ВСТУП

Актуальність теми. Сучасний етап розвитку корпоративного управління, промислового виробництва та функціонування державних установ в Україні характеризується стрімким розширенням масштабів цифровізації, що вимагає впровадження високонадійних засобів обміну оперативною інформацією. Ефективність внутрішньої взаємодії безпосередньо впливає на швидкість прийняття управлінських рішень та стабільність критичних бізнес-процесів організації.

Проте використання публічних хмарних месенджерів (таких як Telegram, Viber, WhatsApp або комерційних SaaS-платформ типу Slack та Microsoft Teams) у внутрішньому контурі підприємств створює критичні загрози для інформаційної безпеки [1]. Основними ризиками є можливість перехоплення конфіденційних даних на закордонних транзитних серверах, залежність від стабільності глобальних каналів зв'язку Інтернет, а також потенційна наявність недекларованих можливостей (бекдорів) у закритому вихідному коді пропрієтарного програмного забезпечення.

Для підприємств оборонно-промислового комплексу, банківських установ, енергетичних компаній та органів державної влади єдиним архітектурним рішенням, що гарантує абсолютний контроль над інформаційними потоками, є використання комунікаційних систем класу on-premise, які розгортаються суворо всередині ізольованої локальної обчислювальної мережі (ЛОМ) організації [2]. Створення таких систем вимагає глибокого розуміння мережевих стеків протоколів, технологій асинхронного паралельного програмування та методів побудови транзакційних сховищ даних.

Найбільш перспективним інструментом для створення високопродуктивних мережевих застосунків є серверна платформа Node.js, яка завдяки подійній моделі та неблокуючому вводу-виводу (двигун V8 та бібліотека libuv) дозволяє ефективно масштабувати систему в умовах

обмежених апаратних ресурсів локального сервера [3]. У поєднанні з протоколом двостороннього зв’язку WebSockets та компонентним клієнтським фреймворком React, даний стек дозволяє побудувати відмовостійку, швидку та безпечну інформаційну систему обміну даними.

Таким чином, розробка чат-системи для локальної мережі організації з використанням Node.js та React є актуальним інженерним завданням у межах спеціальності «Комп'ютерна інженерія».

Об'єкт дослідження – процес автоматизованого асинхронного обміну миттєвими повідомленнями в ізольованих локальних обчислювальних мережах організацій.

Предмет дослідження – методи, алгоритми, архітектурні рішення, протоколи передачі даних реального часу та програмне забезпечення клієнт-серверної інформаційної системи на базі платформ Node.js, React та СУБД PostgreSQL.

Мета роботи – проєктування, алгоритмічне обґрунтування, програмна реалізація та експериментальна верифікація захищеної веб-орієнтованої чат-системи реального часу для локальної мережі організації, що забезпечує мінімальні затримки доставки повідомлень, високу стійкість до відмов та відповідність вимогам інформаційної безпеки.

Для досягнення поставленої мети необхідно вирішити такі **завдання**:

− Проаналізувати нормативно-правове забезпечення України у сфері захисту інформації в інформаційно-комунікаційних системах, сформувати профіль користувачів та специфікувати загрози безпеці даних у ЛОМ.

− Провести порівняльний аналіз архітектур передачі даних реального часу та систем керування базами даних, обґрунтувати вибір технологічного стеку розробки.

− Розробити логічну та фізичну структуру реляційного сховища даних на базі СУБД PostgreSQL з урахуванням каскадних обмежень та вимог цілісності ACID.

− Побудувати математичну модель функціонування асинхронного сервера чат-системи на основі теорії масового обслуговування для оцінки швидкодії при пікових навантаженнях.

− Розробити подієво-орієнтований серверний програмний комплекс на базі Node.js (Express, Socket.io) з інтеграцією безсесійних механізмів автентифікації JWT та хешування Bcrypt.

− Створити інтерактивний клієнтський SPA-інтерфейс на базі React з оптимізованим механізмом оновлення віртуального дерева елементів (Virtual DOM) та превентивним захистом від XSS-атак.

− Провести автоматизоване інтеграційне тестування розроблених маршрутів.

Методи дослідження. Для вирішення поставлених завдань у роботі використано такі методи: методи системного та структурного аналізу – при дослідженні інформаційних потоків і побудові загальної архітектури веб-додатка; методи теорії масового обслуговування та теорії ймовірностей – для математичного моделювання навантаження на серверне ядро; методи об'єктно-орієнтованого та компонентного програмування – для побудови модульної структури клієнтської та серверної частин; методи криптографічного захисту інформації (хешування та симетричного підпису) – для реалізації підсистеми безпеки сесій.

