

Детектор Плагиата v. 2965 - Отчёт оригинальности: 17.06.2026 15:41:33

Проанализированный документ: Диплом_Хомюк.docx Лицензия: ВОЛОДИМИР МАТІЄВСЬКИЙ

? Тип поиска: Поиск переписанного ? Язык: Uk

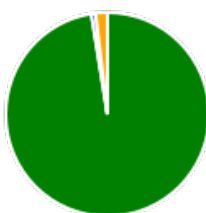
? Тип проверки: Интернет

ТЕЕ и кодировка: DocX n/a

Детальный анализ тела документа:

? Диаграмма соотношения частей:

Плагат 0% Оригинал 97.47%
Кавычки 0.67% ИИ 1.86%



? Подробное сходство:



? Граф распределения зон:



? Источники плагиата: 0

? Детали обработанных ресурсов: 176 - ОК / 10 - Ошибок

? Важные замечания:

Википедия:	Google Книги:	Сервисы платных работ:	Античит:
[не обнаружено]	[не обнаружено]	[не обнаружено]	Обнаружено сокрытие!

? Античит-отчет UACE:

1. Статус: Анализатор **Включен** Нормализатор **Включен** сходство символов установлено на **100%**
2. Обнаруженный процент загрязнения UniCode: **8%** с лимитом: 4%
3. Процент нераспознанных символов после нормализации: **4,6%**
4. Все подозрительные символы будут отмечены фиолетовым цветом: [Abcd...](#)
5. Найдены невидимые символы: 0

Рекомендации по оценке:

Особое внимание следует уделить анализу этого отчета! Предполагается, что этот документ содержит значительное количество символов, чуждых языку документа. Это прямое указание на то, что автор документа использовал специальное программное обеспечение\онлайн-веб-сервис, чтобы эффективно скрыть текст в попытке избежать обнаружения потенциального плагиата. Настоятельно рекомендуется передать это дело на более высокий уровень! В случае сомнений обращайтесь: в службу поддержки Детектора плагиата!

Алфавитная статистика и анализ символов:

 Активные ссылки (URL-адреса, извлеченные из документа):

URL не найдены

 Исключённые ресурсы:

URL не найдены

 Включённые ресурсы:

URL не найдены

Детальний аналіз документа:

Міністерство освіти і науки України Державний заклад

Цитування: 0,01%

id: 1

«Луганський НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ Тараса Шевченка»

Факультет (інститут) Факультет технологій та інформаційних систем (повна назва)
Кафедра Кафедра інформаційних технологій та систем (повна назва) Пояснювальна
записка до кваліфікаційної роботи за першим (бакалаврським) рівнем освіти Розробка веб-
застосунку для обміну повідомленнями Виконав: студент 4 курсу напряму підготовки
(спеціальності) 121

Цитування: 0%

id: 2

«Інженерія програмного забезпечення»_

(шифр і назва напряму підготовки, спеціальності) Хомюк І.В. (прізвище
та ініціали) Керівник __Смагіна О.О. (прізвище та ініціали) Рецензент __Козуб Ю.Г.
(прізвище та ініціали) Лубни - 2026 року ЗМІСТ ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ5 РОЗДІЛ 1.
ТЕОРЕТИЧНІ ОСНОВИ РОЗРОБКИ КЛІЄНТ-СЕРВЕРНИХ ВЕБ-ЗАСТОСУНКІВ ДЛЯ ОБМІНУ
ПОВІДОМЛЕННЯМИ В РЕАЛЬНОМУ ЧАСІ8 1.1. Клієнт-серверна архітектура8 1.2. Моделі
клієнт-серверних систем10 1.3. Протоколи взаємодії клієнта і сервера в реальному часі14
1.4. Особливості розробки веб-додатків реального часу на [JavaScript](#) та [Node.js](#)17 Висновки
до розділу 119 РОЗДІЛ 2. ПРОЕКТУВАННЯ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ ВЕБ-ЗАСТОСУНКУ

Цитування: 0%

id: 3

«CHATTY»

21 2.1. Аналіз вимог до веб-застосунку та архітектурне проектування21 2.1.1.
Функціональні вимоги до системи21 2.1.2. Нефункціональні вимоги до системи24 2.1.3.
Специфікація архітектури системи та взаємодія компонентів26 2.2. Проектування
структури бази даних та моделей даних28 2.2.1. Обґрунтування вибору [NoSQL](#) СУБД
[MongoDB](#)28 2.2.2. Розробка та програмна реалізація моделей даних ([Mongoose Models](#))29
2.2.3. Схема зв'язків між колекціями (Логічна структура БД)31 2.3. Розробка серверної
частини ([Backend](#)) на [Node.js](#) та [Express](#)33 2.3.1. Налаштування оточення та конфігурація
сервера33 2.3.2. Система безпеки, автентифікації та захисту маршрутів34 2.3.3. Реалізація
архітектури [REST API](#) та ендпоінтів36 2.3.4. Обробка файлових потоків та інтеграція
[Multer](#)36 2.3.5. Реалізація сервісу сповіщень на базі [Nodemailer](#)38 2.4. Реалізація
повнодуплексного обміну повідомленнями через [Socket.io](#)40 2.4.1. Ініціалізація [WebSocket](#)-
сервера та інтеграція з [Express](#)40 2.4.2. Архітектура подій ([Event-Driven Architecture](#))41
2.4.3. Менеджмент кімнат ([Rooms](#)) та ізоляція приватних чатів43 2.5. Розробка клієнтської
частини ([Frontend](#)) та інтерфейсу користувача45 2.5.1. Модульна структура веб-додатку на
стороні клієнта46 2.5.2. Стилзація, адаптивний дизайн та кастомізація інтерфейсу47 2.5.3.
Клієнтська [WebSocket](#)-логіка та динамічне оновлення [DOM](#)50 ВИСНОВКИ ДО РОЗДІЛУ 252
ВИСНОВОК55 СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ57 Додаток А60 Додаток Б98 Додаток В99
Додаток Г147 Додаток Д148 Додаток Е149 Додаток Є150 Додаток Ж151 Додаток К152
Додаток Р153 Додаток Л154 Додаток М157 Додаток Н161 Додаток З162 Додаток П164
Додаток Т171 Додаток С174 Додаток У176 Додаток Х178 Додаток Ц181 Додаток Ч184
Додаток Ю187 Додаток Я190 ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ [Atlas](#) - [MongoDB Atlas](#) (хмарна
платформа бази даних) [bcryptjs](#) - бібліотека для хешування паролів [CORS](#) - [Cross-Origin](#)
[Resource Sharing](#) (механізм крос-доменних запитів) [dotenv](#) - бібліотека для завантаження
змінних середовища [Express](#) - фреймворк веб-додатків для [Node.js](#) [HTML](#) - [HyperText Markup](#)
[Language](#) [HTTP](#) - [HyperText Transfer Protocol](#) [HTTPS](#) - [HyperText Transfer Protocol Secure](#) [JSON](#) -
[JavaScript Object Notation](#) [JWT](#) - [JSON Web Token](#) [MongoDB](#) - [MongoDB](#) (документоорієнтована
[NoSQL](#) база даних) [Mongoose](#) - [ODM](#)-бібліотека для роботи з [MongoDB](#) [Multer](#) - [middleware](#)
для обробки завантаження файлів [Nodemailer](#) - бібліотека для відправлення електронних
листів [WebSocket](#) - протокол повнодуплексного обміну даними в реальному часі
Актуальність роботи.

AI generated text detected: ChatGPT 5.3, : 72,37%

id: 4

Сучасні інформаційні технології кардинально змінили спосіб спілкування людей. Щодня
мільйони користувачів обмінюються текстовими повідомленнями в режимі реального
часу. Миттєвий обмін інформацією став невід'ємною частиною як особистого, так і
професійного життя. Однак більшість користувачів не замислюються, наскільки

складними є технології, що забезпечують реальний час у веб-додатках. Основним і найбільш ефективним підходом до реалізації таких систем залишається клієнт-серверна архітектура. На відміну від десктопних [Java](#)-додатків, сучасні веб-застосунки працюють безпосередньо в браузері, що накладає особливі вимоги до технологій обміну даними, масштабованості та безпеки.

Метою бакалаврської роботи є розробка веб-застосунку для обміну повідомленнями в реальному часі [Chatty](#). Об'єкт дослідження: клієнт-серверний веб-застосунок для обміну повідомленнями в реальному часі. Предмет дослідження: технології та методи створення веб-додатків реального часу з використанням [Node.js](#), [WebSocket](#) та сучасних веб-технологій. Відповідно до мети та предмета дослідження були поставлені такі основні завдання: узагальнити теоретичні основи клієнт-серверної архітектури веб-додатків; розглянути технології забезпечення обміну повідомленнями в реальному часі ([WebSocket](#), [Server-Sent Events](#), [Long Polling](#)); проаналізувати особливості розробки серверної частини на [Node.js](#) та клієнтської частини з використанням [JavaScript](#), [HTML](#), [CSS](#); дослідити питання автентифікації, безпеки та зберігання даних у чат-додатках; розробити та протестувати повнофункціональний веб-застосунок [Chatty](#). Для вирішення поставлених завдань використовувалися такі методи дослідження: теоретичні — аналіз наукової літератури та нормативно-технічної документації; емпіричні — порівняльний аналіз технологій реального часу; експериментальні — практична реалізація, тестування та відлагодження програмного продукту. До складу роботи входять три розділи, які висвітлюють теоретичні основи, практичну реалізацію та тестування веб-застосунку для обміну повідомленнями.

 AI generated text detected: ChatGPT 5.3, : 61,38%

id: 5

Практичне значення розробки полягає в створенні функціонального веб-додатку реального часу, який може використовуватися як для навчальних цілей, так і як базовий прототип для подальшої розробки корпоративних або особистих месенджерів. РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ РОЗРОБКИ КЛІЄНТ-СЕРВЕРНИХ ВЕБ-ЗАСТОСУНКІВ ДЛЯ ОБМІНУ ПОВІДОМЛЕННЯМИ В РЕАЛЬНОМУ ЧАСІ 1.1. Клієнт-серверна архітектура Сучасний розвиток інформаційних технологій невід'ємно пов'язаний із розподіленими обчисленнями, серед яких особливе місце посідає клієнт-серверна архітектура. Вона є однією з найбільш поширених і ефективних моделей організації взаємодії в комп'ютерних мережах.

