взаємодії з JavaScript. Завдяки своїм особливостям WebAssembly виступає оптимальним вибором для виконання ресурсомістких операцій, тоді як JavaScript зберігає свою провідну роль у маніпуляції DOM та реалізації бізнес-логіки.

Архітектурні моделі інтеграції WebAssembly є ключовими у виборі оптимального підходу до побудови веб-додатків. Залежно від вимог до продуктивності, функціональності та гнучкості розробки, вони передбачають різний ступінь взаємодії між WebAssembly, JavaScript та іншими компонентами екосистеми веб-технологій [18]. Кожна модель дозволяє адаптувати використання WebAssembly до конкретних потреб застосунку, забезпечуючи баланс між швидкістю виконання, зручністю інтеграції та архітектурною складністю.

Існує кілька архітектурних моделей інтеграції WebAssembly у веб-додатки:

1. Гібридна модель (JavaScript + Wasm): Комбінує переваги обох технологій, делегуючи WebAssembly виконання обчислювально-інтенсивних операцій, тоді як JavaScript зберігає контроль над логікою програми та взаємодією з DOM.
2. Модульна модель: WebAssembly представляється у вигляді окремих модулів, які можна завантажувати динамічно, що спрощує управління складними застосунками.
3. Повністю WebAssembly-додатки: Рідкісний підхід, у якому WebAssembly відповідає за всі аспекти виконання, включно з обробкою UI, наприклад за допомогою таких фреймворків, як Blazor.

Ключовим аспектом інтеграції WebAssembly є ефективна взаємодія компонентів системи. У гібридній моделі механізм комунікації між JavaScript та WebAssembly модулями реалізується через WebAssembly JavaScript API, який забезпечує двосторонній обмін даними та виклики функцій.

Особливої уваги заслуговує питання управління пам'яттю при інтеграції WebAssembly. На відміну від JavaScript, де керування пам'яттю здійснюється автоматично через збирач сміття, WebAssembly модулі потребують явного управління пам'яттю. Це вимагає реалізації коректних механізмів алокації та звільнення ресурсів, особливо при роботі з великими масивами даних та складними структурами.

При використанні модульної моделі інтеграції важливим фактором є оптимізація процесу завантаження модулів. Застосування техніки lazy loading дозволяє завантажувати WebAssembly модулі за потребою, що значно покращує початкову швидкість завантаження веб-додатку. Крім того, модульний підхід спрощує тестування та підтримку окремих компонентів системи.

Суттєвим аспектом інтеграції є забезпечення типової безпеки при взаємодії між JavaScript та WebAssembly. Використання інструментів генерації прив'язок, таких як wasm-bindgen для Rust або Emscripten для C++, дозволяє автоматизувати створення безпечних інтерфейсів взаємодії та мінімізувати можливі помилки при передачі даних між різними середовищами виконання.

У контексті оптимізації продуктивності важливо враховувати особливості компіляції та виконання WebAssembly коду. Сучасні браузери підтримують механізми JIT-компіляції для WebAssembly, що дозволяє досягати близької до нативної швидкості виконання. Проте, ефективність такої оптимізації значною мірою залежить від правильного структурування коду та вибору відповідних паттернів взаємодії між компонентами.

Окремої уваги заслуговує питання налагодження та профілювання інтегрованих WebAssembly додатків. Сучасні інструменти розробки, такі як Chrome DevTools та Firefox Developer Tools, надають спеціалізовані засоби для аналізу продуктивності та відлагодження WebAssembly коду, що значно спрощує процес розробки та оптимізації.

При проектуванні архітектури веб-додатку з використанням WebAssembly важливо також враховувати аспекти масштабування та підтримки. Правильно спроектована система повинна передбачати можливість легкого оновлення як проєктованих модулів, так і JavaScript компонентів, забезпечуючи при цьому зворотну сумісність та мінімізуючи потенційні проблеми при розгортанні нових версій.

2.4 Аналіз сучасних технологій для розробки веб-застосунків

2.4.1 Фреймворк створення серверної частини

У контексті розробки сучасних веб-застосунків особливої уваги заслуговує питання вибору оптимального фреймворку для створення серверної частини. Одним із найперспективніших фреймворків, що відповідає сучасним вимогам розробки серверних застосунків, є NestJS. Даний фреймворк представляє собою прогресивну платформу для створення ефективних, надійних та масштабованих серверних застосунків на Node.js. В основі архітектури NestJS лежать принципи об'єктно-орієнтованого програмування (ООП), функціонального програмування та реактивного програмування.

Архітектурно NestJS побудований з використанням TypeScript, що забезпечує строгу типізацію та підтримку найсучасніших можливостей JavaScript. Фреймворк реалізує паттерн Model-View-Controller (MVC), що дозволяє створювати чітко структуровані та легко тестовані застосунки [19]. Важливою особливістю є використання декораторів, які значно спрощують процес конфігурації та розширення функціональності компонентів системи.

Основними структурними елементами NestJS є:

1. Модулі (Modules) - базові будівельні блоки застосунку, що інкапсулюють пов'язану функціональність;
2. Контролери (Controllers) - відповідають за обробку вхідних запитів та повернення відповідей клієнту;
3. Провайдери (Providers) - сервіси, репозиторії та інші класи, що реалізують бізнес-логіку;
4. Пайпи (Pipes) - забезпечують валідацію та трансформацію вхідних даних;
5. Гварди (Guards) - відповідають за авторизацію та контроль доступу;
6. Інтерсептори (Interceptors) - дозволяють додавати додаткову логіку до обробки запитів та відповідей.

Фреймворк NestJS забезпечує вбудовану підтримку широкого спектра механізмів впровадження залежностей (Dependency Injection), що є ключовим аспектом для розробки слабко зв’язаного, модульного та легко тестованого коду. Архітектура впровадження залежностей у NestJS базується на можливостях TypeScript і застосовує декоратори для визначення та управління залежностями між компонентами. Це підхід не лише спрощує підтримку проєктів, а й сприяє дотриманню принципів SOLID, що є критичними для створення якісного програмного забезпечення.

Важливою перевагою NestJS є наявність широкого набору готових модулів та інтеграцій з популярними базами даних, системами кешування, брокерами повідомлень та іншими сервісами. Це дозволяє швидко розробляти складні системи, використовуючи перевірені та оптимізовані рішення.

Таким чином, NestJS представляє собою потужний та гнучкий фреймворк для розробки серверних застосунків, що поєднує в собі найкращі практики та патерни програмування з сучасними технологічними рішеннями. Його використання дозволяє створювати надійні, масштабовані та безпечні веб-застосунки, що відповідають високим вимогам сучасної розробки програмного забезпечення.

2.4.2 Бібліотека розробки користувацької частини

React є однією з найпоширеніших JavaScript-бібліотек, яка використовується створення інтерфейсів користувача. Його привабливість для програмістів виникає через різні елементи, включаючи декларативний метод створення, модульну структуру, швидке виконання та активну допомогу спільноти [20].

Ця технологія спирається на кількох ключових концепціях, що робить її потужним інструментом для створення веб-додатків:

1. Компонентна архітектура:React дозволяє розробникам розбивати інтерфейс на дрібні, повторно використовувані компоненти. Кожен

компонент інкапсулює свій стан і логіку, що полегшує тестування, відлагодження та масштабування додатка.

2. JSX (JavaScript XML):JSX є розширенням синтаксису JavaScript, яке дозволяє описувати структуру компонентів у форматі, схожому на HTML. Це спрощує читабельність та підтримку коду, адже структура інтерфейсу описується безпосередньо в коді компонентів.
3. Декларативний рендеринг:React оновлює користувацький інтерфейс відповідно до змін у стані чи властивостях компонентів. Це звільняє розробників від необхідності працювати з DOM напряму та зменшує ймовірність помилок.
4. Віртуальний DOM:React використовує концепцію віртуального DOM для оптимізації оновлення інтерфейсу. Замість внесення змін безпосередньо в DOM браузера, React спочатку оновлює віртуальний DOM, що є легковаговою копією реального DOM. Після цього React порівнює стару та нову версії віртуального DOM (процес "дифінгу") і мінімізує кількість змін, що вносяться до реального DOM.

Важливою особливістю React є реалізація однонаправленого потоку даних (one-way data flow), що забезпечує передбачувану поведінку застосунку та спрощує процес відлагодження. Дані передаються від батьківських компонентів до дочірніх через властивості, а зміни стану відбуваються через спеціальні функції-обробники подій.

Однонаправлений потік даних дозволяє чітко відстежувати, як дані проходять через додаток, і забезпечує більшу стабільність, особливо у великих проєктах. React рекомендує керувати станом компонентів через локальний state або використовувати спеціальні бібліотеки для управління станом, такі як Redux, MobX чи Recoil, якщо додаток стає складнішим.

React використовується в багатьох сферах, включаючи:

1. Односторінкові додатки (SPA): Завдяки високій продуктивності та компонентному підходу React ідеально підходить для створення SPA.

2. Мобільні додатки: React Native — це бібліотека, побудована на основі React, яка дозволяє створювати нативні додатки для iOS та Android.
3. Корпоративні веб-додатки: Завдяки масштабованості React використовується для розробки складних інтерфейсів, таких як CRM або ERP-системи.

Завдяки своїй гнучкості, простоті використання та високій продуктивності, дана бібліотека займає провідні позиції серед інструментів розробки сучасних користувацьких інтерфейсів. Її функціональні можливості, активна підтримка спільноти та розвинута екосистема забезпечують ефективну реалізацію як простих, так і висококомплексних веб-застосунків.

2.4.3 Додаткові технології для розробки веб-застосунків

Розробка сучасних додатків передбачає використання різноманітних інструментів і технологій, які спрощують процес створення, тестування та підтримки програмного забезпечення. Додаткові технології дозволяють інтегрувати різні аспекти роботи з даними, інтерфейсом користувача та інфраструктурою. У цьому розділі розглянуто такі популярні технології, як Sequelize, PostgreSQL, Redux Toolkit, Material-UI.

Sequelize — це ORM (Object-Relational Mapping) для Node.js, яка дозволяє розробникам взаємодіяти з реляційними базами даних, такими як PostgreSQL, MySQL, MariaDB, SQLite та Microsoft SQL Server, за допомогою об’єктно-орієнтованого підходу [21]. Цей інструмент абстрагує складність написання SQL-запитів, забезпечуючи при цьому гнучкість і можливість конфігурування. Sequelize надає підтримку транзакцій, асоціацій між моделями, валідації даних та автоматичну генерацію схем бази даних. Завдяки своїй зручності й широкому функціоналу, ця бібліотека значно прискорює розробку, мінімізуючи помилки при роботі з базами даних.

PostgreSQL був обраний як основна реляційна система керування базами даних з відкритим вихідним кодом, яка характеризується високою продуктивністю, масштабованістю. PostgreSQL пропонує розширені

можливості, такі як робота з JSON-даними, транзакції, збережені процедури, реплікація та підтримка розширень [22]. Завдяки своїй надійності, вона широко використовується для розробки сучасних веб-додатків, які потребують обробки великих обсягів даних. PostgreSQL добре інтегрується з сучасними технологіями, включаючи Sequelize, що полегшує її використання в повноцінних стекових рішеннях.

Redux Toolkit (RTK) — потужний інструмент для управління станом у веб-додатках, який спрощує використання Redux шляхом надання набору готових до використання функцій і шаблонів. Redux Toolkit автоматизує багато рутинних завдань, таких як створення редюсерів, дій та середовищ розробки. Завдяки інтеграції з асинхронними операціями через createAsyncThunk та можливості зберігати стан у локальному сховищі або синхронізувати його між вкладками браузера, Redux Toolkit став невід'ємною частиною сучасної розробки масштабованих веб-додатків [23].

Material-UI (MUI) — бібліотека компонентів для React, створена на основі принципів Material Design від Google. Вона пропонує готові адаптивні компоненти та розширені можливості для кастомізації тем, включаючи типографіку, кольорові палітри та інтервали, що забезпечує відповідність дизайну проєкту. Це включає в себе налаштування типографіки, кольорових палітр та інтервалів, що гарантує, що додаток відповідає бажаним принципам дизайну [24].

Використання таких сучасних технологій, як Sequelize, PostgreSQL, Redux Toolkit, Material-UI, полегшує розробку веб-додатків, підвищує продуктивність і пропонує більш гнучкі масштабовані рішення. Кожен інструмент вносить свій внесок у структуру програми, дозволяючи створювати чудові та лідируючі на ринку пропозиції.

2.5 Висновки до розділу 2

1. Технологія WebAssembly (Wasm) забезпечує високу продуктивність веб-додатків завдяки можливості виконання коду ближче до машинного рівня, ефективній роботі з пам’яттю та підтримці мов програмування, що компілюються у байт-код. Wasm є перспективним рішенням для розширення можливостей браузера та створення високопродуктивних серверних застосунків.
2. Найбільш оптимальною мовою програмування для компіляції у WebAssembly є C/C++. Оскільки вона містить необхідний набір функціоналу та демонструє велику продуктивність.
3. Для розробки веб-додатків доцільно використовувати стек технологій, що включає NestJS для серверної частини, React для клієнтського інтерфейсу, а також інтеграцію з WebAssembly для вирішення задач, пов’язаних з підвищенням продуктивності.
4. У якості додаткових технологій доречно застосувати Sequelize, PostgreSQL, Material UI та Redux Toolkit.