Практичне значення отриманих результатів. Розроблена чат-система є повністю автономним, працездатним програмним продуктом класу on-premise, який може бути безпосередньо впроваджений у практику роботи будь-якої вітчизняної організації, установи чи промислового підприємства для розгортання захищеної внутрішньої комунікаційної інфраструктури. Код системи є платформонезалежним, легко масштабується і може служити базою для проектування ширших корпоративних експертних систем у галузі комп'ютерної інженерії.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ КОРПОРАТИВНИХ КОМУНІКАЦІЙ ТА ПОСТАНОВКА ЗАВДАННЯ

1.1 Аналіз нормативно-правової бази захисту інформації та профілів користувачів ЛОМ

Забезпечення інформаційної безпеки та захисту даних є фундаментальною вимогою при проєктуванні та розгортанні комунікаційних платформ усередині сучасних підприємств та організацій. На відміну від публічних розважальних сервісів, корпоративна чат-система оперує потоками інформації, яка безпосередньо підпадає під дію законодавства України у сфері захисту інформації та персональних даних. Побудова архітектури локальної чат-системи повинна суворо спиратися на законодавчі акти та нормативні документи технічного захисту інформації (НД ТЗІ), що діють в Україні.

Основним законодавчим фундаментом у цій галузі є Закон України «Про захист інформації в інформаційно-комунікаційних системах» [4]. Цей закон чітко визначає, що державні інформаційні ресурси або конфіденційна інформація, яка є власністю держави, а також інформація з обмеженим доступом (включаючи комерційну таємницю компаній), повинна оброблятися виключно в системах, де створено комплексну систему захисту інформації (КСЗІ) з підтвердженою відповідністю. Стаття 8 цього Закону наголошує на персональній відповідальності керівників організацій за забезпечення належного рівня захисту даних. При використанні публічних месенджерів (наприклад, Telegram або WhatsApp) конфіденційні тексти та файли передаються через сторонні закордонні проксі-сервери, що автоматично унеможливлює контроль за копіюванням даних та створює пряме порушення вимог законодавства щодо локалізації обробки критичних даних.

Поряд із цим, функціонування корпоративного месенджера нерозривно пов'язане з обробкою облікових записів, паспортних відомостей співробітників, їхніх посад, корпоративних e-mail адрес та безпосереднього змісту

повідомлень, що згідно із Законом України «Про захист персональних даних» [5] визначається як обробка персональних даних. Відповідно до статті 22 цього Закону, володільці персональних даних зобов'язані забезпечити технічний та системний захист цих масивів від випадкової втрати, знищення або незаконного доступу.

З інженерної точки зору, проектування локальної системи «CorpChat» спрямоване на повне виконання вимог міжнародного стандарту ISO/IEC 27001 (ДСТУ ISO/IEC 27001) «Інформаційні технології. Методи захисту. Системи управління інформаційною безпекою» [6]. У межах даного стандарту реалізується тріада безпеки: конфіденційність (Confidentiality), цілісність (Integrity) та доступність (Availability).

Для деталізації архітектурних вимог до швидкодії, рівнів доступу та інтерфейсу розроблюваної системи було виконано системний аналіз та побудовано профілі потенційних користувачів (акторів) системи в межах типової організації:

Рядові співробітники (Кінцеві користувачі). Використовують систему як щоденний робочий інструмент для обміну текстовими повідомленнями, координації завдань усередині відділів та передачі робочих файлів (документів, звітів, специфікацій). Для даної категорії критично важливою є максимальна швидкість реакції інтерфейсу (мінімальний час затримки при відправці/отриманні повідомлень, який не повинен перевищувати 100-150 мілісекунд), наявність візуальних індикаторів мережевого статусу колег (онлайн/офлайн) та простота навігації між робочими каналами. Доступ до системи здійснюється через локальні веб-браузери без необхідності встановлення стороннього пропрієтарного софту на робочі станції.

Керівники підрозділів (Модератори каналів). Цей профіль користувачів потребує розширених функціональних можливостей для створення тематичних або міжвідомчих каналів зв'язку, управління списком учасників конкретних кімнат, очищення або архівування застарілих гілок обговорень. З погляду безпеки, керівники відділів повинні мати можливість

створювати закриті (приватні) канали для обговорення стратегічних або фінансових питань, доступ куди жорстко обмежений списком токенів авторизації.

Системні адміністратори (Адміністратори безпеки ІС). Головні технічні актори, які здійснюють контроль за загальним станом працездатності системи, керують пулом підключень до бази даних, виконують резервне копіювання (backup) транзакційного сховища PostgreSQL та здійснюють аудит дій користувачів. Для адміністраторів критично важливою є наявність зручних інструментів моніторингу логів сервера, низьке споживання апаратних ресурсів (RAM, CPU) серверною частиною застосунку та можливість миттєвого блокування скомпрометованих облікових записів співробітників без зупинки всього комунікаційного сервера.