Клієнт-серверна архітектура — це модель розподіленої обробки даних, у якій системні ресурси і функції розділені між двома основними типами учасників: клієнтами та серверами. Клієнт виступає ініціатором взаємодії, надсилаючи запити на виконання певних дій або отримання інформації. Сервер, у свою чергу, обробляє ці запити, виконує необхідні обчислення, працює з базами даних і повертає клієнту результат. Загальну концептуальну схему взаємодії в межах клієнт-серверної архітектури представлено на рисунку 1.1.

Рисунок 1.1 — Концептуальна схема взаємодії

 Цитування: 0%

id: 6

«Клієнт — Сервер»

Історично клієнт-серверна модель почала активно розвиватися в 1980-х роках із поширенням локальних мереж. Однак справжній прорив стався з розвитком глобальної мережі [Internet](#), коли більшість сучасних сервісів (веб-сайти, месенджери, хмарні сервіси, онлайн-ігри) перейшли саме на цю архітектуру. На відміну від централізованих систем, де всі обчислення виконувалися на одному потужному комп'ютері, клієнт-серверний підхід дозволив значно підвищити ефективність використання ресурсів і забезпечити одночасну роботу тисяч і мільйонів користувачів. Основними компонентами клієнт-серверної архітектури є: Клієнт — програмне забезпечення, що працює на пристрої кінцевого користувача (веб-браузер, мобільний додаток тощо). Його основні функції включають формування запитів, відображення інформації у зручному для користувача вигляді та обробку локальних подій. У веб-додатках роль клієнта виконує браузер, який інтерпретує [HTML](#), [CSS](#) та [JavaScript](#). Сервер — потужний комп'ютер (або група серверів), який постійно знаходиться в режимі очікування запитів. Він виконує бізнес-логіку, автентифікацію користувачів, роботу з базою даних, обробку повідомлень і забезпечує безпеку системи. Сервер може обслуговувати одночасно велику кількість клієнтів. Мережа — канал зв'язку між клієнтом і сервером. У сучасних системах найчастіше використовуються протоколи [TCP/IP](#) та [HTTP/HTTPS](#). Якість і швидкість мережі безпосередньо впливають на продуктивність усього додатку. Важливою перевагою клієнт-серверної архітектури є

централізоване зберігання даних. Вся інформація (профілі користувачів, історія повідомлень, налаштування) зберігається на сервері, що забезпечує доступ до даних з будь-якого пристрою після авторизації. Крім того, така модель значно спрощує оновлення програмного забезпечення: достатньо внести зміни на сервері, і всі клієнти автоматично отримують оновлену версію. У контексті веб-застосунків для обміну повідомленнями клієнт-серверна архітектура набуває особливого значення. Чат-додатки потребують швидкої обробки великої кількості одночасних з'єднань, надійної доставки повідомлень у реальному часі та захисту персональних даних. Саме тому більшість популярних месенджерів (наприклад, [WhatsApp Web](#), [Telegram Web](#), [Slack](#)) побудовані за клієнт-серверною моделлю. Разом з тим, клієнт-серверна архітектура має певні недоліки. Основним з них є залежність від стабільності роботи сервера: у разі його відмови весь сервіс стає недоступним. Також при великій кількості користувачів виникає необхідність у масштабуванні системи (використання балансувальників навантаження, кількох серверів, хмарних рішень тощо). У розробленому веб-застосунку [Chatty](#) реалізовано трирівневу клієнт-серверну архітектуру, де: клієнтський рівень представлений веб-інтерфейсом ([HTML](#), [CSS](#), [JavaScript](#)); серверний рівень — [Node.js](#) з використанням [Express](#) та [WebSocket](#); рівень даних — хмарна база даних [MongoDB Atlas](#). Така організація дозволяє ефективно розподіляти навантаження, забезпечувати високу швидкість обміну повідомленнями та підтримувати надійність системи.

1.2. Моделі клієнт-серверних систем Існує кілька основних моделей організації клієнт-серверної взаємодії, які відрізняються способом розподілу логіки, даних і презентаційного шару. Вибір тієї чи іншої моделі суттєво впливає на масштабованість, безпеку, продуктивність і складність підтримки додатку. [Peer-to-Peer \(P2P\)](#) архітектура (однорангова) — це децентралізована модель, у якій кожен учасник мережі може одночасно виступати як клієнтом, так і сервером. У такій системі відсутній центральний сервер, а вузли мережі обмінюються даними безпосередньо між собою. Перевагами [P2P](#)-архітектури є висока відмовостійкість (відключення одного вузла не впливає на роботу мережі) та відсутність єдиної точки відмови. Однак у контексті веб-додатків для обміну повідомленнями дана модель має суттєві недоліки: складність забезпечення безпеки, проблеми з [NAT](#)-транзитом, складне управління користувачами та історією повідомлень. Тому [P2P](#)-архітектура рідко використовується в сучасних чат-додатках, за винятком окремих спеціалізованих рішень (наприклад, деяких децентралізованих месенджерів).

Дворівнева архітектура ([Two-tier Client-Server](#)) є найпростішою і найпоширенішою формою клієнт-серверної моделі. У ній вся логіка додатку та робота з базою даних зосереджені на сервері, а клієнт відповідає лише за презентаційний шар (інтерфейс) та формування запитів. Така модель добре підходить для невеликих додатків з обмеженою кількістю користувачів. Головними перевагами є простота реалізації та швидкість розробки. Водночас дворівнева архітектура має серйозні обмеження при зростанні навантаження: сервер одночасно виконує і бізнес-логіку, і роботу з даними, що призводить до швидкого зростання навантаження на нього. Крім того, при зміні бізнес-логіки часто потрібно оновлювати як серверну, так і клієнтську частини.

Трирівнева ([Three-tier](#)) та багаторівнева ([N-tier](#)) архітектура на сьогодні є найбільш сучасною та рекомендованою моделлю для розробки веб-застосунків, зокрема чатів. Вона передбачає чітке розділення системи на три основні шари: Клієнтський шар ([Presentation tier](#)) — веб-браузер, який відповідає за інтерфейс користувача, відображення повідомлень, обробку подій та візуалізацію. Серверний шар ([Application / Business Logic tier](#)) — середній рівень, на якому виконується вся бізнес-логіка: автентифікація, обробка повідомлень, робота з [WebSocket](#), валідація даних тощо. У розробленому додатку цей шар реалізований на [Node.js](#) з використанням [Express](#). Шар даних ([Data tier](#)) — база даних, у даному випадку хмарне рішення [MongoDB Atlas](#). Структурну схему трирівневої клієнт-серверної архітектури розробленого веб-застосунку

Цитування: 0%

id: 7

«Chatty»

наведено на рисунку 1.1. Рисунок 1.1 — Трирівнева архітектура веб-застосунку

Цитування: 0%

id: 8

«Chatty»

Таке розділення дозволяє незалежно масштабувати кожен шар, спрощує підтримку та розвиток системи. Наприклад, при збільшенні кількості користувачів можна додавати нові серверні інстанси без зміни бази даних. Крім того, трирівнева архітектура значно підвищує

рівень безпеки, оскільки клієнт не має прямого доступу до бази даних. У сучасних веб-додатках часто використовують ще більш гнучку багаторівневу ([N-tier](#)) архітектуру, де між основними шарами можуть додаватися додаткові (наприклад, шар кешування [Redis](#), шар мікросервісів, шар автентифікації тощо). Порівняльний аналіз моделей показує, що трирівнева архітектура найкраще відповідає вимогам сучасних веб-застосунків для обміну повідомленнями в реальному часі, оскільки забезпечує баланс між продуктивністю, масштабованістю, безпекою та зручністю підтримки. У бакалаврській роботі для веб-застосунку [Chatty](#) обрано трирівневу клієнт-серверну архітектуру. Таке рішення дозволяє ефективно реалізовувати обмін повідомленнями через [WebSocket](#), забезпечувати надійне зберігання даних у [MongoDB Atlas](#) та створювати сучасний, зручний інтерфейс користувача.

1.3. Протоколи взаємодії клієнта і сервера в реальному часі

Для забезпечення обміну повідомленнями в реальному часі стандартного [HTTP](#)-протоколу, який працює за принципом

Цитування: 0%

id: 9

«запит-відповідь»,

часто недостатньо. Класичний [HTTP](#) є одностороннім і [stateless](#)-протоколом, що змушує клієнта постійно ініціювати нові запити для отримання оновлень. Це призводить до високого мережного навантаження та затримок у доставці повідомлень. Для вирішення проблеми реального часу було розроблено декілька технологій і підходів, які відрізняються за ефективністю, складністю реалізації та використанням ресурсів. [Short Polling](#) — найпростіший метод, при якому клієнт через певні проміжки часу (наприклад, кожні 2–5 секунд) надсилає запит на сервер з метою перевірки нових повідомлень. Незважаючи на простоту реалізації, цей підхід має низьку ефективність, оскільки більшість запитів повертає порожню відповідь, створюючи значне навантаження як на сервер, так і на мережу. [Long Polling](#) — удосконалена версія поліngu. Клієнт надсилає запит, а сервер тримає з'єднання відкритим доти, доки не з'явиться нове повідомлення або не мине таймаут. Після отримання відповіді клієнт одразу надсилає новий запит. Такий підхід зменшує кількість

Цитування: 0%

id: 10

«пустих»

відповідей, але все одно створює велику кількість одночасних з'єднань на сервері та не дозволяє повноцінний двосторонній обмін даними. [Server-Sent Events \(SSE\)](#) — технологія, яка дозволяє серверу надсилати дані клієнту в односторонньому порядку через постійне [HTTP](#)-з'єднання. [SSE](#) добре підходить для ситуацій, коли не потрібна двостороння комунікація (наприклад, для стрічок новин або сповіщень). Однак для повноцінного чату, де користувачі активно надсилають і отримують повідомлення, [SSE](#) має обмеження. [WebSocket](#) — сучасний і найбільш ефективний протокол для обміну даними в реальному часі. [WebSocket](#) встановлює постійне повнодуплексне (двостороннє) з'єднання між клієнтом і сервером поверх [TCP](#)-протоколу. Після початкового рукоштовування ([handshake](#)), яке виконується через звичайний [HTTP](#)-запит, з'єднання залишається відкритим, а дані передаються з мінімальними накладними витратами у обох напрямках у режимі реального часу. Основні переваги [WebSocket](#) порівняно з іншими технологіями: Низька затримка доставки повідомлень; Значно менші витрати на мережевий трафік; Підтримка двостороннього зв'язку; Висока масштабованість при великій кількості одночасних користувачів; Вбудована підтримка в усіх сучасних веб-браузерах. У порівнянні з традиційним [HTTP](#), [WebSocket](#) економить серверні ресурси та дозволяє обробляти тисячі активних з'єднань на одному сервері. Алгоритм та послідовність взаємодії між клієнтською та серверною частинами за цим протоколом наочно зображено на рисунку 1.2. Рисунок 1.2 — Схема взаємодії клієнта та сервера за протоколом [WebSocket](#)