3 РОЗРОБКА ЗАСТОСУНКУ З ВИКОРИСТАННЯМ ТЕХНОЛОГІЇ WEB-ASSEMBLY

3.1 Опис предметної області

Предметна область, яка досліджується в рамках цієї роботи, охоплює веб-застосунки з акцентом на підвищення їхньої продуктивності та безпеки за допомогою технології WebAssembly (Wasm). Ресурсоємні веб-застосунки стикаються з викликами традиційних підходів до обробки даних у браузері, що обумовлює потребу у впровадженні інноваційних рішень для досягнення необхідної швидкодії та ефективності.

Однією з головних проблем сучасних веб-додатків є недостатня продуктивність під час виконання ресурсоємних операцій. JavaScript, як основний інструмент фронтенд-розробки, має обмеження через інтерпретовану природу, що призводить до зниження швидкодії у випадках роботи з такими задачами, як відеообробка, фізичне моделювання чи аналіз симуляцій. Використання технології, що забезпечує компільоване виконання коду, суттєво знижує витрати на обчислення та підвищує ефективність.

Другий важливий аспект стосується безпеки. Використання численних бібліотек і залежностей у веб-застосунках створює значні ризики вразливостей, включно з ін'єкційними атаками та виконанням шкідливого коду. Завдяки ізольованому середовищу виконання WebAssembly мінімізує ці ризики, забезпечуючи додатковий рівень захисту для задач, пов'язаних із чутливими даними.

Для дослідження цих викликів у рамках роботи обрано веб-додаток відеоредактор. Відеоредактор є показовим прикладом складної системи, яка одночасно вимагає високої обчислювальної продуктивності для обробки великих обсягів даних, таких як відеофайли, та надійного захисту інформації. Висока обчислювальна складність відеообробки створює ідеальне середовище для тестування ефективності WebAssembly, який дозволяє виконувати

ресурсоємні операції у компільованому коді, значно зменшуючи накладні витрати на обчислення.

Застосування цієї технології у контексті створення відеоредактора дозволяє не лише покращити продуктивність обробки даних, але й забезпечити ізоляцію середовища виконання коду, що мінімізує ризики безпеки. Розробка відеоредактора у двох версіях — з використанням WebAssembly та без нього — дозволить провести детальний аналіз впливу цієї технології на ключові аспекти функціонування веб-застосунків.

Таким чином, розробка відеоредактора у контексті даного дослідження є доцільною, оскільки дозволяє отримати практичні результати щодо впливу WebAssembly на продуктивність та безпеку веб-застосунків. Отримані результати дослідження дозволять оцінити ефективність цієї технології у веб-застосунках, які мають підвищені вимоги до продуктивності та безпеки, а також сформулювати рекомендації щодо її оптимального застосування в різних сферах програмних рішень.

3.2 Архітектура застосунку

Архітектура веб-застосунку відеоредактора побудована за клієнт-серверною моделлю, яка залишається сталою незалежно від підходу до обробки відеофайлів. Метою системи є забезпечення ефективної взаємодії між клієнтською та серверною частинами, гнучкості в обробці даних та масштабованості. Для дослідження впливу WebAssembly (Wasm) на продуктивність і безпеку було реалізовано дві версії додатку: серверна, де обробка відео виконується нативною бібліотекою FFmpeg, і клієнтська, яка переносить обчислення у браузер із використанням Wasm-версії FFmpeg.

У контексті безпеки реалізовано шифрування даних за алгоритмом AES у режимі CBC. Спершу шифрування виконувалося на клієнті за допомогою нативної реалізації, але для оцінки ефективності Wasm було інтегровано

WebAssembly-реалізацію, що підвищило продуктивність криптографічних операцій.

Серверна частина концентрує основну бізнес-логіку, зокрема авторизацію, управління проєктами та збереження відеофайлів. Така централізація обчислювальних процесів сприяє забезпеченню стабільності, безпеки та високої ефективності системи, водночас мінімізуючи залежність від апаратних ресурсів клієнтських пристроїв.

На діаграмі зображеній на рисунку 6 можна побачити загальну архітектуру додатку.

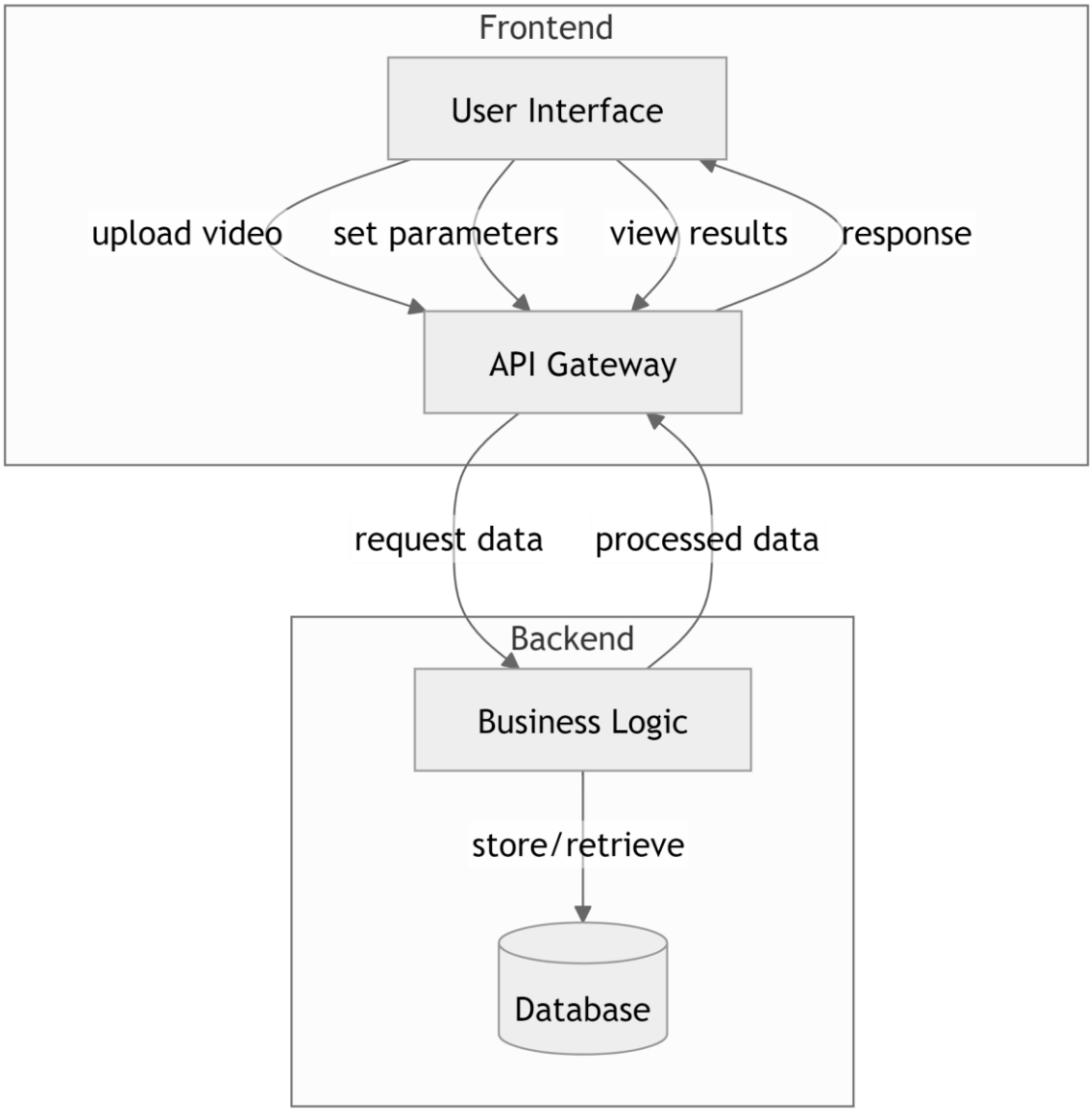


Рисунок 6— Діаграма загальної архітектури застосунку

3.3 Програмна реалізація застосунку

3.3.1 Реалізація бази даних

Ефективна організація даних є ключовою для роботи будь-якого сучасного веб-додатку, особливо коли йдеться про ресурсоємні системи, такі як відеоредактори. База даних відіграє важливу роль, оскільки вона є основним місцем зберігання інформації про користувачів, проєкти, відеофайли, а також права доступу та дії в системі, забезпечуючи продуктивність, безпеку та масштабованість.

Для реалізації системи було обрано реляційну СУБД PostgreSQL завдяки її високій продуктивності, підтримці складних запитів і здатності відповідати вимогам масштабу. Інтеграція з серверною частиною здійснюється за допомогою ORM-фреймворка Sequelize, який спрощує управління даними, знижує ризики SQL-ін'єкцій і забезпечує абстракцію над SQL-запитами. Організація даних ґрунтується на структурі, що відповідає потребам відеоредактора, включаючи підтримку користувачів, збереженням метаданих відеофайлів. У складі бази даних присутні 4 таблиці, кожна з яких відповідає певній сутності.

Сутність «Users» зберігає дані для автентифікації та ідентифікації користувачів, а також інформацію про час створення та оновлення записів. Атрибути цієї сутності наведені в таблиці 1.

Таблиця 1 — Атрибути сутності «Users»

Поле	**Тип**	**Опис**
id	varchar(255)	Первинний ключ
nickname	varchar(255)	Ім'я користувача
email	varchar(255)	Адреса електронної пошти
password	varchar(255)	Пароль користувача
createdAt	date	Дата створення профілю

updatedAt	date	Дата оновлення профілю

Сутність «Tokens» відповідає за управління сесіями користувачів, зберігаючи автентифікаційні токени, терміни їх дії та посилання на користувачів. Атрибути цієї сутності наведені в таблиці 2.

Таблиця 2 — Атрибути сутності «Tokens»

Поле	**Тип**	**Опис**
token	varchar(255)	Данні токену
userId	varchar(255)	Користувач якому належить цей токен
expiresAt	date	Час завершення дії токена

Сутність «Projects» містить інформацію про проєкти створені користувачем, які можуть містити відеофайли. Атрибути цієї сутності наведені в таблиці 3.

Таблиця 3 — Атрибути сутності «Projects»

Поле	**Тип**	**Опис**
id	varchar(255)	Первинний ключ
title	varchar(255)	Назва проєкту
description	varchar(255)	Опис проєкту
userId	varchar(255)	Користувач якому належить цей проєкт
createdAt	date	Дата створення проєкту
updatedAt	date	Дата оновлення проєкту

Сутність «Videos» призначена для зберігання інформації про відеофайли, завантажені користувачами до певного проєкту. Вона містить метаінформацію про відео, включаючи шлях до файлу та мініатюри, що забезпечує зручне управління файлами. Всі ці параметри відображені у таблиці 4.

Таблиця 4 — Атрибути сутності “Videos”

Поле	**Тип**	**Опис**
id	varchar(255)	Первинний ключ
title	varchar(255)	Назва відео файлу
duration	int	Тривалість відео у секундах
fileSize	int	Розмір файлу у кілобайтах
mimetype	varchar(255)	Mimetype відео файлу
filePath	varchar(255)	Шлях до файлу на сервері
thumbnailPath	varchar(255)	Шлях до мініатюри файлу на сервері
projectId	varchar(255)	Проєкт якому належить цей файл
createdAt	date	Дата створення проєкту
updatedAt	date	Дата оновлення проєкту

Для кращого розуміння архітектури даних та зв'язків між основними сутностями бази даних, представлено ER діаграму зображену на рисунку 7, яка ілюструє взаємозв'язки між сутностями та їх ключові атрибути.

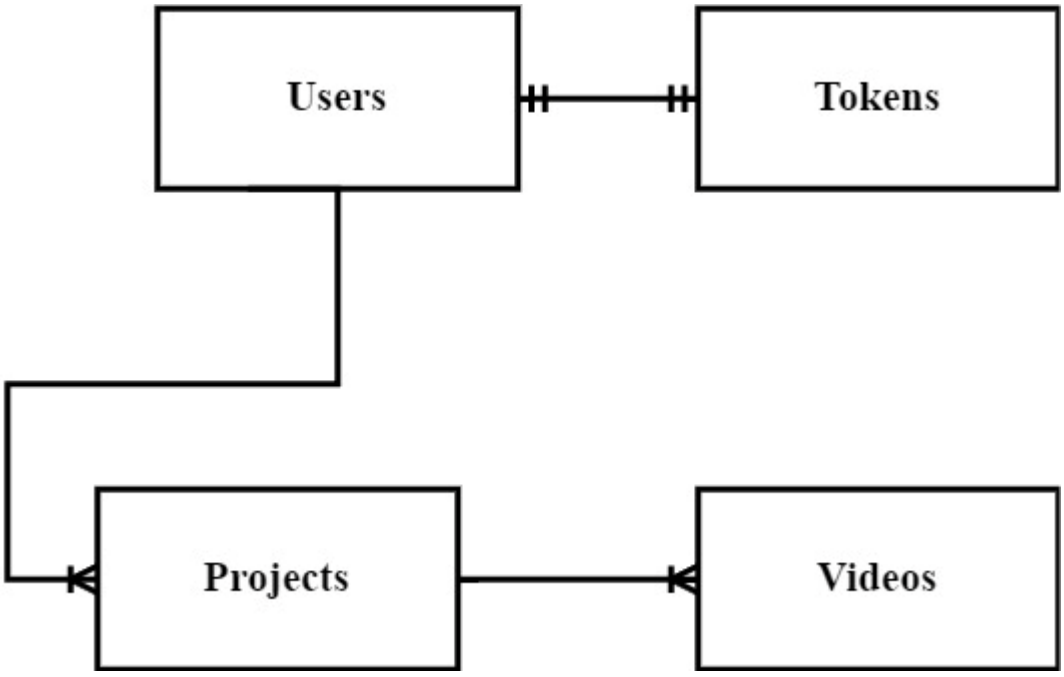


Рисунок 7— ER діаграма БД проекту відеоредактор

3.3.2 Реалізація серверної частини

Серверна частина забезпечує стабільне функціонування додатку, включаючи обробку даних, безпеку, інтеграцію з клієнтським інтерфейсом та управління ресурсами. Для її реалізації використовується фреймворк NestJS, що сприяє створенню модульної та масштабованої архітектури.