Проте детальний аналіз специфіки передачі інформації всередині стандартних локальних обчислювальних мереж організацій виявив ряд серйозних технічних викликів та потенційних загроз (Attack Vectors), які необхідно врахувати на етапі проєктування системи:

Загроза внутрішнього перехоплення трафіку (Packet Sniffing / MITM). Поширена помилка при розробці локального софту полягає у відмові від шифрування, оскільки трафік «не виходить за межі будівлі». На практиці, зловмисник або компрометована робоча станція всередині ЛОМ за допомогою утиліт типу Wireshark може здійснювати перехоплення незахищених HTTP/WS пакетів, отримуючи доступ до паролів у відкритому вигляді та змісту листування відділів. Це вимагає обов'язкового використання захищених протоколів HTTPS та WSS (WebSockets Secure) із шифруванням TLS на рівні локального сервера [1].

Проблема нерегулярних розривів мережевих сесій. У великих корпоративних мережах із складною топологією та використанням бездротових точок доступу Wi-Fi виникає проблема частих короткочасних втрат пакетів або перемикання клієнтів між шлюзами. Традиційні TCP-з'єднання у таких випадках викликають критичні помилки на стороні сервера, що призводить до

«зависання» інтерфейсу користувача. Отже, архітектура системи повинна містити інтелектуальний шар автоматичного відновлення з'єднання (Auto-Reconnection Lifecycle) на рівні WebSocket-клієнта.

Ризик XSS-ін'єкцій через поля введення повідомлень (Cross-Site Scripting). Оскільки чат-система передбачає динамічний рендеринг тексту, надісланого іншими користувачами, існує критична небезпека виконання шкідливого JavaScript-коду в браузерах усіх учасників каналу, якщо один із користувачів надішле в чат рядок виду <script>malicious_code()</script>. Це вимагає проєктування превентивного шару санітизації (Sanitization) та екранування вхідних рядкових байтових потоків до моменту їх рендерингу у Virtual DOM фреймворку React.

1.2 Порівняльний аналіз архітектур корпоративних систем

Успішне проєктування високоефективної інформаційної системи обміну повідомленнями реального часу потребує глибокого аналізу існуючих технологічних підходів до організації мережевої взаємодії та збереження персистентного стану. Ключовим інженерним викликом є вибір між різними моделями побудови клієнт-серверного зв'язку: традиційними HTTP-запитами (включаючи AJAX, Polling, Long Polling) та сучасним дуплексним протоколом WebSockets.

Історично у багатьох веб-додатках використовувався метод короткого опитування (Short Polling), при якому клієнтський інтерфейс із фіксованою періодичністю (наприклад, кожні 2 секунди) надсилає на сервер асинхронні HTTP-запити GET /api/messages для перевірки наявності нових повідомлень. Даний підхід демонструє катастрофічно низьку обчислювальну ефективність. Кожен окремий HTTP-запит змушує систему заново виконувати тристороннє TCP-рукостискання (Three-way Handshake), передавати масивні заголовки (Headers), які часто перевищують обсяг корисного навантаження (Payload) в

десятки разів, та створювати надлишкове навантаження на процесор сервера, який змушений постійно відкривати й закривати системні дескриптори файлів.