Клієнт виступає ініціатором У розробленому веб-застосунку [Chatty](#) основним протоколом взаємодії обрано [WebSocket](#). Для його реалізації на серверній стороні використовується бібліотека [Socket.io](#) поверх [Express.js](#), яка забезпечує надійне з'єднання, автоматичне відновлення при розривах, підтримку [fallback](#)-механізмів та зручну роботу з подіями ([emit](#) / [on](#)). Клієнтська частина взаємодіє з сервером через [WebSocket API](#) браузера. Додатково в роботі використовується протокол [HTTPS](#) для забезпечення безпеки початкового рукоштовування та передачі конфіденційної інформації (авторизація, реєстрація). Для роботи з базою даних застосовується драйвер [MongoDB](#) та [ODM](#)-бібліотека [Mongoose](#). Вибір [WebSocket](#) як основного протоколу дозволяє повністю задовольнити вимоги технічного завдання щодо

отримання та надсилання повідомлень у режимі реального часу, забезпечуючи високу швидкість та комфортну взаємодію користувачів.

1.4. Особливості розробки веб-додатків реального часу на [JavaScript](#) та [Node.js](#)

[JavaScript](#), який традиційно використовувався лише для клієнтської частини веб-додатків, завдяки середовищу виконання [Node.js](#) став повноцінною серверною технологією. Це дозволило розробникам використовувати одну мову програмування на всьому стеку, що значно спрощує розробку, підтримку та масштабованість сучасних веб-застосунків. [Node.js](#) — це асинхронне, подієво-орієнтоване середовище виконання [JavaScript](#), побудоване на рушії [V8](#). Його основною перевагою є неблокуюча модель введення-виведення ([non-blocking I/O](#)), яка ідеально підходить для додатків з високим рівнем паралелізму, таких як чати, де одночасно може бути велика кількість активних з'єднань. На відміну від традиційних серверних мов ([PHP](#), [Python](#)), [Node.js](#) ефективно обробляє тисячі одночасних [WebSocket](#)-з'єднань на одному потоці. Для побудови серверної частини в роботі використовується фреймворк [Express](#), який забезпечує зручну маршрутизацію, [middleware](#)-архітектуру та швидку обробку [HTTP](#)-запитів. Це дозволяє чітко розділити [RESTful](#) ендпоінти (реєстрація, авторизація, завантаження файлів) від [WebSocket](#)-логіки. Однією з ключових технологій забезпечення реального часу є бібліотека [Socket.io](#). Вона працює поверх [WebSocket](#) і надає надійний двосторонній канал зв'язку, автоматичне відновлення з'єднання при розривах, підтримку [fallback](#)-механізмів ([Long Polling](#)) та зручний [API](#) для роботи з подіями ([emit](#), [on](#), [broadcast](#)). Для роботи з даними обрано [MongoDB Atlas](#) — хмарну документоорієнтовану [NoSQL](#) базу даних. Завдяки бібліотеці [Mongoose](#) забезпечується об'єктно-документне відображення ([ODM](#)), валідація схем, [middleware](#) та зручна взаємодія з колекціями користувачів і повідомлень. Такий вибір дозволяє гнучко зберігати історію чатів, підтримувати різні типи повідомлень та швидко масштабувати систему. Важливими аспектами безпеки є використання: [JWT](#) ([JSON Web Token](#)) — для [stateless](#)-автентифікації користувачів; [bcryptjs](#) — для надійного хешування паролів; [CORS](#) — для керування крос-доменними запитами. Додатково в проєкті застосовуються бібліотеки [dotenv](#) (управління змінними середовища), [Multer](#) (обробка завантаження файлів) та [Nodemailer](#) (відправка електронних листів, наприклад, для підтвердження реєстрації). Переваги єдиного [JavaScript](#)-стеку ([Fullstack JavaScript](#)): Єдина мова програмування на клієнті та сервері; Швидка розробка та легке перенесення коду; Велика екосистема готових пакетів ([npm](#)); Висока продуктивність при обробці реального часу; Простота найму розробників та підтримки проєкту. Серед особливостей та викликів розробки на [Node.js](#) варто відзначити необхідність правильної організації асинхронного коду, обробки помилок, масштабування (кластеризація, використання [PM2](#) або [Docker](#)) та забезпечення безпеки (захист від [XSS](#), [rate limiting](#), валідація даних). У розробленому веб-застосунку [Chatty](#) поєднання [Node.js](#), [Express](#), [Socket.io](#) та [MongoDB Atlas](#) дозволило створити сучасний, швидкий і масштабований чат-додаток, який повністю відповідає вимогам технічного завдання щодо функціонування в реальному часі.

Висновки до розділу 1

У першому розділі бакалаврської роботи були розглянуті теоретичні основи розробки клієнт-серверних веб-застосунків для обміну повідомленнями в реальному часі. Проведений аналіз показав, що клієнт-серверна архітектура є фундаментальною моделлю сучасних мережевих додатків і найбільш ефективним підходом для реалізації чат-систем. Були детально вивчені основні моделі клієнт-серверних систем: [Peer-to-Peer](#), дворівнева та трирівнева (багаторівнева) архітектури. Було обґрунтовано, що для сучасного веб-додатку реального часу найбільш доцільною є трирівнева архітектура, яка забезпечує чітке розділення відповідальності між клієнтським шаром, серверним шаром бізнес-логіки та шаром даних, сприяє кращій масштабованості, безпеці та зручності підтримки системи. Особлива увага приділялася протоколам забезпечення обміну даними в реальному часі. Проведено порівняльний аналіз [Short Polling](#), [Long Polling](#), [Server-Sent Events](#) та [WebSocket](#). Зроблено висновок, що протокол [WebSocket](#) є оптимальним рішенням для чат-додатків завдяки повнодуплексному зв'язку, мінімальним затримкам і високій ефективності використання мережевих ресурсів. Також були розглянуті особливості розробки веб-додатків реального часу на [JavaScript](#) та [Node.js](#). Встановлено, що використання єдиного [JavaScript](#)-стеку дозволяє значно прискорити розробку, спростити підтримку коду та підвищити продуктивність системи. Проаналізовано ключові технології та бібліотеки, що використовуються в роботі: [Express.js](#), [Socket.io](#), [MongoDB Atlas](#), [Mongoose](#), [JWT](#), [bcryptjs](#) та інші. Показано, що обраний технологічний стек повністю відповідає сучасним вимогам до веб-додатків такого типу. Таким чином, теоретичний аналіз, проведений у першому розділі, дозволив обґрунтувати вибір архітектури, протоколів та інструментів розробки для створення веб-застосунку

Chatty. Вибрані рішення забезпечують високу швидкість обміну повідомленнями, надійність роботи, безпеку даних та можливість подальшого масштабування системи. Результати теоретичного дослідження стали основою для практичної реалізації веб-додатку в наступних розділах роботи. РОЗДІЛ 2. ПРОЕКТУВАННЯ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ ВЕБ-ЗАСТОСУНКУ

Цитування: 0%

id: 11

«CHATTY»

2.1. Аналіз вимог до веб-застосунку та архітектурне проектування Етап аналізу вимог та проектування архітектури є фундаментальним кроком у життєвому циклі розробки програмного забезпечення. Особливо це стосується систем обміну повідомленнями у реальному часі (**Real-Time Messengers**), де правильний розподіл навантаження та чітке визначення меж функціональності безпосередньо впливають на стабільність, безпеку та швидкість доставки інформації кінцевому користувачу. У межах розробки веб-застосунку

Цитування: 0%

id: 12

«Chatty»

було проведено детальний збір, аналіз та класифікацію технічних вимог до системи. Усі вимоги було розподілено на дві основні групи: функціональні (які описують безпосередні можливості користувача та логіку поведінки системи) та нефункціональні (що визначають якісні характеристики, обмеження, продуктивність та системні властивості продукту).

2.1.1. Функціональні вимоги до системи Функціональні вимоги визначають сервіси та функції, які програмний продукт має надавати користувачам для вирішення їхніх повсякденних завдань. Для веб-застосунку

Цитування: 0%

id: 13

«Chatty»

було сформовано та реалізовано такий перелік ключових функціональних вимог: Управління обліковими записами та профілями користувачів: Реєстрація користувачів: створення нового облікового запису в системі з обов'язковою валідацією полів (унікальний email, ім'я користувача, надійний пароль) та інтеграцією сервісу верифікації через надсилання системних повідомлень. Автентифікація та авторизація: забезпечення безпечного входу до системи за допомогою сучасних криптографічних протоколів, збереження сесії користувача та розмежування прав доступу до приватних маршрутів (маршрутизація API). Редагування профілю: можливість завантаження та зміни персонального аватара користувача, оновлення текстових даних та налаштування мовної локалізації. Обмін повідомленнями в режимі реального часу (**Real-Time Messaging**): Миттєве надсилання та отримання повідомлень: користувач повинен мати можливість відправляти текстові повідомлення у приватні чати з мінімальною затримкою, при цьому отримувач має бачити повідомлення миттєво без потреби ручного оновлення веб-сторінки. Індикація активності (**Typing Indicator**): відображення статусу введення тексту іншим користувачем у поточному діалозі для підвищення інтерактивності інтерфейсу. Історія повідомлень: автоматичне збереження текстових діалогів та медіафайлів у базі даних із можливістю їх підвантаження під час відкриття конкретного чату. Робота з мультимедійними даними та файловими потоками: Завантаження файлів: підтримка надсилання зображень та інших файлових вкладень безпосередньо у вікно чату. Попередній перегляд (**File Preview**): генерація та візуалізація прев'ю-зображень для завантажених медіафайлів безпосередньо перед надсиланням або всередині стрічки повідомлень для покращення користувацького досвіду (**UX**). Кастомізація інтерфейсу та локалізація: Керування темами оформлення: підтримка динамічного перемикання між світлою (**Light Mode**) та темною (**Dark Mode**) темами інтерфейсу із збереженням вибору користувача у локальному сховищі браузера. Багатомовність (**Localization**): підтримка інтерфейсу як українською (**uk**), англійською (**en**), німецькою (**de**), так і іспанською (**es**) мовами для гнучкого використання залежно від уподобань користувача. Обмін повідомленнями в режимі реального часу (**Real-Time Messaging**): Миттєве надсилання та отримання повідомлень: користувач повинен мати можливість відправляти текстові повідомлення у приватні чати з мінімальною затримкою, при цьому отримувач має бачити повідомлення миттєво без потреби ручного оновлення веб-сторінки. Індикація активності (**Typing Indicator**): відображення статусу введення тексту іншим користувачем у поточному діалозі для підвищення інтерактивності інтерфейсу. Історія повідомлень: автоматичне збереження текстових діалогів та медіафайлів у базі