NestJS організовує серверну частину у вигляді модулів — незалежних структурних компонентів. За допомогою декоратора @Module групуються сервіси, контролери та провайдери в логічні одиниці. Це забезпечує модульність, легкість підтримки та масштабування коду. Кожен модуль виконує окрему задачу, що спрощує тестування і розширення системи. Сама серверна частина включає в себе такі модулі:

1. AuthModule – відповідає за функціонал аутентифікації та авторизації користувачів.
2. UserModule – управляє операціями, пов'язаними з даними користувачів.
3. ProjectModule – забезпечує роботу з проектами користувачів.
4. FileUploadModule – реалізує функції завантаження та збереження файлів.
5. VideoModule – обробляє відеофайли та їх метадані.
6. FFmpegModule – інтегрує бібліотеку FFmpeg для виконання завдань обробки відео.

Кожен модуль має строго визначену функціональність та взаємодіє з іншими модулями через сервіси, що інкапсулюють бізнес-логіку. Такий підхід сприяє дотриманню принципу розподілу відповідальності, що, своєю чергою, полегшує подальший розвиток і підтримку системи.

Функціональні можливості модулів реалізуються за допомогою сервісів, контролерів та репозиторіїв, які взаємодіють із базою даних через ORM Sequelize. Це забезпечує зручне управління сутностями, такими як

користувачі, проєкти та відеофайли, відповідно до специфікацій і вимог системи.

Для спрощення створення репозиторіїв що будуть працювати з сутностями з бази даних у серверній частині реалізовано узагальнений клас BaseAbstractRepository (див. лістинг 1). Він визначає стандартні CRUD-операції, що можуть бути використані для будь-якої сутності.

Лістинг 1 — *Узагальнений класс BaseAbstractRepository*

```
export abstract class BaseAbstractRepository<T extends Model<T>>
  implements IBaseRepository<T>
{
  constructor(@InjectModel(Model) private readonly entityModel: ModelCtor<T>) {}
  create(dto: CreationAttributes<T>): Promise<T> {
    return this.entityModel.create(dto);
  }
  findOneById(id: string): Promise<T | null> {
    return this.entityModel.findByPk(id);
  }
  findOne(options?: FindOptions<Attributes<T>>) {
    return this.entityModel.findOne(options);
  }
  findMany(options?: FindOptions<Attributes<T>>): Promise<T[]> {
    return this.entityModel.findAll(options);
  }
  update(entity: T, dto: Attributes<T>): Promise<T> {
    return entity.update(dto);
  }
  delete(id: WhereAttributeHashValue<Attributes<T>['id']>): Promise<number> {
    return this.entityModel.destroy({ where: { id } });
  }
```

```
}
```

У цьому класі реалізовано базові CRUD-операції для роботи з базою даних через ORM Sequelize. Він використовує узагальнення, що дозволяє адаптувати його для роботи з будь-якою моделлю, яка відповідає інтерфейсу Model. Конструктор класу передбачає ін’єкцію конкретної моделі через декоратор @InjectModel(Model), що забезпечує зручність та легкість інтеграції.

Методи класу включають створення нових записів у базі даних за допомогою методу create, пошук записів за первинним ключем (findOneById), пошук одного запису за заданими умовами (findOne), отримання множини записів (findMany), оновлення даних існуючого запису (update) та видалення записів за ідентифікатором (delete). Усі методи базуються на стандартних можливостях Sequelize, що забезпечує ефективність і зручність роботи з даними.

Для інтеграції в модульну архітектуру NestJS кожен модуль використовує відповідну реалізацію репозиторію, що базується на BaseAbstractRepository. Наприклад, для роботи з користувачами створюється UserRepository, який наслідує базовий клас BaseAbstractRepository (див. лістинг 2).

Лістинг 2 — *Класс UserRepository*

```
@Injectable()
export class UserRepository extends BaseAbstractRepository<User> {
  constructor(@InjectModel(User) userModel: typeof User) {
    super(userModel);
  }
  findOneByEmail(email: string) {
    return this.findOne({ where: { email } });
  }
}
```

Як можна побачити, цей клас реалізує спеціалізований репозиторій для роботи з користувачами, успадковуючи базовий функціонал від BaseAbstractRepository. Завдяки цьому стандартні CRUD-операції, такі як створення, пошук, оновлення та видалення записів, вже доступні без необхідності їхньої повторної реалізації. Додатково, клас додає метод findOneByEmail, який виконує пошук користувача за електронною адресою. Цей метод використовує можливості базового методу findOne, передаючи йому умови пошуку.

На наступному етапі доцільно перейти до розгляду класу UserService, який взаємодіє з цим репозиторієм та визначає бізнес-логіку для роботи з користувачами (див. лістинг 3).

Лістинг 3 — *Класс UserService*

```
@Injectable()
export class UserService {
  constructor(private userRepository: UserRepository) {}

  getAll(limit: number, offset: number): Promise<User[]> {
    return this.userRepository.findMany({limit,offset});
  }

  getOneByEmail(email:string): Promise<User>{
    const user = this.userRepository.findOneByEmail(email);
    if(!user){
      throw new NotFoundException('User with this email is
not exist');
    }
    return user;
  }

  getOneById(id:string):Promise<User>{
    const user = this.userRepository.findOneById(id);
```

```
        if(!user){
          throw new NotFoundException('User with this id is not
exist');
        }
        return user;
      }

      async create(dto: CreateUserDto): Promise<User> {
        const              user              =              await
this.userRepository.findOneByEmail(dto.email);
        if (user) {
          throw  new  NotFoundException('User  with  this  email
already exist');
        }
        return this.userRepository.create({ ...dto });
      }

      async update(id:string, dto:UpdateUserDto): Promise<User>
{
        const user = await this.userRepository.findOneById(id);
        if(!user){
          throw new NotFoundException('User with this id is not
exist');
        }
        return this.userRepository.update(user,{...dto});
      }

      delete(id: string): Promise<Number> {
        return this.userRepository.delete(id);
      }
    }
```

Цей код представляє клас UserService, який реалізує бізнес-логіку для управління користувачами. Він використовує репозиторій UserRepository для виконання CRUD-операцій. Методи getAll, getOneByEmail та getOneById

забезпечують отримання списку користувачів, пошук за електронною адресою або ідентифікатором, із обробкою винятків для випадків, коли користувач не знайдений.

Методи create та update відповідають за створення нового користувача та оновлення даних існуючого, перевіряючи попередньо наявність користувача. Метод delete забезпечує видалення запису за ідентифікатором. Такий підхід дозволяє додати рівень перевірок та гарантує коректну обробку даних.

Для роботи з відео файлами у системі передбачено використання спеціалізованого контролера VideoController, який відповідає за обробку запитів, пов’язаних із відеофайлами. Він забезпечує взаємодію між користувачем та сервісами, які реалізують бізнес-логіку, зокрема створення, обробку, оновлення та видалення відеофайлів. Контролер виступає ключовою ланкою в обробці HTTP-запитів, спрямованих на управління відеофайлами, і гарантує коректність передачі даних до відповідних сервісів що можна побачити на лістингу 4.

Лістинг 4 — *Класс VideoController*

```
@Controller('video')
export class VideoController {
    constructor(
        private readonly videoService: VideoService,
    ) { }

    @Post('upload/:projectId')
    @UseInterceptors(FileInterceptor('file'))
    @UseGuards(JwtAuthGuard)
    async uploadVideo(@UploadedFile() file: Express.Multer.File, @Req() req, @Param('projectId') projectId: string) {
        return this.videoService.handleVideoUpload(file, req.user.id, projectId, STATIC_FOLDER_PATH);
```

```
    }

    @Get('project/:projectId')
    async getVideosByProject(@Param('projectId') projectId:
string) {
        return
this.videoService.findVideosByProject(projectId);
    }

    @Post('process/:id')
    @UseGuards(JwtAuthGuard)
    @HttpCode(HttpStatus.OK)
    async processVideo(
        @Req() req,
        @Param('id') id: string,
        @Body() dto: ProcessVideoDto,
        @Res() res: Response
    ) {
        console.log(dto);
        return                                          await
this.videoService.handleVideoProcessing(id,   req.user.id,   dto,
STATIC_FOLDER_PATH, req, res);
    }

    @Get(':id')
    async getVideoById(@Param('id') id: string, @Req() req:
Request, @Res() res: Response) {
        console.log("Aboba");
        await  this.videoService.handleGetVideoById(id,  req,
res);
    }

    @Delete(':id')
    async deleteVideo(@Param('id') id: string) {
        await this.videoService.deleteVideoById(id);
```

```
            return { message: 'Video successfully deleted' };
        }
    }
```

Цей клас забезпечує управління відеофайлами, виконуючи функції маршрутизації, передачі даних до сервісів, обробки запитів, пов’язаних із завантаженням, обробкою, отриманням і видаленням відео. Він виступає посередником між клієнтськими запитами та бізнес-логікою, використовуючи механізми аутентифікації для захисту операцій і забезпечення коректності обробки файлів та параметрів запиту.

Маршрут /video/upload/:projectId призначений для завантаження відеофайлів у межах конкретного проєкту. Він обробляє файли за допомогою інтерсептора та передає їх у сервіс разом із даними про користувача й проєкт.

Маршрут /video/project/:projectId дозволяє отримати список усіх відеофайлів, прив’язаних до певного проєкту. Ідентифікатор проєкту використовується для пошуку відповідних даних у базі.

Маршрут /video/process/:id виконує обробку відеофайлу за його ідентифікатором. Він приймає додаткові параметри з тіла запиту, перевіряє права доступу користувача та викликає відповідний метод сервісу.

Маршрут /video/:id забезпечує доступ до конкретного відеофайлу за його унікальним ідентифікатором. Сервіс отримує файл і передає його у відповідь клієнту.

Маршрут /video/:id дозволяє видалити відеофайл із системи за його ідентифікатором. Сервіс видаляє запис та сам відеофайл, а клієнт отримує підтвердження про успішну операцію.

3.3.3 Реалізація користувацького інтерфейсу

Користувацький інтерфейс відеоредактора є ключовим елементом системи, що забезпечує зручну та ефективну взаємодію користувачів із функціоналом додатку. Використання TypeScript забезпечує статичну

типізацію, що сприяє підвищенню надійності та зручності підтримки коду. Для створення сучасного дизайну застосовано бібліотеку Material-UI (MUI), яка надає адаптивні та зручні у використанні компоненти.

Розробка інтерфейсу базується на модульному підході: код організовано у вигляді незалежних компонентів, кожен з яких виконує конкретну функцію. Основні елементи включають навігацію, авторизацію, завантаження відео, інструменти редагування та перегляд інформації про проекти.

Для ефективної роботи з API використовується RTK Query з Redux Toolkit. Вона спрощує запити до серверу, забезпечує кешування та автоматичне управління станом. Наприклад, функціонал завантаження та обробки відео реалізовано через RTK Query для оптимізації роботи з REST API та динамічного оновлення інтерфейсу.

Ключовим модулем є завантаження й на лістингу 5 зображено реалізацію інтеграції API для роботи з відеофайлами за допомогою RTK Query, яка забезпечує функціональність завантаження, отримання, обробки та видалення відео.

Лістинг 5 — *API-запити для роботи з відео через RTK Query*

```
export const videoAPI = enchancedApi.injectEndpoints({
  endpoints: (builder) => ({
    uploadVideo: builder.mutation<any, { formData: FormData; projectId: string }>({
      query: ({ formData, projectId }) => ({
        headers: {
          Authorization: `Bearer ${localStorage.getItem('token')}`
        },
        url: `/video/upload/${projectId}`,
        method: "POST",
        body: formData,
      }),
      invalidatesTags: ["Video"],
```

```
        }),
        fetchVideoById: builder.query({
          query: (id) => ({
            url: `video/${id}`,
            method: 'GET',
            headers: {
              Range: 'bytes=0-',
            },
            responseHandler: (response) => {
              if (response instanceof Response) {
                return response.blob();
              }
              throw new Error('Response is not an instance of
Response');
            },
          }),
          transformResponse: async (encryptedBlob: Blob, meta) =>
{
            const ivBase64 = meta?.response?.headers.get('X-
Encryption-IV');
            if (!ivBase64) {
              throw new Error('Missing IV in response headers');
            }
            return await decryptBlob(encryptedBlob, ivBase64);
          },
        }),
        fetchVideosByProject: builder.query<IVideo[], string>({
          query: (projectId) => ({
            headers: {
              Authorization:                                `Bearer
${localStorage.getItem('token')}`,
            },
            url: `/video/project/${projectId}`,
            method: "GET",
          }),
```

```
        providesTags: ["Video"],
      }),
      deleteVideo: builder.mutation<void, string>({
        query: (videoId) => ({
          headers: {
            Authorization:                                  `Bearer
${localStorage.getItem('token')}`,
          },
          url: `/video/${videoId}`,
          method: "DELETE",
        }),
        invalidatesTags: ["Video"],
      }),
      processVideo:  builder.mutation<Blob,  {  id:  string;
operationType: VideoOperationType; filterType?: VideoFilterType;
startTime?: string; duration?: string }>({
        query:  ({  id,  operationType,  filterType,  startTime,
duration }) => ({
          headers: {
            Authorization:                                  `Bearer
${localStorage.getItem('token')}`,
          },
          url: `/video/process/${id}`,
          method: "POST",
          body:   {   operationType,   filterType,   startTime,
duration },
          responseHandler: (response) => response.blob(),
        }),
      }),
    })
  });
```

У цьому коді реалізовано інтеграцію API для управління відеофайлами, що базується на використанні RTK Query. Взаємодія з сервером охоплює такі операції як: завантаження, отримання, обробки та видалення відеофайлів.

Завантаження реалізовано через POST-запити із передачею авторизаційних токенів, що забезпечує безпеку операцій. Отримання даних підтримує як завантаження окремих відеофайлів у форматі Blob, так і отримання списків, прив’язаних до проєктів. Видалення файлів виконується через DELETE-запити з автоматичним оновленням стану кешу. Функціонал обробки передбачає передачу параметрів у форматі JSON і повернення оброблених даних у форматі Blob.

Наступним важливим етапом реалізації є створення сторінки авторизації, яка забезпечить доступ користувачів до функціональності відеоредактора (див. лістинг 6). Логіка автентифікації та авторизації, зокрема обробка введених даних і взаємодія з бекендом, була реалізована з урахуванням сучасних підходів до безпеки та зручності використання.

Лістинг 6 — *Сторінка авторизації користувача*

```
const Login = () => {
    const dispatch = useAppDispatch();
    const [email, setEmail] = useState('');
    const [password, setPassword] = useState('');
    const [login,{data}] = authAPI.useLoginMutation();

    useEffect(()=>{
        if(data?.user){
            dispatch(setUser({...data.user}));
        }
        if(data?.token){
            localStorage.setItem('token',data.token);
        }
    },[data])

    const loginFunc = () => {
        login({ email, password });
        setPassword('');
        setEmail('');
```

```
        }

        return (
            <Container maxWidth="lg" className={styles.login}>
                <Paper elevation={10} sx={{p:10}}>
                    <Stack spacing={2}>
                        <Typography component="h1"
variant="h5">Sign in</Typography>
                        <TextField label="Email"
placeholder="Enter email" value={email} onChange={(e) => {
                            setEmail(e.target.value)
                        }} />
                        <TextField label="Password"
placeholder="Enter password" type="password" value={password}
onChange={(e) => {
                            setPassword(e.target.value)
                        }} />
                        <Button variant="contained"
onClick={loginFunc}>Sign in</Button>
                        <Typography>Don't have an account?
                            <Link href="/register"
underline="none"> Sign up</Link>
                        </Typography>
                    </Stack>
                </Paper>
            </Container>
        )
    }
    export default Login;
```

Ця сторінка є компонентом авторизації користувачів у веб-додатку. Вона забезпечує введення облікових даних (електронної пошти та пароля) для здійснення автентифікації через API. Компонент використовує хуки useState для зберігання введених даних та useEffect для обробки відповіді сервера,

включаючи збереження токена в локальному сховищі та оновлення стану користувача в додатку. Інтерфейс сторінки містить поля для введення електронної пошти та пароля, кнопку для входу в систему, а також посилання для переходу на сторінку реєстрації.

3.4 Функціонал застосунку

Розробка сучасного веб-додатку потребує не лише глибокого розуміння технологій, але й здатності вирішувати реальні проблеми користувачів. В основі створення відеоредактора лежить прагнення забезпечити інструменти, що спрощують обробку відеоконтенту, роблячи цей процес швидким, ефективним та доступним для широкої аудиторії.

Основний функціонал відеоредактора включає авторизацію користувачів, завантаження, перегляд і базову обробку відео, а також створення, редагування та видалення проєктів із можливістю збереження в них відеофайлів. Система авторизації забезпечує безпеку даних через токен-автентифікацію. Користувачі можуть реєструватися, входити в систему, керувати своїми проєктами та профілем. На рисунку 8 показано сторінку авторизації.

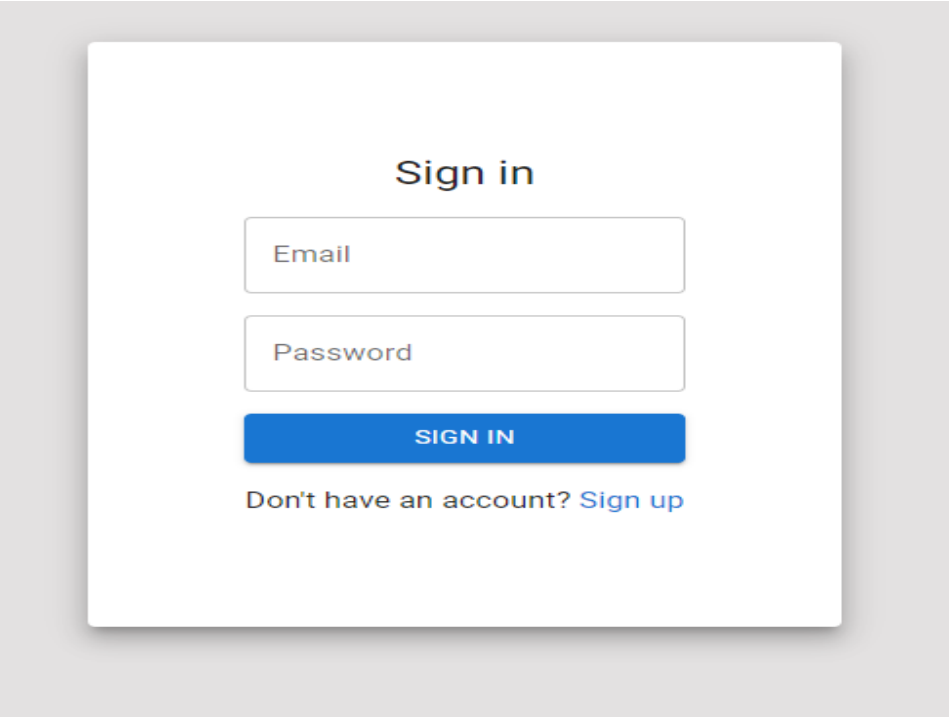


Рисунок 8 — Сторінка авторизації користувача

Сторінка авторизації веб-додатку призначена для ідентифікації користувачів та забезпечення доступу до функціоналу системи. Інтерфейс містить два текстові поля для введення електронної пошти та пароля, які є обов'язковими для заповнення. Нижче розташована кнопка підтвердження дій користувача «Sign in», що ініціює процес автентифікації. Додатково передбачено гіперпосилання «Sign up» для перенаправлення користувачів, які ще не мають облікового запису, на сторінку реєстрації. Мінімалістичний дизайн сторінки сприяє інтуїтивній взаємодії, що підвищує її зручність та функціональність.

Після успішного входу до свого облікового запису. Користувач потрапить на головну сторінку сторінку додатку (див. Рис. 9).

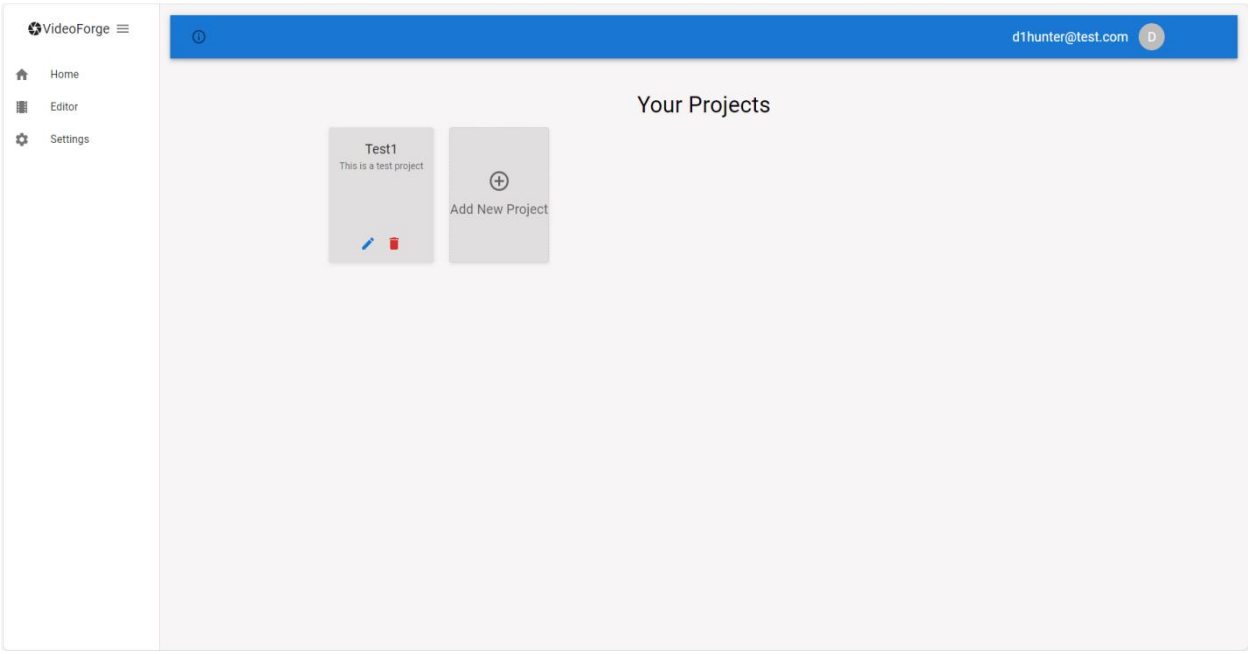


Рисунок 9 — Головна сторінка веб-застосунку

Сторінка головного меню веб-додатку забезпечує управління проектами користувача. Вона містить панель навігації з розділами «Home», «Editor» і «Settings», а також інформацію про поточного користувача. Основна частина відображає проекти у вигляді карток із можливістю редагування, видалення або створення нових проектів через кнопку «Add New Project». Структурований дизайн сторінки сприяє швидкому доступу до основного функціоналу.

Основний із ключових елементі це сторінка редактора, що надає інструменти для базової обробки відео. Перейти на цю сторінку можна двома способами: обравши пункт "Editor" у навігаційній панелі або відкривши конкретний проект на головній сторінці, завантаживши або обравши відео для редагування та натиснувши кнопку "Edit". Інтерфейс редактора (див. Рис. 10) забезпечує інтуїтивно зрозумілий доступ до функцій редагування, таких як обрізання, зміна швидкості відтворення та додавання базових ефектів, що підвищує зручність використання для кінцевих користувачів.

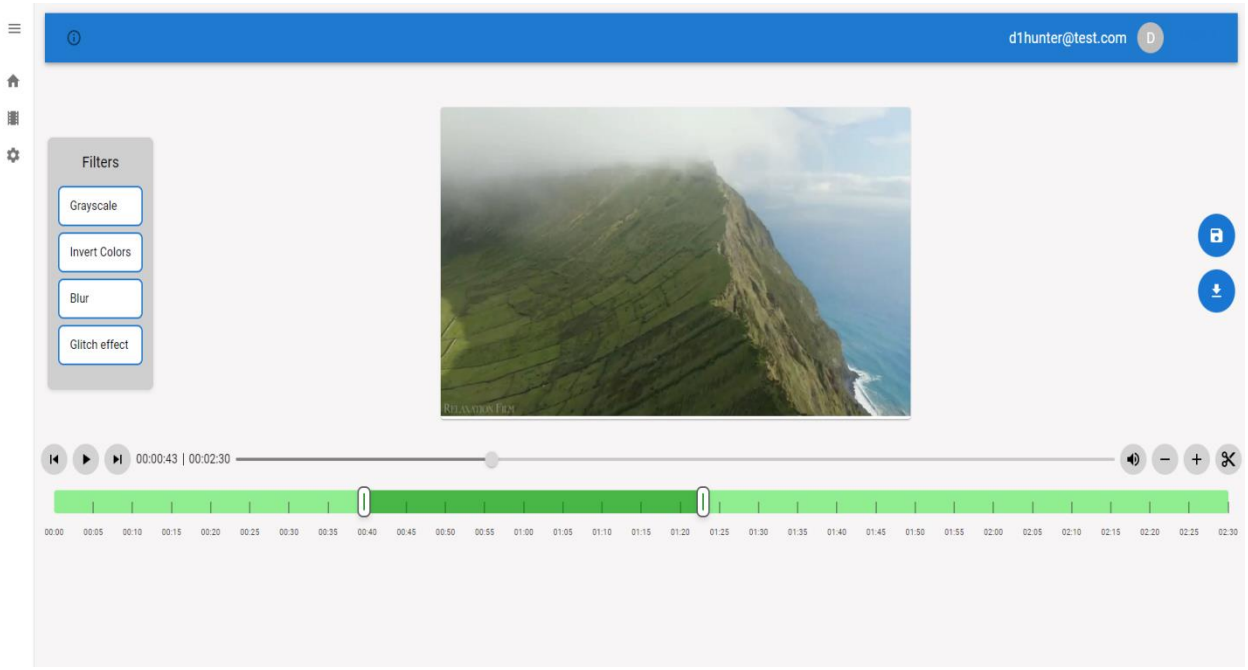


Рисунок 10 — Сторінка редактору

Сторінка відеоредактора забезпечує інтерактивний інтерфейс для роботи з відеоконтентом. Вона містить область попереднього перегляду відео, панель інструментів для базової обробки, таких як обрізання відео, зміна швидкості відтворення та застосування базових ефектів. Завантажене або обране для редагування відео відображається в режимі реального часу, що дозволяє користувачам оперативно оцінювати зміни. Дизайн сторінки розроблено для забезпечення максимальної зручності роботи, динамічне оновлення змін та зворотний зв’язок через інтуїтивно зрозумілий інтерфейс.