даних із можливістю їх підвантаження під час відкриття конкретного чату. Робота з мультимедійними даними та файловими потоками: Завантаження файлів: підтримка надсилання зображень та інших файлових вкладень безпосередньо у вікно чату. Попередній перегляд ([File Preview](#)): генерація та візуалізація прев'ю-зображень для завантажених медіафайлів безпосередньо перед надсиланням або всередині стрічки повідомлень для покращення користувацького досвіду ([UX](#)). Кастомізація інтерфейсу та локалізація: Керування темами оформлення: підтримка динамічного перемикання між світлою ([Light Mode](#)) та темною ([Dark Mode](#)) темами інтерфейсу із збереженням вибору користувача у локальному сховищі браузера. Багатомовність ([Localization](#)): підтримка інтерфейсу як українською ([uk](#)), так і англійською ([en](#)) мовами для гнучкого використання залежно від уподобань користувача. Концептуальну модель взаємодії користувача з функціональними можливостями месенджера у вигляді [UML](#)-діаграми прецедентів представлено на рисунку 2.1 Рисунок 2.1 — [UML](#)-діаграма прецедентів використання веб-застосунку

Цитування: 0%

id: 14

«Chatty»

2.1.2. Нефункціональні вимоги до системи Нефункціональні вимоги визначають критерії якості, атрибути системи та технічні обмеження, яким має відповідати розроблений веб-застосунок для забезпечення стабільного функціонування під навантаженням. До веб-застосунку

Цитування: 0%

id: 15

«Chatty»

висуното такі нефункціональні вимоги: Продуктивність та швидкість роботи ([Performance](#)): Максимальна затримка ([Latency](#)) при доставці текстового повідомлення між активними клієнтами в мережі через протокол [WebSocket](#) не повинна перевищувати 200–300 мс при стабільному інтернет-з'єднанні. Час відгуку серверної частини на стандартні [HTTP](#)-запити ([REST API](#)) для авторизації чи реєстрації не повинен перевищувати 500 мс. Безпека та захист даних ([Security](#)): Шифрування паролів: забороняється збереження паролів користувачів у базі даних у відкритому текстовому вигляді. Усі паролі мають проходити обов'язкове одностороннє хешування за допомогою алгоритму [bcrypt](#). Безпека сесій: захист клієнт-серверної взаємодії за допомогою механізму [Stateless](#)-автентифікації на базі [JSON Web Tokens \(JWT\)](#). Контроль доступу: налаштування політики [Cross-Origin Resource Sharing \(CORS\)](#) для обмеження несанкціонованих запитів до серверної частини з боку сторонніх доменів. Захист каналів зв'язку: використання захищених протоколів [HTTPS](#) та [WSS \(WebSocket Secure\)](#) для передачі конфіденційних даних та захисту від атак типу

Цитування: 0%

id: 16

«Man-in-the-Middle»

([MitM](#)). Сумісність та адаптивність інтерфейсу ([Usability & Responsiveness](#)): Клієнтська частина додатка повинна мати 100% адаптивний дизайн ([Responsive Web Design](#)), що забезпечує коректне відображення всіх елементів керування, списку чатів та вікна повідомлень як на десктопних моніторах, так і на мобільних пристроях (смартфонах, планшетах) з різною роздільною здатністю екрана. Кроссбраузерність: стабільна робота інтерфейсу в усіх сучасних популярних браузерах ([Google Chrome](#), [Mozilla Firefox](#), [Apple Safari](#), [Microsoft Edge](#)). 2.1.3. Специфікація архітектури системи та взаємодія компонентів На основі детального аналізу вимог було розроблено специфікацію архітектури програмного продукту. Для забезпечення високого рівня масштабованості, гнучкості та безпеки було обрано трирівневу архітектурну модель ([Three-tier Architecture](#)). Дана модель передбачає повне розділення системи на три незалежні рівні (шари), кожен з яких виконує строго визначений набір функцій і взаємодіє з іншими рівнями через стандартизовані інтерфейси розробки (рис. 2.1). 1. Рівень представлення ([Presentation Tier / Frontend](#)): Цей рівень функціонує безпосередньо на пристрої кінцевого користувача всередині веб-браузера. Він побудований на базі стандартного технологічного стеку: [HTML5](#) для структурування контенту (Додаток Я), [CSS3](#) для стилізації, адаптивної верстки та забезпечення темної/світлої тем (Додаток Б), а також [JavaScript](#) для реалізації динамічної логіки інтерфейсу (Додаток В). Головне завдання клієнтського рівня — візуалізація даних, отриманих від сервера, обробка дій користувача (кліків, введення тексту) та підтримання постійного зв'язку через [WebSocket](#)-клієнт для миттєвого відображення нових повідомлень

без перезавантаження сторінки. 2. Рівень бізнес-логіки ([Application / Business Logic Tier / Backend](#)): Центральний компонент системи, реалізований як веб-сервер на базі середовища виконання [Node.js](#) з використанням мікрофреймворку [Express](#). Цей шар відповідає за обробку всієї логіки додатка: валідацію вхідних даних, генерацію та перевірку токенів автентифікації ([JWT](#)), хешування паролів, маршрутизацію [HTTP](#)-запитів та оркестрацію файлових потоків за допомогою [middleware](#)-модулів ([Multer](#)). Окремим критично важливим ядром цього шару є [WebSocket](#)-сервер на базі бібліотеки [Socket.io](#), який управляє пулом активних мережових з'єднань, обробляє події надсилання повідомлень та здійснює їх ретрансляцію відповідним клієнтам у реальному часі. 3. Рівень даних ([Data Tier / Database](#)): Нижній шар архітектури, представлений хмарною [NoSQL](#) базою даних [MongoDB Atlas](#). Даний рівень відповідає за надійне, довготривале та структуроване зберігання інформації. Взаємодія між сервером бізнес-логіки ([Node.js](#)) та рівнем даних реалізована за допомогою [ODM](#)-бібліотеки ([Object-Document Mapping](#)) [Mongoose](#). [Mongoose](#) дозволяє описати чіткі схеми документів, виконувати швидку типізацію та валідацію даних перед їх записом у колекції, а також оптимізувати пошукові запити для швидкого вилучення історії листування. Повна програмна конфігурація та вихідний код головного файлу ініціалізації сервера міститься у (Додатку Т). Взаємодія компонентів архітектури відбувається за строгою схемою: клієнт ніколи не взаємодіє з базою даних безпосередньо, що виключає можливість прямих ін'єкцій чи компрометації даних. Усі запити проходять через захищений проміжний серверний шар, де виконується автентифікація, перевірка прав доступу та логування подій. Такий інженерний підхід дозволяє незалежно модифікувати та масштабувати кожен рівень системи (наприклад, перенести базу даних на інший кластер або додати нові серверні потужності) без необхідності переписування коду клієнтської частини месенджера

Цитування: 0%

id: 17

«Chatty».

Налаштування підключення, конфігурацію пулу з'єднань та ініціалізацію взаємодії із СКБД на рівні серверної платформи наведено у (Додатку Г). Рисунок 2.2 — Структурна схема трирівневої архітектури веб-застосунку

Цитування: 0%

id: 18

«Chatty»

2.2. Проектування структури бази даних та моделей даних Сучасні системи обміну повідомленнями оперують величезними обсягами неструктурованих або напівструктурованих даних, які постійно оновлюються в реальному часі. Успішне функціонування чат-додатка залежить від швидкості запису нових повідомлень та ефективності вибірки історії листування. Тому етап проектування шару даних вимагає ретельного підбору архітектури системи управління базами даних (СУБД) та детального моделювання сутностей. 2.2.1. Обґрунтування вибору [NoSQL](#) СУБД [MongoDB](#) Під час проектування веб-застосунку

Цитування: 0%

id: 19

«Chatty»

буловедено порівняльний аналіз між класичними реляційними СУБД (наприклад, [PostgreSQL](#), [MySQL](#)) та нереляційними системами класу [NoSQL](#). Для реалізації ядра зберігання даних було обрано документоорієнтовану [NoSQL](#) базу даних [MongoDB](#) (у вигляді хмарного рішення [MongoDB Atlas](#)), керуючись такими інженерними критеріями: Гнучкість схем даних ([Schemaless](#)): На відміну від реляційних баз, де кожна зміна структури таблиці потребує виконання складних міграцій ([ALTER TABLE](#)) та може призвести до простою системи, [MongoDB](#) зберігає дані у гнучкому форматі [BSON](#) (бінарний [JSON](#)). Це дозволяє легко модифікувати структуру документів на етапі розширення функціоналу. Наприклад, якщо в майбутньому до повідомлення знадобиться додати нові поля (статус прочитання, реакції-емодзі тощо), це можна зробити миттєво без перебудови всієї бази даних. Висока швидкість операцій запису ([Write Performance](#)): У чат-додатках операції запису відбуваються безперервно — щоразу, коли будь-який користувач надсилає повідомлення, система фіксує його в базі. [MongoDB](#) оптимізована для швидкого вставлення ([Insert](#)) документів за рахунок архітектури пам'яті та відсутності необхідності перевірки складних каскадних зв'язків ([Foreign Keys](#)) та транзакцій [ACID](#) на рівні таблиць, що забезпечує мінімальну затримку ([latency](#)) під навантаженням. Ефективне збереження вкладених