3.5 Реалізація Wasm-модулю

3.5.1 Налаштування компіляційного середовища Emscripten

Першим етапом для створення WebAssembly (WASM) модуля є налаштування компіляційного середовища, яке забезпечує перетворення коду низькорівневих мов програмування у формат, придатний для виконання у веб-застосунках. Одним із ключових інструментів для цього є Emscripten SDK, який забезпечує повний набір засобів для роботи з мовами C/C++.

Emscripten SDK базується на компіляторі LLVM, що дозволяє трансформувати код C/C++ у високооптимізований байткод WebAssembly, забезпечуючи міжплатформну сумісність [25]. Окрім базового перетворення, він адаптує семантику програмних конструкцій до нового середовища. Головною особливістю Emscripten є автоматичне генерування допоміжного JavaScript-коду, що спрощує інтеграцію модулів у веб-застосунки, зберігаючи високу продуктивність, наближену до нативних додатків.

Передумовою успішної інсталяції виступає попередня підготовка операційного середовища, що включає інтеграцію системних залежностей та базових інструментів розробки. Необхідним є попереднє встановлення актуальних версій компіляторів, інтерпретаторів та систем керування пакунками, зокрема cmake, nodejs та python. Особливої уваги потребує узгодження версійності компонентів для уникнення потенційних конфліктів сумісності.

Процедура інсталяції Emscripten SDK передбачає послідовне клонування офіційного репозиторію з платформи GitHub. Для цього використовується наступний командний скрипт зазначений у лістингу 7.

Лістинг 7 — *Скрипт встановлення та активації emscripten*

```
git clone https://github.com/emscripten-core/emsdk.git
cd emsdk
./emsdk install latest
./emsdk activate latest
source ./emsdk_env.sh
```

Після успішної активації Emscripten SDK середовище готове до використання. Для перевірки його працездатності рекомендується виконати команду emcc -v, яка виводить інформацію про версію компілятора та параметри конфігурації. Цей крок дозволяє переконатися у правильності встановлення та готовності системи до роботи.

3.5.2 Розробка функціонального модуля на основі C++

Для підвищення безпеки у веб-додатку потрібно було розробити модуль шифрування та дешифрування відеофайлів. У якості реалізації було обрано мову програмування C++ із використанням бібліотеки Tiny AES. Основна мета полягала у створенні ефективного інструменту, здатного забезпечити надійний захист даних та інтеграцію з браузерним середовищем за допомогою WebAssembly (WASM).

Розроблений модуль базується на реалізації алгоритму шифрування AES у режимі CBC, що забезпечує простоту інтеграції та високу продуктивність. Бібліотека Tiny AES була обрана завдяки її компактності та відповідності стандартам AES. Код розробленого модуля можна побачити на лістингу 8.

Лістинг 8 — *Код модулю шифрування та дешифрування*

```
#ifdef __cplusplus
extern "C"
{
#endif

#include "aes.h"

#ifdef __cplusplus
}
#endif

#include <cstring>
#include <emscripten.h>

#include "aes.h"

extern "C"
{
```

```
        void encryptAES(uint8_t *data, size_t *dataLength, const
uint8_t *key, const uint8_t *iv)
        {
            struct AES_ctx ctx;
            AES_init_ctx_iv(&ctx, key, iv);
            AES_CBC_encrypt_buffer(&ctx, data, *dataLength);
        }

        void decryptAES(uint8_t *data, size_t *dataLength, const
uint8_t *key, const uint8_t *iv)
        {
            struct AES_ctx ctx;
            AES_init_ctx_iv(&ctx, key, iv);
            AES_CBC_decrypt_buffer(&ctx, data, *dataLength);
        }

        void* malloc(size_t size) {
            return malloc(size);
        }

        void free(void* ptr) {
            free(ptr);
        }
    }
```

У цьому коді реалізовано функції encryptAES та decryptAES, які обробляють дані блоками по 16 байтів. Шифрування та дешифрування виконуються на основі ключа шифрування (key) і ініціалізаційного вектора (iv). Застосування структури AES_ctx забезпечує управління контекстом шифрування, тоді як функції AES_CBC_encrypt_buffer та AES_CBC_decrypt_buffer виконують безпосередню обробку даних.

Ключовим аспектом модуля є використання директиви extern "C", яка забезпечує збереження імен функцій у форматі, сумісному з WebAssembly, і і запобігає зміні імен компілятором. Це дозволяє легко викликати функції

шифрування та дешифрування з JavaScript або інших мов, які взаємодіють із WASM-модулями.

Додатково реалізовано функції malloc та free, які дозволяють динамічно виділяти та звільняти пам’ять у межах WASM-модуля, що спрощує роботу з даними змінної довжини.

Таким чином, розроблений модуль забезпечує ефективне виконання шифрування та дешифрування даних у середовищі WebAssembly, демонструючи високу продуктивність і сумісність із сучасними стандартами безпеки.

3.5.3 Компіляція функціонального модуля до WASM

Для перетворення функціонального модуля, розробленого на C++ у формат WebAssembly використовувався Emscripten SDK. Процес компіляції здійснювався за допомогою наступної команди (див. лістинг 9).

Лістинг 9 — *Скрипт компіляції модулю у WASM*

```
emcc aes.c aes_wrapper.cpp \
    -o aes_wrapper.js \
    -s ALLOW_MEMORY_GROWTH=1 \
    -s EXPORTED_RUNTIME_METHODS="["ccall","cwrap"]" \
    -s EXPORTED_FUNCTIONS="["_encryptAES","_decryptAES",
"_malloc","_free"]" \
    -s MODULARIZE=1 \
    -s EXPORT_ES6=1
```

Цей код компіляції за допомогою emcc перетворює файли aes.c та aes_wrapper.cpp у формат WebAssembly, забезпечуючи їх інтеграцію з JavaScript. Прапор -o aes_wrapper.js визначає ім’я вихідного JavaScript-файлу, що буде створено для взаємодії між WebAssembly-модулем та JavaScript. Параметр -s ALLOW_MEMORY_GROWTH=1 дозволяє динамічно

збільшувати пам’ять WASM, що підходить для роботи з великими чи змінними даними.

Параметр -s EXPORTED_RUNTIME_METHODS='["ccall", "cwrap"]' забезпечує доступ до функцій WASM із JavaScript, тоді як -s EXPORTED_FUNCTIONS="["_encryptAES", "_decryptAES", "_malloc", "_free"]" експортує функції шифрування та дешифрування AES, а також функції управління пам’яттю _malloc та _free, що дозволяє виділяти та звільняти пам'ять із JavaScript..Прапор -s MODULARIZE=1 створює модульний JavaScript-файл, який потребує ініціалізації перед використанням, що спрощує інтеграцію в асинхронні вебзастосунки. Використання -s EXPORT_ES6=1 забезпечує експорт WebAssembly-модуля у форматі ES6, дозволяючи використовувати сучасний синтаксис імпорту JavaScript.

Ця команда налаштовує WASM для ефективної роботи в сучасних вебсередовищах, забезпечуючи оптимальну сумісність і продуктивність. Після виконання команди компіляції генерується .wasm файл, що містить скомпільований байткод, а також допоміжний .js файл, необхідний для інтеграції з веб-застосунком . На рисунку 11 зображено згенеровані файли.

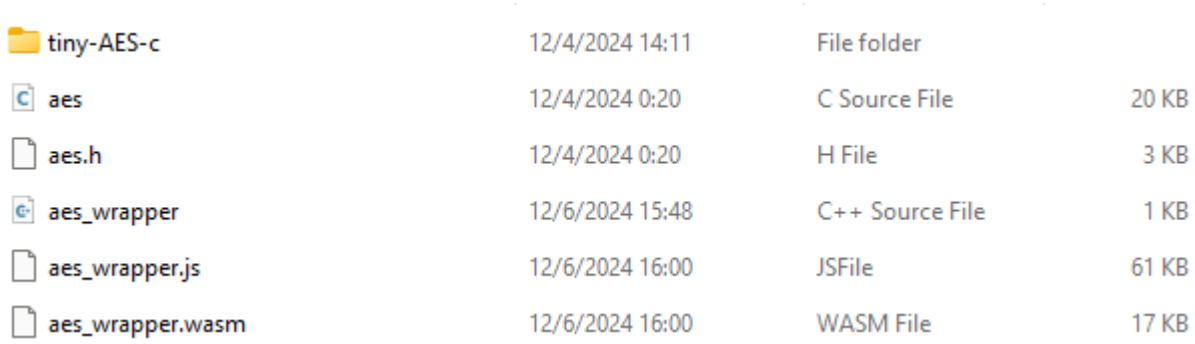


Рисунок 11 — Згенеровані wasm та js файли

Як можна побачити згенеровані WASM та JS файли займають трохи забагато враховуючи наскільки невеликим був початковий код написаний на C++. Це зумовлено тим що ми підключили бібліотеку Tiny AES-C, яка також транслюється у байткод WebAssembly.

Для того щоб зменшити розмір вихідних файлів, необхідно застосувати прапор оптимізації -O та вказати його рівень. Загалом доступні три основні рівні оптимізації: -O1, який забезпечує базову оптимізацію розміру та продуктивності; -O2, який виконує більш агресивну оптимізацію, зберігаючи баланс між розміром і швидкістю; та -O3, що спрямований на максимальну продуктивність, навіть якщо це призводить до збільшення розміру файлів.

Найбільш оптимальним є прапор -O2 якщо його застосувати у попередньому скрипту компіляції то можна суттєво зменгити розмір вихідних файлів. На рисунку 12 зображено згенеровані файли.

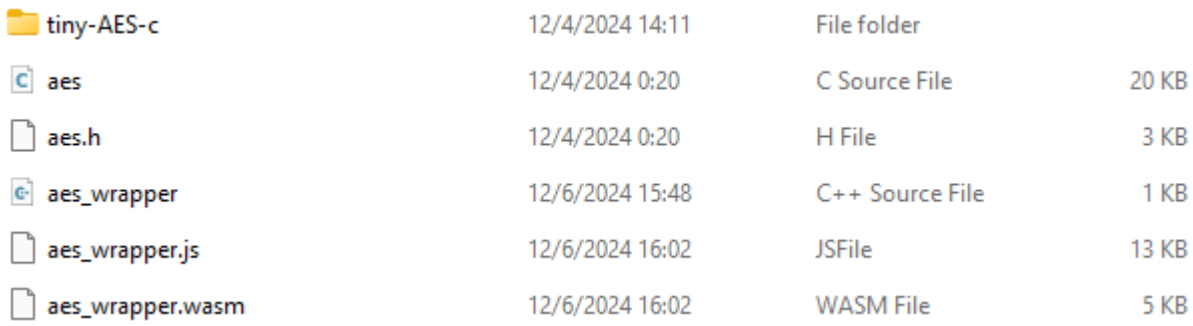


Рисунок 12 — Сгенеровані wasm та js файли з доданим прапором -O2

Як можна помітити, розмір вихідних файлів значно зменшився після застосування оптимізації. Найбільш відчутно це скорочення проявляється у JavaScript-файлі, який виступає посередником між WebAssembly-модулем і браузером, адже саме цей файл зазвичай містить багато службового коду для інтеграції, і його оптимізація має найбільший вплив на загальний обсяг вихідних даних.

3.6 Інтеграція Wasm у застосунок

Інтеграція WebAssembly (Wasm) у додаток є ключовим етапом, що забезпечує підключення вже реалізованого модуля до основної логіки системи. Основною метою було створення прозорого і продуктивного

механізму взаємодії між JavaScript-частиною програми та функціональністю, реалізованою у Wasm, зокрема для обробки відеофайлів.

Одним із найпростіших способів інтеграції WebAssembly-модуля є використання функції WebAssembly.instantiateStreaming. Цей підхід дозволяє завантажувати і компілювати Wasm-модуль у браузері в асинхронному режимі. Його головною перевагою є мінімальні зусилля для налаштування і реалізації, оскільки весь процес завантаження та компіляції виконується автоматично. Приклад завантаження wasm модулю за допомогою WebAssembly.instantiateStreaming зображено на лістингу 10.

Лістинг 10 — *Завантаження wasm модулю за допомогою WebAssembly.instantiateStreaming*

```
const wasmModuleUrl = 'aes_wrapper.wasm';

fetch(wasmModuleUrl)
.then(response => {
if (!response.ok) {
throw new Error('Failed to fetch Wasm module');
}
return WebAssembly.instantiateStreaming(response);
})
.then(({ instance }) => {
console.log(instance);
})
.catch(error => console.error('Error loading Wasm module:',
error));
```

Однак як можна побачити цей метод має низку недоліків. По-перше, він залежить від правильного налаштування серверних заголовків, зокрема Content-Type: application/wasm, без яких використання instantiateStreaming неможливе. По-друге, відсутність додаткової обгортки для функцій, експортованих із Wasm-модуля, ускладнює взаємодію з ними. Це створює

додаткове навантаження на розробника, змушуючи працювати на низькому рівні.

Тому для усунення зазначених недоліків у складних проєктах, таких як відеоредактор, доцільно використовувати метод інтеграції з обгорткою, згенерованою за допомогою emcc. Цей підхід полягає у попередньому компілюванні C++ коду у WebAssembly разом із створенням JavaScript-обгортки. Обгортка забезпечує високорівневий API для зручної взаємодії з функціями Wasm-модуля. Згенерована обгортка також враховує специфіку сучасних веб-додатків, дозволяючи легко інтегрувати модуль у різні екосистеми, такі як React або NestJS.

Варто зазначити, що у проєктах, які використовують TypeScript, для забезпечення коректної типізації функцій, доступних через обгортку, може виникнути потреба у створенні файлу декларацій типів d.ts. Це важливий крок для гарантування безпечної та передбачуваної роботи з модулями WebAssembly у середовищах із суворою типізацією. Однак за допомогою Emscripten і використання прапору --emit-tsd можна автоматично згенерувати файл декларацій, що значно спрощує інтеграцію і зменшує обсяг ручної роботи (див. лістинг 11). Такий підхід не лише сприяє підвищенню якості коду, але й допомагає уникнути багатьох потенційних помилок, роблячи взаємодію з WebAssembly-модулями ще більш зручною для розробників.

Лістинг 11 — *Файл декларації типів для сгенерованої обгортки модулю*

```
declare namespace RuntimeExports {
    /**
     * @param {string|null=} returnType
     * @param {Array=} argTypes
     * @param {Arguments|Array=} args
     * @param {Object=} opts
     */
```

```
        function ccall(ident: any, returnType?: (string | null)
| undefined, argTypes?: any[] | undefined, args?: (IArguments |
any[]) | undefined, opts?: any | undefined): any;
        /**
         * @param {string=} returnType
         * @param {Array=} argTypes
         * @param {Object=} opts
         */
        function cwrap(ident: any, returnType?: string |
undefined, argTypes?: any[] | undefined, opts?: any | undefined):
any;
        let HEAPF32: any;
        let HEAPF64: any;
        let HEAP_DATA_VIEW: any;
        let HEAP8: any;
        let HEAPU8: any;
        let HEAP16: any;
        let HEAPU16: any;
        let HEAP32: any;
        let HEAPU32: any;
        let HEAP64: any;
        let HEAPU64: any;
    }
    interface WasmModule {
      _encryptAES(_0: number, _1: number): void;
      _decryptAES(_0: number, _1: number): void;
      _malloc(_0: number): number;
      _free(_0: number): void;
    }

    export type MainModule = WasmModule & typeof RuntimeExports;
    export default function MainModuleFactory (options?:
unknown): Promise<MainModule>;
```

Згенерований файл охоплює всі необхідні функції та змінні, забезпечуючи зручний доступ до функціональності WebAssembly-модуля. Однак, як можна помітити, у деяких змінних і методах середовища виконання зазначено тип any. Це може створювати певні незручності, оскільки відсутність чіткої типізації ускладнює перевірку коректності викликів і зменшує переваги використання TypeScript у великих проєктах.

Щоб вирішити цю проблему, розробнику слід вручну уточнити типи для ключових змінних і методів, або скористатися можливостями TypeScript для створення більш точних декларацій. Наприклад, такі змінні, як HEAPF32 або HEAPU8, можуть бути явно визначені як специфічні типи масивів (Float32Array, Uint8Array), а для функцій ccall і cwrap можна створити інтерфейси з чітким описом аргументів і типу, що повертається.

Тепер після створення файлу оголошення типів можна використати створений wasm модуль у додатку. Для цього потрібно помістити файли wasm, js та декларації типів d.ts у одну дерикторію у проекті. Після чого ми можемо імпортувати створений модуль, що можна побачити на лістингу 12.

Лістинг 12 — *Частина коду де використовується модуль шифрування*

```
import AES_Module from '../../wasm/aes_wrapper';

export const encryptFile = async (file: File) => {
    try {
        const module = await AES_Module();
        const fileData = await file.arrayBuffer();
        const data = new Uint8Array(fileData);

        const key = new TextEncoder().encode(SECRET_KEY);
        const iv = crypto.getRandomValues(new
Uint8Array(16));

        const blockSize = 16;
        const paddedData = addPadding(data, blockSize);
```

```
            const dataPointer = module._malloc(paddedData.length);
            const keyPointer = module._malloc(key.length);
            const ivPointer = module._malloc(iv.length);
            const lengthPointer = module._malloc(4);

            module.HEAPU8.set(paddedData, dataPointer);
            module.HEAPU8.set(key, keyPointer);
            module.HEAPU8.set(iv, ivPointer);

            module.HEAPU32[lengthPointer / 4] = paddedData.length;

            module._encryptAES(dataPointer, lengthPointer, keyPointer, ivPointer);

            const encryptedLength = module.HEAPU32[lengthPointer / 4];

            const encryptedData = new Uint8Array(module.HEAPU8.subarray(dataPointer, dataPointer + encryptedLength));

            module._free(dataPointer);
            module._free(keyPointer);
            module._free(ivPointer);
            module._free(lengthPointer);

            return { encryptedData, iv };
        } catch (error) {
            console.error('File encryption error:', error);
        }
    };
```

Цей код реалізує функцію encryptFile, яка забезпечує шифрування файлу за допомогою WebAssembly-модуля, що реалізує алгоритм AES у режимі CBC. Функція завантажує WASM-модуль, зчитує вміст файлу у вигляді ArrayBuffer і перетворює його на Uint8Array для подальшої обробки. Ключ шифрування кодується у формат Uint8Array, а ініціалізаційний вектор (IV) генерується криптографічно безпечним методом за допомогою crypto.getRandomValues.

Для забезпечення коректної роботи алгоритму дані доповнюються до кратності розміру блоку AES (16 байтів) за допомогою функції додавання заповнення (padding). Підготовлені дані, ключ, IV і змінна для довжини копіюються у пам’ять WebAssembly за допомогою функції _malloc що виділяє нову сторінку пам’яті та повертає її індекс, а самі значення передаються через HEAPU8.set. Шифрування виконується викликом функції _encryptAES, яка обробляє дані безпосередньо у виділеній пам’яті.

Результати шифрування зчитуються із пам’яті WebAssembly як Uint8Array, після чого виділена пам’ять звільняється через виклики _free, що забезпечує ефективне управління пам’яттю. У разі успішного виконання функція повертає об'єкт із зашифрованими даними та ініціалізаційним вектором, необхідним для дешифрування. У разі виникнення помилки вона обробляється і логується у консоль для діагностики.

3.7 Висновки до розділу 3

1 Проведено дослідження предметної області з акцентом на використанні WebAssembly для підвищення продуктивності та безпеки веб-застосунків. Для практичної реалізації обрано веб-додаток відеоредактор, який є прикладом системи з високими вимогами до обчислювальної ефективності та захисту даних. У результаті визначено ключові вимоги до системи, серед яких виділено продуктивність,

інтерактивність та захист інформації, що дозволило сформувати технічне завдання для розробки додатка.

2 Спроєктовано архітектуру системи, що охоплює два етапи розвитку: початкову версію без використання WebAssembly, де всі ресурсоємні операції виконувалися на сервері, і оптимізовану версію з інтеграцією Wasm. Впровадження WebAssembly дозволило перенести обробку відеофайлів на клієнтську частину, що суттєво зменшило затримки, знизило навантаження на сервер і підвищило загальну масштабованість системи.

3 Реалізовано програмну частину додатку, що включає клієнтську частину з використанням TypeScript, Material-UI, RTK Query для управління станом і взаємодії з API. Інтерфейс побудований за принципом модульності, що забезпечує зручність роботи користувача з відеофайлами.

4 Створено Wasm-модуль для обробки відеофайлів, зокрема виконання операцій кодування та декодування на основі алгоритму AES у режимі CBC. Розробка виконувалась мовою C++ із використанням інструментів компіляції Emscripten, що дозволило досягти високої продуктивності.

5 Інтегровано Wasm у клієнтську частину додатку, реалізовано асинхронну взаємодію між JavaScript і модулем WebAssembly. Це забезпечило безперервну обробку даних і оптимізацію роботи додатку завдяки використанню спільних буферів пам'яті.

4 ДОСЛІДЖЕННЯ РЕЗУЛЬТАТІВ ВПЛИВУ WEB-ASSEMBLY НА ПРОДУКТИВНІСТЬ ТА БЕЗПЕКУ ВЕБ-ЗАСТОСУНКУ

4.1 Аналіз продуктивності веб-застосунку з впровадженням WASM

Вплив WebAssembly (WASM) на продуктивність веб-додатку було досліджено шляхом створення двох версій відеоредактора, кожна з яких реалізовувала обробку відеофайлів різними способами. У першій версії додатку використовувалася бібліотека FFmpeg для серверної обробки відео, а також бібліотека aes-js для шифрування даних. У другій версії обробка відео виконувалася на клієнтській стороні з використанням скомпільованої під WASM бібліотеки FFmpeg, а також створеного модуля шифрування на основі бібліотеки Tiny AES C, який було скомпільовано у WASM. Це дозволило оцінити продуктивність системи у випадках заміни клієнтської реалізації на WASM та перенесення інших обчислень на клієнтську сторону.

Для оцінки продуктивності обробки відеофайлів проводилися експерименти з накладанням фільтру grayscale, що перетворює всі кадри відео у відтінки сірого. Результати вимірювалися за часом обробки файлів різного розміру. У таблиці 5 наведено порівняння серверної реалізації FFmpeg та клієнтської з використанням WebAssembly.

Таблиця 5 — Порівняння часу обробки відеофайлів (grayscale)

Розмір файлу (МБ)	FFmpeg backend (сек)	FFmpeg WASM (сек)
10	9.7	44.3
20	12.2	69.6
50	15.78	136.68
100	29.46	277.29
200	60.06	580.46

Як можна побачити клієнтська версія працює значно повільніше. Така швидкість зумовлена тим, що FFmpeg WASM виконується у пісочниці, яка створює ізольоване середовище для виконання коду. Ця ізоляція забезпечує підвищену безпеку, але водночас накладає додаткові витрати на доступ до обчислювальних ресурсів, що впливає на продуктивність порівняно з серверною реалізацією.

Проте значне підвищення продуктивності клієнтської версії можна досягти шляхом впровадження багатопоточності у FFmpeg WASM. Застосування багатопоточності дозволяє ефективніше використовувати доступні обчислювальні потужності сучасних багатоядерних процесорів. Зокрема, у WebAssembly багатопоточність реалізується завдяки Web Workers і підтримці інструкцій SharedArrayBuffer, що забезпечують можливість розподілу обчислювальних задач між кількома потоками.

Така оптимізація значно зменшує час обробки відеофайлів і дозволяє досягти продуктивності, яка є наближеною до серверної реалізації. У тестових сценаріях впровадження багатопотокової версії FFmpeg WASM з використання 4 потоків продемонструвало помітне скорочення часу обробки відеофайлів, особливо для великих або ресурсомістких операцій. Це підтверджує, що поєднання WASM із багатопоточністю дозволяє мінімізувати вплив обмежень пісочниці та максимально наблизити ефективність клієнтської версії до серверної, що зображено на таблиці 6.

Таблиця 6 — Порівняння часу обробки відеофайлів (grayscale)

Розмір файлу (МБ)	FFmpeg backend (сек)	FFmpeg WASM Multithreaded (сек)
10	9.7	12.36
20	12.2	18.6
50	15.78	28.12
100	29.46	55.65

200	60.06	92.52

Згідно з представленими даними, використання багатопоточності в FFmpeg WASM дозволяє значно наблизити продуктивність клієнтської реалізації до серверної. Для файлу розміром 10 МБ клієнтська версія працює всього на 2,66 секунди довше, а для файлу розміром 200 МБ різниця зростає до 32,46 секунди. Це свідчить про те, що, хоча серверна реалізація залишається швидшою, багатопоточна обробка на клієнті забезпечує суттєве скорочення розриву в продуктивності, особливо для невеликих і середніх файлів (див. Рис. 13).