структур та масивів: Повідомлення в месенджерах часто містять динамічні масиви даних (наприклад, список посилань на завантажені медіафайли чи зображення-прев'ю). У реляційних БД для цього довелося б створювати окрему проміжну таблицю та використовувати операції [JOIN](#), які уповільнюють роботу системи. В [MongoDB](#) такі дані зберігаються як звичайний масив всередині одного документа, що дозволяє отримати всю інформацію про повідомлення за один швидкий запит. Хмарна екосистема [MongoDB Atlas](#): Використання керованого хмарного рішення дозволяє делегувати завдання адміністрування, резервного копіювання ([backups](#)) та горизонтального масштабування ([sharding](#)) хмарній інфраструктурі, гарантуючи доступність бази даних 24/7. 2.2.2. Розробка та програмна реалізація моделей даних ([Mongoose Models](#)) Для інтеграції СУБД [MongoDB](#) із серверною частиною на платформі [Node.js](#) було використано [ODM](#)-бібліотеку ([Object-Document Mapping](#)) [Mongoose](#). Вона дозволяє накладати строгу валідацію, типізацію та структурування на документи [MongoDB](#) безпосередньо на рівні прикладного коду сервера. В межах архітектури веб-застосунку

Цитування: 0%

id: 20

«Chatty»

було спроектовано три ключові моделі даних: [User](#) (Користувачі), [Message](#) (Повідомлення) та [Chat](#) (Чати / Кімнати). Опис та лістинг моделі користувача ([User](#)) Модель [User](#) описує логічну структуру та параметри облікового запису користувача в системі. Вона містить обов'язкові індивідуальні поля для автентифікації та ідентифікації в мережі ([nickname](#), [email](#), [tag](#)), персональну інформацію профілю ([displayName](#), [phone](#), [bio](#), [avatar](#)), а також рекурсивний масив [contacts](#) для організації списку зв'язків між користувачами в межах однієї колекції. Програмну реалізацію структури схеми та експорту моделі даних користувача, включаючи проміжний метод автоматичного асинхронного хешування паролів, наведено у Додатку К. Опис та лістинг моделі повідомлення ([Message](#)) Модель [Message](#) відповідає за довготривале збереження історії листування. Зв'язок між документом повідомлення та користувачем-відправником реалізовано за допомогою механізму референсів ([mongoose.Schema.Types.ObjectId](#)), що вказують на конкретні ідентифікатори в колекції користувачів. Структура також підтримує вкладений об'єкт [file](#) для збереження метаданих та [URL](#)-адреси завантажених медіафайлів. Структура також підтримує вкладений об'єкт [file](#) для збереження метаданих та [URL](#)-адреси завантажених медіафайлів. Програмну реалізацію структури документа повідомлення з автоматичною фіксацією міток часу ([timestamps](#)) наведено у Додатку 3. Опис та лістинг моделі чату ([Chat](#)) Модель [Chat](#) є центральною сполучною ланкою в архітектурі бази даних месенджера

Цитування: 0%

id: 21

«Chatty».

Вона призначена для створення, ідентифікації та адміністрування просторів для спілкування. Завдяки гнучкій структурі [NoSQL](#), ця модель універсально обслуговує як приватні діалоги між двома користувачами ([Direct Messages](#)), так і групові чати ([Group Chats](#)) без необхідності створення додаткових таблиць чи колекцій. Окремою важливою архітектурною особливістю моделі є збереження вкладеного об'єкта [lastMessage](#). Замість того, щоб під час кожного рендерингу списку чатів виконувати важкий пошук останнього повідомлення в усій колекції [messages](#), копія останнього повідомлення дублюється безпосередньо в документ чату. Це забезпечує денормалізацію даних ([Data Denormalization](#)) та оптимізує навантаження на хмарну СУБД. Повний лістинг програмної реалізації моделі чату наведено у Додатку Ж. 2.2.3. Схема зв'язків між колекціями (Логічна структура БД) Незважаючи на те, що [MongoDB](#) належить до класу нереляційних баз даних, між її колекціями існують стійкі логічні зв'язки. У розробленому веб-застосунку застосовано зв'язки типу

Цитування: 0%

id: 22

«один-до-багатьох»

([One-to-Many](#)). Один користувач ([User](#)) може виступати автором багатьох повідомлень ([Message](#)) та бути учасником багатьох чатів ([Chat](#)), у той час як один чат містить у собі множину повідомлень, пов'язаних через ідентифікатор [chatId](#). Також реалізовано рекурсивний зв'язок на рівні колекції користувачів для побудови списків контактів. Концептуальну графічну схему зв'язків між сутностями (ЕР-діаграму) та логічну структуру розробленої бази даних в [MongoDB Atlas](#) представлено на рисунку 2.2. Рисунок 2.3 —

Логічна схема зв'язків та структура моделей даних у СКБД [MongoDB](#) Організація зв'язків через ідентифікатори [ObjectId](#) дозволяє серверній частині додатка ефективно використовувати метод динамічного об'єднання документів [.populate\(\)](#) на рівні [ODM Mongoose](#), що забезпечує високу швидкість агрегації даних під час формування відповідей для клієнтської частини. 2.3. Розробка серверної частини ([Backend](#)) на [Node.js](#) та [Express](#) Серверна частина ([Backend](#)) є ядром координації та забезпечення бізнес-логіки веб-застосунку

Цитирования: 0%

id: 23

«Chatty».

Вона виступає надійним посередником між клієнтським інтерфейсом та хмарною базою даних, забезпечуючи безпеку, валідацію, маршрутизацію запитів та керування сесіями користувачів. Для реалізації серверної сторони було обрано програмну платформу [Node.js](#) у поєднанні з гнучким веб-фреймворком [Express](#), що дозволило створити високопродуктивне, асинхронне та легко масштабоване серверне рішення. 2.3.1. Налаштування оточення та конфігурація сервера Першим етапом розробки серверної частини є створення безпечної конфігурації прикладного середовища. Веб-застосунок оперує конфіденційними даними, такими як рядки підключення до бази даних [MongoDB Atlas](#), секретні ключі для підпису токенів та паролі від поштових шлюзів. Зберігання таких даних безпосередньо в коді програми є грубим порушенням правил безпеки. Для вирішення цієї проблеми в проєкті було інтегровано та налаштовано бібліотеку [dotenv](#). Вона дозволяє винести всі чутливі налаштування в окремий ізольований файл конфігурації середовища [.env](#), який не потрапляє у відкритий репозиторій. Під час запуску сервера бібліотека автоматично зчитує ці змінні та інjektує їх у глобальний об'єкт [Node.js](#) — [process.env](#). Окремим критично важливим налаштуванням конфігурації є забезпечення механізму [CORS \(Cross-Origin Resource Sharing\)](#). Оскільки клієнтська частина чату ([Frontend](#)) та серверна бізнес-логіка ([Backend](#)) під час розробки та розгортання можуть функціонувати на різних доменах, піддоменах або портах, браузері за замовчуванням блокують такі крос-доменні [HTTP](#)-запити з міркувань безпеки. Для надання санкціонованого доступу клієнту до розробленого [REST API](#) на сервері було впроваджено проміжне програмне забезпечення ([middleware](#)) [cors](#), де чітко специфіковано дозволені білі списки ([Allow-Origins](#)), [HTTP](#)-методи ([GET](#), [POST](#), [PUT](#), [DELETE](#)) та дозволені заголовки ([Headers](#)). Окремим критично важливим налаштуванням конфігурації є забезпечення механізму [CORS \(Cross-Origin Resource Sharing\)](#). Оскільки клієнтська частина чату ([Frontend](#)) та серверна бізнес-логіка ([Backend](#)) під час розробки та розгортання можуть функціонувати на різних доменах, піддоменах або портах, браузері за замовчуванням блокують такі крос-доменні [HTTP](#)-запити з міркувань безпеки. Для надання санкціонованого доступу клієнту до розробленого [REST API](#) на сервері було впроваджено проміжне програмне забезпечення ([middleware](#)) [cors](#), де чітко специфіковано дозволені білі списки ([Allow-Origins](#)), [HTTP](#)-методи ([GET](#), [POST](#), [PUT](#), [DELETE](#)) та дозволені заголовки ([Headers](#)). Повний вихідний код програмної конфігурації ініціалізації сервера, підключення модулів безпеки, обробки [JSON](#)-пакетів та зчитування файлу оточення наведено у Додатку Т. 2.3.2. Система безпеки, автентифікації та захисту маршрутів Безпека користувацьких даних є пріоритетним завданням при розробці сучасних месенджерів. Система безпеки сервера

Цитирования: 0%

id: 24

«Chatty»

базується на двох фундаментальних механізмах: надійному криптографічному захисті паролів та безсесійній автентифікації на основі токенів. Хешування паролів за допомогою [bcryptjs](#): Зберігання паролів у базі даних у відкритому або просто зашифрованому вигляді є неприпустимим. Для захисту облікових записів від компрометації (наприклад, у випадку витоку бази даних) в проєкті використано бібліотеку [bcryptjs](#). Під час реєстрації користувача або зміни пароля, рядок символів проходить обробку алгоритмом адаптивного хешування з додаванням випадкової солі ([salt](#)) з коефіцієнтом розширення ([salt rounds](#)) рівним 10. Алгоритм є одностороннім — отримати вихідний пароль із готового хешу технічно неможливо, а автентифікація відбувається шляхом криптографічного порівняння введеного користувачем рядка із наявним у базі хешем за допомогою методу [bcrypt.compare\(\)](#). Автентифікація на базі [JSON Web Tokens \(JWT\)](#): Для реалізації механізму захищеного доступу до функцій месенджера було обрано концепцію [Stateless JWT](#). Після