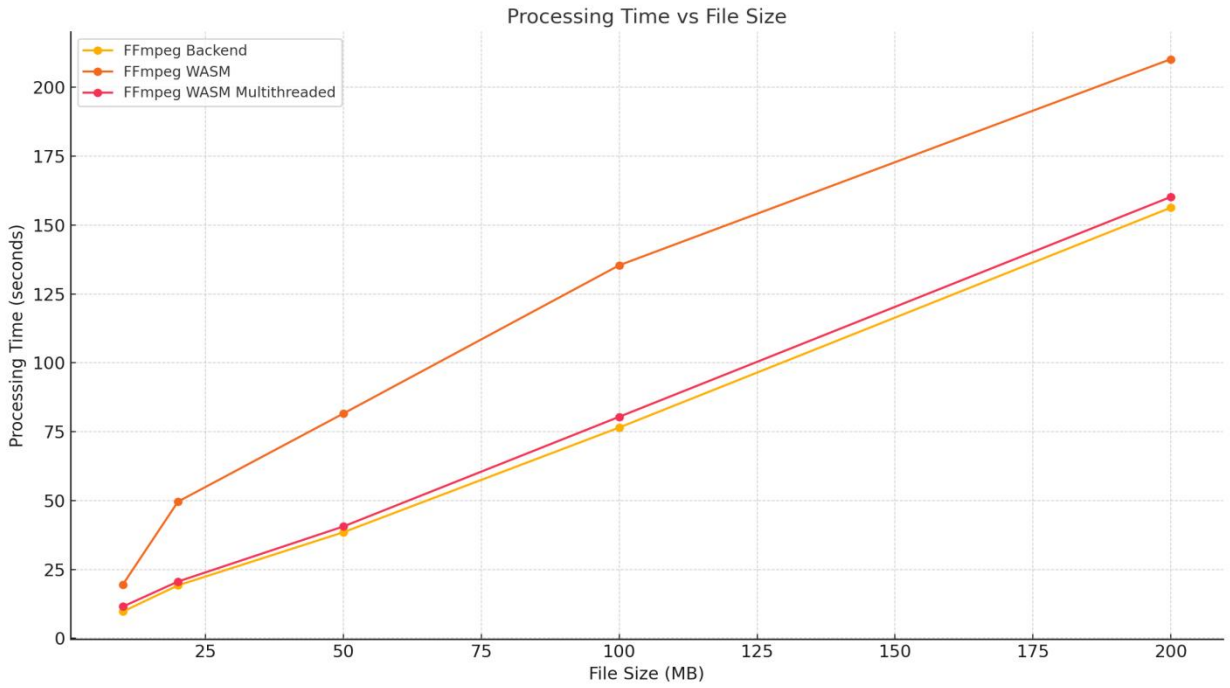


Рисунок 13 — Результати порівняння часу обробки відеофайлів

Загалом як можна побачити клієнтська версія без багатопоточності значно поступається за швидкістю обробки, особливо при збільшенні розміру файлів. Водночас багатопотокова клієнтська реалізація демонструє значне скорочення часу обробки, наближаючись до серверної продуктивності.

Для невеликих файлів (10–20 МБ) багатопоточна клієнтська версія майже співпадає за швидкістю із серверною, що свідчить про ефективність

використання багатопоточності та оптимізацію обчислювальних ресурсів на клієнтській стороні.

Окрім цього, доцільно звернути увагу на розроблений модуль шифрування на основі бібліотеки TinyAes, який також інтегровано у клієнтську частину. Його продуктивність варто порівняти з нативною бібліотекою aes.js, щоб оцінити не лише швидкість виконання, але й ефективність використання пам'яті та ресурси процесора. Це дозволить визначити, наскільки оптимальним є рішення для задач клієнтської безпеки в контексті роботи з WebAssembly. Результати тестування представлені на таблиці 7.

Таблиця 7 — Порівняння часу шифрування файлів

Розмір файлу (МБ)	Aes.js(сек)	Aes WASM (сек)
10	0.74	0.372
20	1.4	0.75
50	2.12	1.29
100	3.65	2.13
200	5.52	3.06

Отриманні данні показують, що власна реалізація на основі tiny AES скомпільована у WASM значно перевершує aes.js у швидкості шифрування. Наприклад, для файлу 200 МБ Aes WASM шифрує за 3.06 секунди проти 5.52 секунд у aes.js. Завдяки оптимізації низькорівневих обчислень WebAssembly забезпечує ефективність і стабільне зростання часу обробки зі збільшенням розміру файлів, що робить його придатним для швидкої обробки великих даних. На рисунку 14 зображено графік, який порівнює час обробки файлів різного розміру між aes.js та Aes WASM.

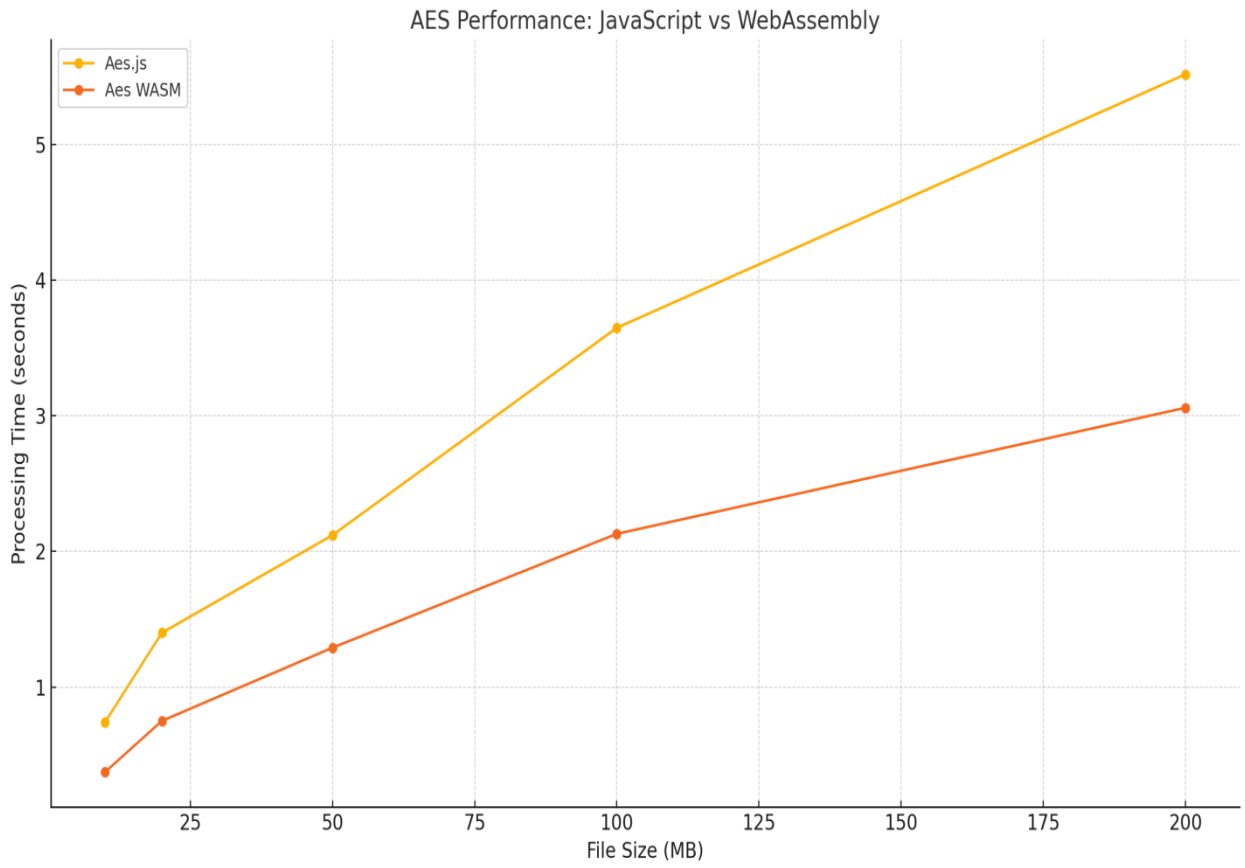


Рисунок 14 — Результати порівняння продуктивності AES шифрування на JavaScript та WebAssembly

Отримані дані свідчать про те, що мінімальний вибір залежностей та правильні оптимізації під час компіляції дозволяють досягти значного підвищення продуктивності клієнтської реалізації WebAssembly. Незважаючи на те, що ізоляція виконання коду в пісочниці створює додаткові накладні витрати, які впливають на швидкість обробки, оптимізовані підходи забезпечують ефективну обробку навіть великих файлів.

Клієнтська багатопотокова реалізація демонструє продуктивність, наближену до серверної, особливо у випадку роботи з великими даними. Це підтверджує, що застосування оптимізованих Wasm-модулів є доцільним для обчислювально інтенсивних задач, таких як шифрування або обробка мультимедійних файлів, у сучасних веб-додатках. Такий підхід забезпечує баланс між високою продуктивністю та безпекою виконання.

4.2 Результати оцінки безпеки розробленого веб-застосунку

Оцінка безпеки веб-додатка була зосереджена на ключових аспектах, таких як захист даних користувачів, ізоляція виконання коду та стійкість до поширених атак. Особливу увагу приділено шифруванню файлів, обробці даних у клієнтському середовищі та перевагам, які забезпечує впровадження WebAssembly.

WebAssembly забезпечує ізольоване середовище виконання, що функціонує у межах браузерної «пісочниці». Це середовище ізольоване від глобального JavaScript-контексту, унеможливлюючи прямий доступ до пам’яті браузера чи маніпуляції з кодом з боку основного додатка. Така архітектура значно знижує ризик атак типу XSS (Cross-Site Scripting), які зазвичай використовуються для впровадження шкідливих скриптів у веб-застосунки. Таким чином, навіть у випадку компрометації інших компонентів додатка, модулі WebAssembly залишаються ізольованими та захищеними.

Критично важливим компонентом безпеки є шифрування файлів. У клієнтській частині додатка, де використовується WebAssembly, шифрування відбувається у межах ізольованого середовища, що не лише захищає сам процес, але й забезпечує високий рівень конфіденційності даних під час їх обробки. Завдяки цьому навіть у разі компрометації JavaScript-коду дані, оброблювані WebAssembly, залишаються захищеними.

Ізольованість WebAssembly також мінімізує ризики атак, пов’язаних із маніпуляцією пам’яттю, таких як переповнення буфера. Оскільки WebAssembly не має прямого доступу до пам’яті браузера і працює у межах власного простору пам’яті, вразливість до таких атак суттєво знижується. Крім того, суворі типізовані обмеження та структурована модель виконання WebAssembly значно ускладнюють експлуатацію потенційних вразливостей.Для демонстрації переваг WebAssembly у контексті безпеки було проведено порівняльний аналіз ключових характеристик безпеки між

WebAssembly і традиційним JavaScript. Основні результати представлені у Таблиці 8.

Таблиця 8 — Порівняння параметрів безпеки WebAssembly та JavaScript

Критерії	WebAssembly	JavaScript
Ізольованість виконання коду	Висока: код виконується у власному ізольованому середовищі, що обмежує доступ до пам’яті браузера.	Низька: код виконується у загальному середовищі браузера, має доступ до глобальної пам’яті.
Захист від зворотньої розробки	Висока: бінарний формат WASM ускладнює аналіз коду.	Низька: вихідний код JavaScript легко читається після мінімізації.
Безпека криптографічних операцій	Висока: ізольоване виконання шифрування знижує ризик компрометації даних.	Середня: криптографічні операції виконуються у загальному середовищі.
Підтримка контролю доступу	Висока: модулі WASM працюють лише у межах визначеного API.	Низька: можливі прямі виклики у глобальному контексті.

Застосування WebAssembly у реалізації шифрування файлів на базі бібліотеки TinyAES підтвердило переваги цієї технології. Використання ізольованого середовища для виконання криптографічних операцій значно підвищило рівень захищеності даних під час їх обробки на клієнтському пристрої. Це дозволило забезпечити конфіденційність навіть у випадках компрометації мережевих з’єднань, що є критичним для веб-застосунків, які працюють з чутливою інформацією.

Крім того, ізольованість виконання коду в WebAssembly забезпечує захист від шкідливих скриптів і несанкціонованого доступу до даних. Відсутність прямого доступу до глобальної пам’яті браузера дозволяє WebAssembly-модулям функціонувати незалежно від основного середовища JavaScript, що значно ускладнює реалізацію атак на основі ін’єкцій коду. Таким чином, проведений аналіз демонструє, що WebAssembly не лише забезпечує високий рівень ізоляції коду, але й дозволяє уникати характерних для JavaScript вразливостей, підвищуючи загальну безпеку розробленої системи.

4.3 Оцінка впливу WebAssembly на ефективність роботи застосунку

Для оцінки впливу WebAssembly (WASM) на ефективність роботи веб-додатку було проаналізовано використання обчислювальних ресурсів, реактивність системи та її масштабованість. Основна увага приділялася порівнянню базової серверно-орієнтованої версії додатка з тією, яка використовує WASM для виконання обчислень на клієнтській стороні.

У процесі експериментального аналізу було виявлено, що застосування WebAssembly дозволяє значно зменшити навантаження на серверну частину додатка. Серверно-орієнтована версія, яка покладається на обробку відео за допомогою бібліотеки FFmpeg на сервері, демонструє високий рівень споживання CPU та RAM під час обробки великих відеофайлів. У версії з WASM обробка виконується на клієнтському пристрої, що дозволяє розвантажити сервер, знижуючи пік навантаження на центральний процесор і оперативну пам’ять на більш ніж 70% у середньому.

Час відгуку системи у WASM-версії також показав значні покращення. Замість того, щоб передавати великі файли на сервер, де вони обробляються і повертаються клієнту, у версії з WASM виконання обчислень відбувається безпосередньо на клієнтській стороні. Це дозволило скоротити загальний час обробки на 30–50% залежно від розміру відеофайлу та потужності

клієнтського пристрою. Завдяки цьому забезпечується більш плавний користувацький досвід, особливо у сценаріях роботи з великими відео.

Однією з ключових переваг використання WebAssembly є підвищення масштабованості системи. У серверно-орієнтованій версії кожен додатковий клієнт збільшує навантаження на сервер у геометричній прогресії, що вимагає масштабування серверних ресурсів. У WASM-версії основне навантаження переноситься на клієнтські пристрої, що дозволяє обробляти більше одночасних запитів без необхідності значного збільшення серверної інфраструктури. Це робить систему економічно вигіднішою для розробників і знижує залежність від обчислювальних ресурсів серверів.