успішного введення логіна та пароля сервер генерує унікальний рядок — [JWT](#)-токен, зашифрований секретним ключем додатка. Токен містить у своєму корисні навантаженні ([payload](#)) унікальний ідентифікатор користувача ([id](#)) та термін дії. Клієнт зберігає цей токен і додає його до заголовка [Authorization: Bearer token](#) кожного наступного [HTTP](#)-запиту. Для захисту приватних маршрутів (наприклад, отримання списку чатів чи профілю користувача) було розроблено спеціальне захисне [middleware](#) автентифікації. Воно перехоплює запит, вилучає токен, перевіряє його валідність за допомогою секретного ключа і, у разі успіху, декодує дані користувача, передаючи керування кінцевому контролеру. Якщо токен відсутній, підроблений або термін його дії закінчився, сервер миттєво блокує запит, повертаючи помилку з [HTTP](#)-статусом 401 [Unauthorized](#). Програмну реалізацію проміжного шару захисту приватних маршрутів за допомогою [JWT](#) представлено у Додатку Е. 2.3.3. Реалізація архітектури [REST API](#) та ендпоінтів Взаємодія між клієнтом та сервером для виконання стандартних операцій (реєстрація, автентифікація, робота з профілем) побудована за принципами архітектурного стилю [REST API](#). Сервер надає набір уніфікованих точок доступу (ендпоінтів), які приймають та повертають дані у форматі [JSON](#), використовуючи стандартні методи протоколу [HTTP](#). В межах розробки було спроектовано та програмно реалізовано такі ключові маршрути (маршрутизатори [Express](#)): [POST /api/auth/register](#) — ендпоінт реєстрації нового користувача. Приймає об'єкти з параметрами [nickname](#), [email](#), [password](#). Проводить валідацію на унікальність полів, викликає сервіс надсилання листів верифікації, зберігає нового користувача з відхешованим паролем та повертає згенерований [JWT](#)-токен. [POST /api/auth/login](#) — ендпоінт авторизації. Перевіряє наявність користувача за [email](#), порівнює паролі через [bcryptjs](#) і у разі успіху повертає токен для клієнтської сесії. [PUT /api/user/profile](#) — захищений маршрут для оновлення персональних даних користувача (зміна полів [displayName](#), [bio](#), [phone](#)). [POST /api/user/avatar](#) — захищений ендпоінт для завантаження або оновлення файлу зображення профілю (аватара). Детальний опис архітектури побудови маршрутизації, а також повні лістинги програмної реалізації логіки контролерів для забезпечення процесів реєстрації та автентифікації користувачів наведено у Додатку Д. 2.3.4. Обробка файлових потоків та інтеграція [Multer](#) Функціональні вимоги до месенджера

Цитування: 0%

id: 25

«Chatty»

передбачають можливість обміну медіафайлами у повідомленнях, а також завантаження персональних аватарів. Стандартний фреймворк [Express](#) не має вбудованих інструментів для аналізу багатокомпонентних даних типу [multipart/form-data](#), які використовуються для завантаження файлів через форми. Для реалізації цього функціоналу на сервері було впроваджено та налаштовано бібліотеку [Multer](#) як спеціалізоване [middleware](#) для обробки файлових потоків. [Multer](#) перехоплює вхідний потік даних, виділяє з нього файл, проводить жорстку інженерну валідацію за такими критеріями: Валідація за типом файлу ([Mimetype filter](#)): обмеження завантаження потенційно небезпечних файлів (наприклад, [.exe](#) чи [.bat](#) скриптів). Для аватарів дозволено виключно графічні формати ([image/jpeg](#), [image/png](#), [image/webp](#)). Обмеження розміру ([File size limit](#)): встановлення лімітів на максимальний розмір завантажуваного файлу для запобігання переповненню дискового простору сервера та перевантаження мережі. Після успішної перевірки [Multer](#) автоматично генерує унікальне ім'я файлу (щоб уникнути колізій при завантаженні файлів з однаковими назвами різними користувачами), зберігає його у цільову директорію на сервері або у хмарне сховище, а метадані файлу (шлях, назву, розмір, [mimetype](#)) записує в об'єкт запису [Express req.file](#) або [req.files](#). Після цього контролер повідомлень може зберегти ці дані у вкладений об'єкт [file](#) моделі [Message](#). Програмне налаштування [middleware Multer](#) для завантаження файлів та управління дисковим сховищем наведено у Додатку Є. Рисунок 2.4 — Схема процесу обробки та валідації файлових потоків за допомогою модуля [Multer](#) 2.3.5. Реалізація сервісу сповіщень на базі [Nodemailer](#) Для підтвердження реєстрації облікових записів, надсилання системних кодів та підвищення рівня безпеки до серверної частини додатка було інтегровано підсистему [Email](#)-сповіщень. Практична реалізація цього сервісу виконана за допомогою бібліотеки [Nodemailer](#). [Nodemailer](#) налаштовано на роботу за протоколом [SMTP \(Simple Mail Transfer Protocol\)](#) із використанням безпечного шифрованого з'єднання [TLS/SSL](#). Сервер бізнес-логіки автентифікується на зовнішньому поштовому шлюзі (наприклад, [SendGrid](#) або [SMTP](#)-сервері поштового провайдера) за допомогою захищених змінних оточення та ініціює відправку системних повідомлень. Архітектурною особливістю розробленого сервісу сповіщень є підтримка двомовної локалізації ([en/uk](#)), що

безпосередньо відповідає налаштуванням у профілі користувача. Перед відправкою листа сервіс перевіряє поле [locale](#) отримувача і динамічно підставляє відповідний [HTML](#)-шаблон повідомлення: При виборі локалі [uk](#) користувач отримує системний лист українською мовою з офіційним підтвердженням та інструкціями; При виборі [en](#) сервіс рендерить та відправляє повністю англomовний аналог шаблону. Програмний код модульної реалізації поштового сервісу з інтеграцією поштового шлюзу представлено у Додатку С. Загальну високорівневу логіку сервісу верифікації (обробка типів запитів на реєстрацію або відновлення доступу, генерація 6-значних числових кодів) наведено у Додатку Р. Програмний модуль конфігурації системи інтернаціоналізації ([18n](#)) розташовано у Додатку У, а файли статичних локалізованих перекладів та мовних ресурсів представлені відповідно у Додатку Х ([uk.json](#)), Додатку Ц ([es.json](#)), Додатку Ч ([en.json](#)) та Додатку Ю ([de.json](#)). 2.4. Реалізація повнодуплексного обміну повідомленнями через [Socket.io](#) Традиційні веб-технології, що базуються на класичних [HTTP](#)-запитах типу

Цитування: 0%

id: 26

«запит-відповідь»,

не здатні забезпечити ефективне функціонування сучасних інтерактивних чат-додатків. Постійне опитування сервера ([polling](#)) створює критичне навантаження на мережу та серверні потужності, а також спричиняє затримку при доставці даних. Для забезпечення миттєвого обміну інформацією між користувачами месенджера

Цитування: 0%

id: 27

«Chatty»

було впроваджено технологію [WebSocket](#) за допомогою спеціалізованого фреймворку [Socket.io](#). Це дозволило реалізувати стійке, низькозатримкове повнодуплексне (двостороннє) з'єднання, в якому як клієнт, так і сервер можуть ініціювати відправку даних у будь-який момент часу. 2.4.1. Ініціалізація [WebSocket](#)-сервера та інтеграція з [Express](#) Процес впровадження [Socket.io](#) в архітектуру серверної частини вимагає інтеграції протоколу [WebSocket](#) безпосередньо поверх базового [HTTP](#)-сервера додатка. Оскільки стандартний фреймворк [Express](#) самостійно керує лише [HTTP](#)-маршрутизацією, для розширення його можливостей було використано вбудований у [Node.js](#) модуль [http](#). Архітектурний процес ініціалізації сервера реалізовано за такою схемою: За допомогою методу [http.createServer\(app\)](#) створюється екземпляр базового веб-сервера, куди як обробник передається додаток [Express](#). Це дозволяє серверу одночасно обслуговувати як стандартні [REST API](#) ендпоінти, так і нові типи з'єднань. Створюється екземпляр сокет-сервера шляхом передачі [HTTP](#)-сервера в конструктор [new Server\(httpServer, config\)](#). У конфігурації ініціалізації сокетів обов'язково налаштовуються параметри політики безпеки [CORS](#). Серверу сокетів передаються санкціоновані параметри клієнтського домену та дозволені методи обміну ([methods](#)): [

Цитування: 0%

id: 28

"GET",

Цитування: 0%

id: 29

"POST"

]), що унеможливорює несанкціоноване підключення сторонніх скриптів до каналів зв'язку. Після успішної ініціалізації сервер починає прослуховувати визначений порт, де під час першого звернення клієнта відбувається процедура [HTTP Handshake](#) (рукостискання), після чого з'єднання автоматично та безперервно переводиться на захищений повнодуплексний протокол [WebSocket](#). 2.4.2. Архітектура подій ([Event-Driven Architecture](#)) Взаємодія в [Socket.io](#) побудована на базі подійно-орієнтованої архітектури ([Event-Driven Architecture](#)). Замість обробки статичних маршрутів, сервер і клієнт підписуються на певні іменовані події та надсилають (генерують) їх разом із корисним навантаженням у форматі [JSON](#). У межах розробки бізнес-логіки чату

Цитування: 0%

id: 30

«Chatty»

на серверній стороні було спроектовано та реалізовано систему обробників для таких базових подій: [connection](#) — головна системна подія, яка спрацьовує, коли новий клієнт успішно пройшов етап рукостискання. Сервер фіксує унікальний ідентифікатор сокета

([socket.id](#)), що виділяється кожній активній вкладці браузера, та відкриває індивідуальний асинхронний канал зв'язку з цим користувачем. [sendMessage](#) — користувацька подія, яка генерується клієнтом у момент натискання кнопки відправки повідомлення. Обробник цієї події на сервері приймає об'єкт повідомлення (ідентифікатор чату, текст, метадані файлу). Сервер проводить первинну валідацію отриманих даних, асинхронно звертається до [ODM](#) [Mongoose](#) для фізичного збереження запису в базі даних [MongoDB Atlas](#), після чого ініціює ретрансляцію даних. [message](#) — подія доставки повідомлення. Після успішного збереження в базі даних сервер пересилає сформований документ повідомлення цільовим учасникам чату. Завдяки цьому отримувач миттєво бачить нове повідомлення у своєму вікні інтерфейсу, а відправник отримує підтвердження успішної доставки. [typing](#) / [stopTyping](#) — події для забезпечення інтерактивності інтерфейсу (індикатори введення тексту). Коли користувач починає набирати текст у полі введення, клієнтська частина з невеликою затримкою ([debounce](#)) генерує подію [typing](#) та передає її на сервер разом із [ID](#) чату. Сервер миттєво пересилає цю подію іншому учаснику діалогу, що дозволяє відобразити на екрані анімоване сповіщення

Цитирования: 0,01%

id: 31

«користувач набирає повідомлення...».

При припиненні введення генерується подія [stopTyping](#), яка прибирає індикатор. [disconnect](#) — системна подія розриву з'єднання. Спрацьовує автоматично, коли користувач закриває вкладку застосунку, оновлює сторінку або втрачає мережеве підключення. Обробник на сервері коректно очищує пам'ять від неактивного сокета, видаляє користувача зі списку присутніх у кімнатах чату та може транслявати іншим учасникам статус

Цитирования: 0%

id: 32

«офлайн».

Рисунок 2.5 — Діаграма послідовності обробки мережевих подій у реальному часі 2.4.3. Менеджмент кімнат ([Rooms](#)) та ізоляція приватних чатів Критично важливим завданням при розробці архітектури месенджера є ізоляція потоків даних. Повідомлення, надіслане у приватний чат між Користувачем А та Користувачем Б, у жодному разі не повинно трансляватися іншим підключеним до сервера клієнтам. Для вирішення цього завдання було використано вбудований у [Socket.io](#) потужний механізм кімнат ([Rooms](#)). Кімнати — це віртуальні ізольовані канали на рівні сервера, в які сокети користувачів можуть об'єднуватися для спільного прийому та передачі подій. Логіка менеджменту приватних чатів у

Цитирования: 0%

id: 33

«[Chatty](#)»

побудована за таким алгоритмом: Під час відкриття вікна діалогу клієнт ініціює запит на підключення до конкретного простору спілкування. Сервер отримує унікальний ідентифікатор чату ([chatId](#)) з моделі даних [Chat](#). За допомогою вбудованого методу [socket.join\(chatId\)](#) сервер примусово додає поточний сокет користувача у віртуальну кімнату, назвою якої є сам [chatId](#). Якщо кімнати з такою назвою ще не існувало в оперативній пам'яті сервера, [Socket.io](#) створює її автоматично. Коли один із учасників надсилає повідомлення через подію [sendMessage](#), сервер не робить загальну розсилку ([broadcast](#)) всім користувачам мережі. Замість цього він використовує конструкцію [io.to\(chatId\).emit\('message', data\)](#), яка здійснює цільову доставку повідомлення виключно тим сокетам, що зараз додані до цієї конкретної кімнати [chatId](#). Під час виходу з чату або відкриття іншого діалогу автоматично викликається метод [socket.leave\(chatId\)](#), що відключає користувача від поточного віртуального каналу та захищає конфіденційність листування. Така архітектурна організація за допомогою кімнат забезпечує високу безпеку системи, надійну ізоляцію приватних даних, а також відмінну масштабованість, дозволяючи без зміни логіки сервера в майбутньому створювати групові кімнати на будь-яку кількість учасників. Повний лістинг програмної реалізації ініціалізації [WebSocket](#)-сервера, конфігурації [CORS](#) для сокетів та обробників усієї подійно-орієнтованої логіки чату разом із менеджментом кімнат винесено у Додаток Г. Рисунок 2.6 — Схема ізоляції приватних чатів через механізм віртуальних кімнат [Socket.io Rooms](#) 2.5. Розробка клієнтської частини ([Frontend](#)) та інтерфейсу користувача Клієнтська частина ([Frontend](#)) веб-застосунку

Цитирования: 0%

id: 34

«Chatty»

є основним середовищем взаємодії кінцевого користувача з системою. Оскільки месенджер орієнтований на повсякденне використання, до його інтерфейсу висуваються високі вимоги щодо швидкості відгуку, інтерактивності, ергономіки (UX) та візуальної привабливості (UI). Клієнтська частина розроблена як SPA-орієнтована (Single Page Application) система на базі сучасного технологічного стеку HTML5, CSS3 та JavaScript (ES6+), що дозволило забезпечити плавну роботу інтерфейсу без постійного перезавантаження сторінок браузера.

2.5.1. Модульна структура веб-додатку на стороні клієнта Для забезпечення чистоти коду, простоти тестування та зручності подальшого розширення функціоналу, архітектуру Frontend-частини було спроектовано за модульним принципом. Весь клієнтський код чітко розподілено на функціональні шари: Шар представлення (HTML5 Documents / Templates): відповідає за декларативну розмітку структури сторінок та елементів керування. Використання семантичних тегів HTML5 (aside, main, section, header) дозволило створити логічний каркас додатка, оптимізований для зчитування браузерними рушіями та забезпечення доступності (Accessibility). Шар стилізації (CSS3 Modules / Themes): ізольовані файли стилів, які відповідають за геометрію інтерфейсу, шрифти, адаптивну сітку та анімаційні ефекти. Організація стилів базується на використанні CSS-змінних (Custom Properties), що є ключовим інструментом для динамічної кастомізації. Шар логіки та управління станом (JavaScript Modules): ядро клієнтської частини, що розподілене на окремі логічні модулі: api.js — модуль, що відповідає за відправку асинхронних HTTP-запитів (fetch/axios) до REST API сервера для автентифікації та завантаження профілю; socket.js — модуль, який ініціалізує WebSocket-з'єднання та інкапсулює в собі обробники подій реального часу; ui.js / dom.js — модуль маніпуляції елементами інтерфейсу, що відповідає за динамічний рендеринг повідомлень та оновлення екрана. Взаємодію між архітектурними шарами представлення, стилізації та логіки керування станом додатка на стороні браузера схематично зображено на рисунку 2.7

Рисунок 2.7 — Модульна структура та схема взаємодії компонентів клієнтської частини

2.5.2. Стилзація, адаптивний дизайн та кастомізація інтерфейсу Візуальна складова месенджера розроблена з використанням сучасних методологій CSS-верстки. Головними завданнями при проектуванні UI-шару були забезпечення адаптивності та реалізація механізму зміни тем. Адаптивний дизайн (Responsive Web Design): для того, щоб інтерфейс однаково зручно функціонував як на моніторах десктопів, так і на екранах смартфонів, було використано технології CSS Flexbox та CSS Grid Layout у поєднанні з медіа-запитами (@media). На великих екранах додаток має класичну двопанельну структуру (ліворуч — список чатів та контактів, праворуч — активне вікно діалогу). При зменшенні ширини екрана (на мобільних пристроях) інтерфейс динамічно трансформується у однопанельний режим, де список чатів та відкритий діалог перемикаються за допомогою плавних анімаційних переходів, оптимізованих під сенсорне керування. Реалізація світлої та темної тем (Theme Customization): кастомізація колірної палітри реалізована за допомогою глобальних CSS-змінних, оголошених у псевдокласі :root. Було створено два набори колірних схем (для світлого та темного режимів), які керують колірним фоном, текстом, картками повідомлень та елементами навігації. Перемикання тем відбувається шляхом додавання або видалення дата-атрибута (наприклад, data-theme=

Цитування: 0%

id: 35

"dark"

) до кореневого елемента html за допомогою JavaScript. Обрана користувачем тема автоматично фіксується у локальному сховищі браузера (localStorage), що дозволяє зберігати налаштування при повторних візитах на сайт. Рисунок 2.8 — Інтерфейс користувача веб-застосунку

Цитування: 0%

id: 36

«Chatty»

у десктопній (Light Mode) та (Dark Mode) версіях Файлові прев'ю та візуальні ефекти: для покращення користувацького досвіду (UX) при обміні медіафайлами було розроблено модуль попереднього перегляду (File Preview). Коли користувач обирає файл або зображення для відправки, JavaScript за допомогою API URL.createObjectURL() або FileReader миттєво генерує тимчасове локальне посилання на файл в пам'яті браузера. Це дозволяє відобразити мініатюру зображення або іконку документа з індикатором завантаження

безпосередньо у полі введення тексту до того, як файл буде фізично відправлено на сервер. Також реалізовано плавні [CSS](#)-анімації для демонстрації статусів надсилання, появи нових повідомлень та ефекту

Цитування: 0%

id: 37

«пульсації»

для індикатора введення тексту іншим користувачем. Візуальну реалізацію компонента попереднього перегляду медіафайлів усередині форми введення тексту наведено на рисунку 2.9 Рисунок 2.9 — Інтерфейс компонента попереднього перегляду медіафайлів ([File Preview](#)) перед відправкою 2.5.3. Клієнтська [WebSocket](#)-логіка та динамічне оновлення [DOM](#) Ключовою особливістю [Frontend](#)-частини месенджера

Цитування: 0%

id: 38

«Chatty»

є робота з асинхронними потоками даних у реальному часі без перезавантаження сторінки. Цей функціонал забезпечується клієнтською бібліотекою [socket.io-client](#). Під час успішного проходження авторизації додаток ініціює підключення до сервера за допомогою методу [io\(SERVER_URL\)](#). Клієнтський сокет отримує унікальний ідентифікатор і підписується на події сервера. Процес динамічного оновлення інтерфейсу під час обміну повідомленнями відбувається за таким алгоритмом: Користувач вводить текст, прикріплює файл і натискає кнопку

Цитування: 0%

id: 39

«Надіслати».