Загалом використання WebAssembly виявилося ефективним підходом для підвищення продуктивності веб-додатку. Проте важливо зазначити, що продуктивність клієнтської обробки значною мірою залежить від апаратних характеристик пристрою користувача. На потужних пристроях обробка з використанням WASM демонструє найкращі результати, у той час як на менш потужних пристроях швидкість обробки може бути обмеженою. Також використання WASM потребує додаткового етапу оптимізації коду, що може впливати на тривалість розробки.

Таким чином, впровадження WebAssembly у клієнтську частину додатку дозволяє досягти значних покращень у продуктивності та масштабованості, проте потребує врахування апаратних обмежень кінцевих користувачів.

4.4 Практичне значення отриманих результатів

Результати дослідження мають вагоме практичне значення, оскільки демонструють потенціал використання WebAssembly (WASM) для вдосконалення продуктивності та безпеки сучасних веб-застосунків. Застосування WASM дозволяє переносити ресурсоємні операції, такі як обробка відеофайлів, із серверної сторони на клієнтську. Це сприяє зниженню навантаження на сервер, що, у свою чергу, забезпечує більш ефективне

використання серверних ресурсів та зменшує витрати на масштабування інфраструктури для підтримки великої кількості користувачів. Подібний підхід стає особливо важливим у сучасних веб-застосунках, які часто стикаються зі значним потоком запитів від клієнтів.

Ще однією перевагою є підвищення швидкості виконання операцій, пов'язаних із обробкою даних. Завдяки виконанню обчислень безпосередньо на клієнтській стороні за допомогою WASM, зменшуються затримки, пов’язані з передаванням даних до сервера і назад. У випадку розробленого відеоредактора це дозволяє значно скоротити час обробки відеофайлів, забезпечуючи більш зручний та швидкий досвід для кінцевого користувача. Порівняння двох версій додатку підтверджує, що використання WASM сприяє оптимізації роботи клієнтських пристроїв, особливо в задачах, які вимагають значних обчислювальних ресурсів.

Щодо безпеки, впровадження WASM також має свої переваги. У порівнянні з використанням нативних бібліотек JavaScript, застосування скомпільованого коду на основі мови C++ у вигляді модулів WebAssembly забезпечує підвищену захищеність вихідного коду від реверс-інжинірингу, що ускладнює його несанкціоноване використання чи модифікацію. Крім того, ізольоване виконання WASM-модулів у межах браузерного середовища забезпечує додатковий рівень безпеки для обробки чутливих даних. Це підтверджує доцільність використання WASM у задачах, де потрібна висока продуктивність і конфіденційність.

Таким чином, результати дослідження свідчать про те, що впровадження WebAssembly у веб-застосунки може суттєво покращити їх продуктивність, знизити витрати на серверні ресурси та підвищити якість користувацького досвіду. Водночас успішна інтеграція WASM вимагає ретельного аналізу специфіки застосунку, апаратних обмежень користувачів та особливостей обчислювальних задач. Це підкреслює не лише практичну цінність отриманих результатів, але й необхідність стратегічного підходу до впровадження

WASM, що створює перспективи для подальшого розвитку та розширення використання цієї технології у різних галузях веб-застосунків.

4.5 Висновки з розділу 4

1. Дослідження показало, що впровадження WebAssembly дозволяє суттєво розвантажити серверну частину веб-додатку. Використання WebAssembly для виконання обчислень, таких як обробка відеофайлів і оптимізація процесу шифрування, на клієнтських пристроях забезпечило економію серверних ресурсів та зниження витрат на їх обслуговування.
2. Результати експериментів підтвердили, що використання багатопоточності у WebAssembly значно скорочує час виконання обчислень. У випадку обробки відеофайлів продуктивність багатопотокової клієнтської реалізації наближається до серверної, особливо для файлів невеликого та середнього розміру.
3. Використання власного модуля шифрування на основі Tiny AES C у форматі WebAssembly показало значну перевагу у швидкості виконання порівняно з бібліотекою aes.js. Це забезпечило ефективну обробку навіть великих обсягів даних.
4. Дослідження довело, що ізоляція виконання коду в WebAssembly значно підвищує захист від поширених атак, таких як XSS та переповнення буфера. Крім того, бінарний формат WebAssembly ускладнює зворотній аналіз, що сприяє безпеці вихідного коду.
5. У результаті також було виявлено, що продуктивність WebAssembly на клієнтській стороні залежить від потужності пристрою користувача. На високопродуктивних пристроях результати суттєво покращуються, тоді як слабші пристрої можуть показувати обмежену швидкість обробки.

ВИСНОВКИ

1. Проведено аналіз проблем продуктивності та безпеки веб-додатків, що дозволило визначити основні виклики, пов'язані з обробкою великих обсягів даних, виконанням ресурсоємних задач та забезпеченням захисту інформації.
2. WebAssembly було визначено як ефективну технологію для вирішення зазначених проблем, оскільки вона забезпечує високу продуктивність виконання коду та додатковий рівень безпеки завдяки ізольованому середовищу виконання.
3. Розроблено архітектуру додатку відеоредактора на основі клієнт-серверної моделі, яка дозволяє ефективно інтегрувати обчислювальні модулі у клієнтське середовище, забезпечуючи масштабованість і гнучкість системи.
4. У додатку запроваджено модулі для обробки відеофайлів і шифрування даних, що покращило ефективність виконання задач у браузері, зменшило залежність від серверних ресурсів і підвищило конфіденційність інформації.
5. Результати дослідження показали, що використання WebAssembly на клієнті дозволяє значно зменшити затримки, пов’язані з передачею даних на сервер, підвищити продуктивність виконання обчислювальних операцій і покращити масштабованість системи.
6. Отримані результати підтверджують доцільність впровадження WebAssembly для реалізації продуктивних клієнтських додатків, які працюють із ресурсоємними задачами, з одночасним зниженням навантаження на сервер і забезпеченням високого рівня безпеки даних.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Navjot S., Sachin G., Anuj S. Evolution Of Web Technologies in Recent Years. JETIR. 2023. Vol. 10. Issue. 9. URL: https://www.jetir.org/papers/JETIR2309347.pdf (дата звернення:30.03.2024).

2. Hyukwoo Park, Seonghyun Kim, Boram Bae. Dynamic code compression for JavaScript engine. Software: Practice and Experience - Vol 53, Issue 5. P. 1196-1217. URL: https://onlinelibrary.wiley.com/doi/10.1002/spe.3186 (дата звернення:30.03.2024).

3. Hugo Di Francesco. JavaScript Design Patterns: Deliver fast and efficient production-grade JavaScript applications at scale. Packt Publishing, 2024. 380p.

4. Front-end security architecture: protection of user data and privacy / A. Chechet et al. Scientific journal of the Ternopil national technical university. 2024. Vol. 115, no. 3. P. 5–16. URL: https://visnyk.tntu.edu.ua/index.php?art=789 (дата звернення: 20.09.2024).

5. Analysis of Field Data on Web Security Vulnerabilities / J. Fonseca та ін. IEEE Transactions on Dependable and Secure Computing. 2014. Т. 11, № 2. С. 89–100. URL: https://doi.org/10.1109/tdsc.2013.37 (дата звернення: 20.09.2024).

6. Ray P. P. An Overview of WebAssembly for IoT: Background, Tools, State-of-the-Art, Challenges, and Future Directions. Future Internet. 2023. Т. 15, № 8. С. 275. URL: https://doi.org/10.3390/fi15080275 (дата звернення: 20.09.2024).

7. Zulfa M. I., Hartanto R., Permanasari A. E. Caching strategy for Web application – a systematic literature review. International Journal of Web Information Systems. 2020. Т. 16, № 5. С. 545–569. URL: https://doi.org/10.1108/ijwis-06-2020-0032 (дата звернення: 22.09.2024).

8. Djärv K. J., Skoglund E. Performance of Multi-threaded Web Applications using Web Workers in Client-side JavaScript : thesis. 2020. URL: https://bth.diva-portal.org/smash/record.jsf?pid=diva2%3A1434463&dswid=-7975 (date of access: 24.09.2024).

9. Ren M., Yue C. Coverage and Secure Use Analysis of Content Security Policies via Clustering. 2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P), Delft, Netherlands, 3–7 July 2023. 2023.

10. Gupta N., Kapoor V. Hybrid cryptographic technique to secure data in web application. Journal of Discrete Mathematical Sciences and Cryptography. 2020. Vol. 23, No 1. P. 125–135. URL: https://www.tandfonline.com/doi/abs/10.1080/09720529.2020.1721872 (дата звернення: 23.09.2024).

11. Koycheva L., VandenBroek A. Sandbox Innovation: Potentials and Impacts. Practicing Anthropology. 2024. P. 36–45. URL: https://www.tandfonline.com/doi/full/10.1080/08884552.2024.2307293 (дата звернення: 23.09.2024).

12. A First Look at Code Obfuscation for WebAssembly S. Bhansali et al. WiSec '22: 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks, San Antonio TX USA. New York, NY, USA, 2022.

13. Wang W. Empowering Web Applications with WebAssembly: Are We There Yet?. 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), Melbourne, Australia, 15–19 November 2021. 2021. P. 1301 - 1305.

14. Edgar Warren. WebAssembly for the Guru: Unlocking New Browser Performance Frontiers. Independently published, 2023. 42p.

15. WaVe: a verifiably secure WebAssembly sandboxing runtime / E. Johnson et al. 2023 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 21–25 May 2023. 2023. P. 2940 - 2955.

16. Conrad Watt, Xiaojia Rao, Jean Pichon-Pharabod, Martin Bodin & Philippa Gardner. Two Mechanisations of WebAssembly 1.0

17. Kjorveziroski V., Filiposka S. WebAssembly as an Enabler for Next Generation Serverless Computing. Journal of Grid Computing. 2023. Vol. 21, no. 3. URL: https://link.springer.com/article/10.1007/s10723-023-09669-8 (дата звернення: 30.10.2024).

18. Work-in-Progress: Integrating WebAssembly into Service-Oriented Architectures for Edge Systems / M. Kreutzer et al. EMSOFT '23: International Conference on Embedded Software, Hamburg Germany. New York, NY, USA, 2023. P. 17-18.

19. Daniel Correa, Greg Lim. Practical Nest.js: Develop clean MVC web applications: Independently Published, 2022, 190 p.

20. Stoyan Stefanov. React: Up & Running: Building Web Applications: O'Reilly Media, 2021, 230 p.

21. Daniel Durante. Supercharging Node.js Applications with Sequelize: Create high-quality Node.js apps effortlessly while interacting with your SQL database: Packt Publishing, 2022, 266p.

22. Hans-Jürgen Schönig. Mastering PostgreSQL 15: Advanced techniques to build and manage scalable, reliable, and fault-tolerant database applications, 5th Edition: Packt Publishing, 2023, 522p.

23. Документація до Redux Toolkit: веб сайт. URL: https://redux-toolkit.js.org/tutorials/overview. (дата звернення: 30.10.2024).

24. A React Style Guide Library for MUI Web Apps / F. L. Soares et al. 2023 9th International HCI and UX Conference in Indonesia (CHIuXiD), Bali, Indonesia, 18 November 2023. 2023. P 77 - 82.

25. Gallant G. WebAssembly in Action: With Examples Using C++ and Emscripten. Manning Publications Co. LLC, 2019, 448p.

26. Булигін Володимир магістрант 1 курсу ІННІ ім. Ю. М. Потебні Наук. кер.: канд. фіз.-мат. наук, доц. Попівщий В.І. ОСОБЛИВОСТІ ВИКОРИСТАННЯ WEBASSEMBLY (WASM) ДЛЯ ПІДВИЩЕННЯ ПРОДУКТИВНОСТІ ТА БЕЗПЕКИ WEB-ЗАСТОСУНКІВ. ЗБІРНИК наукових праць студентів, аспірантів, докторантів і молодих вчених «МОЛОДА НАУКА-2024», Запоріжжя : Запорізький національний університет, 2024. С.172-173.

27. Булигін В. В., Попівщий В. І. Дослідження особливостей WebAssembly (WASM) та його впливу на розвиток Web-технологій Інженерні інновації та

розбудова національної економіки : матеріали І Міжнародної науково-практичної конференції Інженерного навчально-наукового інституту ім. Ю. М. Потебні (09–10 травня 2024 року, м. Запоріжжя) / наук. ред. Н. Г. Метеленко ; Інженерний навчально-науковий інститут ім. Ю. М. Потебні Запорізького національного університету. Одеса : Видавничий дім «Гельветика», 2024. С. 604-605.

Декларація

академічної доброчесності

здобувача ступеня вищої освіти ЗНУ

Я, Булигін Володимир Володимирович , студент 2 курсу, форми навчання денної, Інженерного навчально-наукового інституту, спеціальність 121 Інженерія програмного забезпечення, адреса електронної пошти ipz19bd-04@stu.zsea.edu.ua, — підтверджую, що написана мною кваліфікаційна робота на тему «**Особливості використання WebAssembly (Wasm) для підвищення продуктивності та безпеки Web-застосунків**» відповідає вимогам академічної доброчесності та не містить порушень, що визначені у ст.42 Закону України «Про освіту», зі змістом яких ознайомлений.

- заявляю, що надана мною для перевірки електронна версія роботи є ідентичною її друкованій версії;

згоден/згодна на перевірку моєї роботи на відповідність критеріям академічної доброчесності у будь-який спосіб, у тому числі за допомогою інтернет-системи, а також на архівування моєї роботи в базі даних цієї системи.

Дата 12.12.2024 Підпис_____ Булигін Володимир Володимирович
(студент)

Дата 12. 12. 2024 Підпис _____ Полякова Наталія Петрівна
(науковий керівник)