[JavaScript](#)-контролер перехоплює подію відправки форми ([submit](#)), блокує стандартну поведінку браузера (щоб сторінка не перезавантажувалася) та формує об'єкт даних. Клієнт генерує подію [socket.emit\('sendMessage', messageData\)](#). При отриманні сервером цього повідомлення та його ретрансляції у кімнату чату, у сокета отримувача (та відправника) спрацьовує слухач події [socket.on\('message', \(newMessage\) => { ... }\)](#). В середині цього обробника викликається функція динамічного оновлення [DOM](#)-дерева ([Document Object Model](#)). [JavaScript](#) за допомогою методів [document.createElement\(\)](#) створює нові блоки [HTML](#)-розмітки для картки повідомлення, наповнює їх отриманим текстом, метаданими файлу та міткою часу, після чого інтегрує новий елемент у кінець списку повідомлень за допомогою методу [.appendChild\(\)](#). Одразу після модифікації [DOM](#)-дерева викликається метод автоматичного скролінгу [.scrollIntoView\({ behavior: 'smooth' }\)](#), який плавно зміщує екран до останнього надісланого повідомлення, забезпечуючи природну поведінку стрічки чату. Такий підхід дозволяє досягти монолітності інтерфейсу, коли всі зміни — поява тексту, зміна статусів, робота індикаторів введення тексту ([typing](#)) — відбуваються миттєво та непомітно для користувача на рівні оперативної пам'яті браузера, гарантуючи найвищу швидкість роботи веб-застосунку. Повні структури стилів кастомізації тем, [CSS](#)-конфігурацію адаптивної верстки, а також лістинги клієнтського модуля обробки сокетних подій та динамічної маніпуляції [DOM](#)-деревом винесено у Додаток Т. ВИСНОВКИ ДО РОЗДІЛУ 2 У другому розділі кваліфікаційної роботи було виконано повний комплекс інженерних завдань, пов'язаних із системним аналізом технічних вимог, архітектурним проектуванням, розробкою структури бази даних, а також програмною реалізацією серверної та клієнтської частин веб-застосунку

Цитування: 0%

id: 40

«Chatty»

для обміну повідомленнями в режимі реального часу. На основі проведених досліджень та виконаної розробки можна зробити такі науково-практичні висновки: За результатами аналізу вимог та архітектурного проектування було обґрунтовано доцільність застосування класичної трирівневої архітектурної моделі ([Three-tier Architecture](#)). Розподіл системи на ізольовані шари — рівень представлення ([Frontend](#)), рівень бізнес-логіки ([Backend](#)) та рівень даних ([Database](#)) — дозволив створити гнучку структуру з чітким розмежуванням зон відповідальності компонентів. Такий підхід забезпечує високий рівень безпеки (виключаючи пряму взаємодію клієнта з базою даних) та створює технологічний фундамент для незалежного масштабування кожного рівня системи в умовах зростання кількості активних користувачів. Впровадження документоорієнтованої [NoSQL](#) СУБД [MongoDB](#) (на базі хмарного кластера [MongoDB Atlas](#)) виявилось найбільш ефективним

інженерним рішенням для специфіки [real-time](#) месенджера у порівнянні з реляційними аналогами. Гнучка структура документів у форматі [JSON](#) дозволила уникнути складних операцій об'єднання таблиць ([JOIN](#)) та ресурсомістких міграцій на етапі розширення функціоналу. Застосування концепції денормалізації даних, зокрема дублювання копії останнього повідомлення безпосередньо у документ моделі чату ([lastMessage](#)), дозволило мінімізувати кількість запитів до бази даних під час рендерингу списків, суттєво знизивши навантаження на дискову підсистему та хмарний процесор. Використання [ODM](#)-бібліотеки [Mongoose](#) забезпечило строгу типізацію та валідацію схем даних на рівні прикладного коду сервера. Розробка серверної частини на базі платформи [Node.js](#) та фреймворку [Express](#) дозволила реалізувати високопродуктивне асинхронне [REST API](#) для виконання стандартних операцій (реєстрація, автентифікація, менеджмент профілів). Безпека користувацьких даних була забезпечена впровадженням сучасних криптографічних протоколів: одностороннього хешування паролів за допомогою алгоритму [bcryptjs](#) із додаванням випадкової солі, а також механізму безсесійної автентифікації на основі токенів [JSON Web Tokens \(JWT\)](#). Це дозволило захистити приватні маршрути додатка та знизити навантаження на оперативну пам'ять сервера за рахунок реалізації архітектурного принципу [Stateless](#). Інтеграція спеціалізованого [middleware Multer](#) забезпечила надійне керування багатокомпонентними файловими потоками ([multipart/form-data](#)). Створена система інженерної валідації вхідних файлів за [MIME](#)-типом та максимальним розміром дозволила нейтралізувати ризики завантаження потенційно небезпечних скриптів та переповнення серверного сховища. Впровадження сервісу сповіщень на базі бібліотеки [Nodemailer](#) за протоколом [SMTP](#) із шифруванням [TLS](#) забезпечило безпечну верифікацію користувачів під час реєстрації, а інтеграція модулів інтернаціоналізації (i18n) дозволила динамічно адаптувати [HTML](#)-шаблони системних листів під обрану локаль користувача. Реалізація повнодуплексного обміну даними була успішно виконана за допомогою [WebSocket](#)-фреймворку [Socket.io](#). Подійно-орієнтована архітектура ([Event-Driven Architecture](#)) дозволила відійти від неефективних методів постійного опитування сервера ([polling](#)) та знизити затримку доставки повідомлень ([latency](#)) до нормативних 200–300 мс. Використання вбудованого механізму віртуальних кімнат ([Rooms](#)) забезпечило абсолютну ізоляцію приватних каналів зв'язку між користувачами, гарантуючи конфіденційність листування та виключаючи можливість несанкціонованого перехоплення даних сторонніми клієнтами сокет-сервера. Проектування та розробка клієнтської частини ([Frontend](#)) у концепції [Single Page Application \(SPA\)](#) на базі [HTML5](#), [CSS3](#) та [JavaScript \(ES6+\)](#) дозволили створити монолітний інтерактивний інтерфейс. Використання сучасних методологій [CSS Flexbox](#) та [CSS Grid](#) у поєднанні з медіа-запитами ([@media](#)) забезпечило 100% адаптивність дизайну додатка для мобільних та десктопних пристроїв. Динамічна кастомізація кольірних схем ([Light/Dark](#) режими) була реалізована через глобальні [CSS](#)-змінні із фіксацією вибору користувача у локальному сховищі браузера ([localStorage](#)). Завдяки використанню клієнтського сокет-клієнта та методів маніпуляції [DOM](#)-деревом (зокрема, [appendChild](#) та [scrollIntoView](#)), усі інтерфейсні події — від появи повідомлень до роботи анімованих індикаторів введення тексту ([typing](#)) — відбуваються миттєво в оперативній пам'яті браузера без перезавантаження веб-сторінок. Таким чином, розроблений у другому розділі програмно-архітектурний комплекс повністю відповідає висунутим функціональним та нефункціональним технічним вимогам, є масштабованим, безпечним та оптимізованим для стабільної роботи в умовах реального часу, що створює необхідну базу для подальшого розгортання, тестування та впровадження продукту. ВИСНОВОК У кваліфікаційній роботі успішно вирішено науково-практичне завдання з дослідження, проектування, програмної реалізації та тестування кросплатформеного веб-застосунку для обміну повідомленнями в режимі реального часу

Цитування: 0%

id: 41

«Chatty».

Створений програмний продукт повністю відповідає інженерним стандартам, критеріям безпеки та специфікаціям технічного завдання. На основі проведеного аналізу теоретичних засад розподілених систем обґрунтовано доцільність застосування трирівневої архітектури, яка дозволила чітко розмежувати клієнтський шар представлення, серверну логіку та шар збереження даних, мінімізуючи ризики прямого доступу до СКБД. Дослідження мережевих протоколів взаємодії довело неефективність традиційних [HTTP](#)-підходів, натомість інтеграція повнодуплексного подієво-орієнтованого

протоколу [WebSocket](#) забезпечила миттєву доставку повідомлень у межах нормативних 200–300 мс. Для побудови гнучкої та стійкої структури бази даних було обрано [NoSQL](#) СУБД [MongoDB](#) у хмарі [MongoDB Atlas](#). Відмова від реляційної моделі на користь [BSON](#)-документів та впровадження концепції денормалізації через дублювання об'єкта останнього повідомлення у колекції чатів дозволили оптимізувати вибірку даних та знизити навантаження на хмарний кластер, а використання [ODM Mongoose](#) забезпечило строгу валідацію схем на рівні бекенду. Серверна частина застосунку, конфігурована на базі платформи [Node.js](#) та фреймворку [Express](#) з асинхронною моделлю введення-виведення, гарантує стабільну обробку великої кількості одночасних підключень. Безпека системи реалізована за принципом [Stateless](#) на основі токенів [JSON Web Tokens](#) та криптографічного хешування паролів алгоритмом [bcryptjs](#). Налаштування [middleware Multer](#) забезпечило жорстку валідацію вхідних медіафайлів, а впровадження модуля [Nodemailer](#) за протоколом [SMTP](#) — надійну верифікацію користувачів. Інтеграція інструментів інтернаціоналізації ([i18n](#)) дозволила динамічно змінювати мову системних повідомлень відповідно до локалі користувача. Ядро реального часу побудовано на базі [Socket.io](#), де завдяки механізму віртуальних кімнат ([Rooms](#)) досягнуто абсолютної ізоляції приватних каналів спілкування та інтерактивних індикаторів введення тексту ([typing](#)). Клієнтську частину реалізовано в концепції [Single Page Application \(SPA\)](#) за допомогою стеків [HTML5](#), [CSS3](#) та асинхронного [JavaScript \(ES6+\)](#). Завдяки методологіям [CSS Flexbox](#), [Grid](#) та медіа-запитам забезпечено 100% адаптивність та кросбраузерність інтерфейсу додатка для мобільних і десктопних пристроїв, а також впроваджено динамічну кастомізацію тем ([Light/Dark](#)) через локальне сховище браузера. Створення компонента попереднього перегляду файлів через [File API](#) покращило користувацький досвід ([UX](#)), а прямі маніпуляції з [DOM](#)-деревом дозволили досягти монолітності інтерфейсу, оскільки всі оновлення екрана відбуваються без перезавантаження сторінок. Практична цінність роботи полягає у створенні готового до розгортання, масштабованого та безпечного месенджера. Розроблений [Fullstack JavaScript](#) комплекс може слугувати технологічним прототипом для корпоративних чат-систем або базою для подальших досліджень у сфері оптимізації високозавантажених [real-time](#) застосунків. Поставлені завдання роботи виконано в повному обсязі, а мету дослідження успішно досягнуто.

Заявление об ограничении ответственности:

Этот отчет должен быть правильно истолкован и проанализирован квалифицированным специалистом, который несет ответственность за оценку!

Любая информация, представленная в этом отчете, не является окончательной и подлежит ручному просмотру и анализу. Пожалуйста, следуйте инструкциям: [Рекомендации по оценке](#)

Детектор Плагіата - Ваше право на оригінальність! ☐ SkyLine LLC