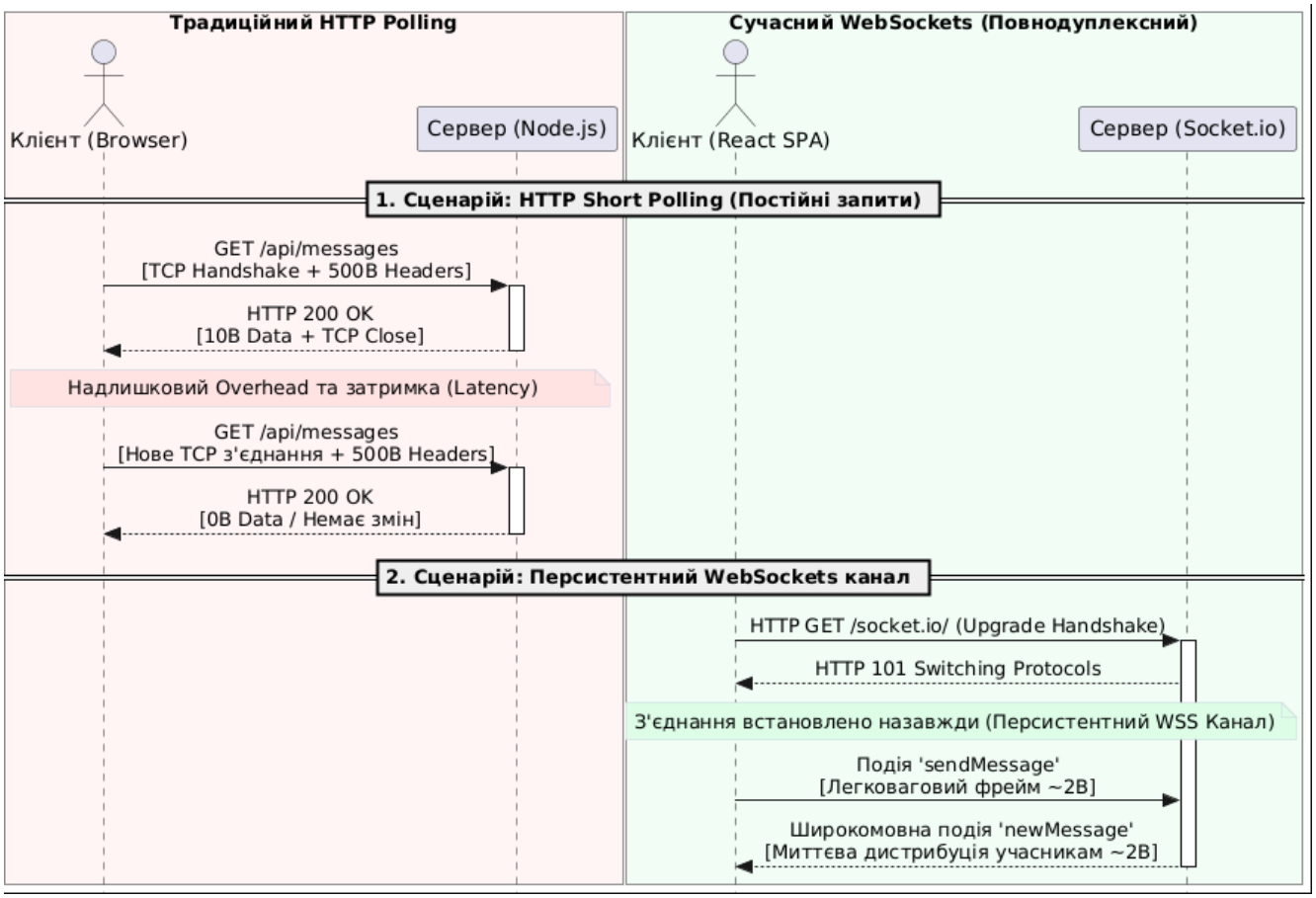

Рисунок 1.1 – Порівняльний аналіз структурної організації та накладних витрат мережевих стеків

Модифікація Long Polling частково вирішує проблему, утримуючи HTTP-з'єднання відкритим до моменту появи нових даних на сервері. Проте при високій інтенсивності спілкування у корпоративних чатах Long Polling все одно вироджується у серію послідовних HTTP-запитів, що призводить до так званого ефекту «join pain» на рівні мережевих стеків сервера та викликає затримки відображення повідомлень, що є неприпустимим для систем реального часу.

Для подолання вказаних архітектурних обмежень у межах розробки інформаційного комплексу «CorpChat» обґрунтовано перехід до повнодуплексного протоколу WebSockets (RFC 6455) [7]. На відміну від HTTP,

який працює за строгою парадигмою «запит-відповідь», WebSockets дозволяє встановити єдине персистентне TCP-з'єднання між клієнтом і сервером за допомогою початкової процедури Upgrade (WebSocket Handshake). Після успішного перемикання протоколу обидві сторони можуть надсилати дані в будь-який момент часу у вигляді легковагових фреймворкових бінарних або текстових пакетів з мінімальним розміром заголовка (всього від 2 до 10 байт). Порівняльний аналіз накладних витрат мережевого трафіку для HTTP Polling та WebSockets наведено на рисунку 1.1.

З погляду серверної обробки, використання WebSockets вимагає застосування спеціалізованих бібліотек абстракції, найбільш стабільною серед яких є Socket.io для екосистеми Node.js. Socket.io реалізує патерн проєктування «Подієво-орієнтована шина» (Event Emitter), автоматично підтримує логіку кімнат (Rooms) для ізоляції чат-каналів та забезпечує прозорий фолбек (fallback) на Long Polling, якщо мережеве обладнання локальної мережі організації блокує прямі WebSocket-фрейми.

1.3 Огляд аналогів

Щоб сформувати концепцію програмного продукту, було виконано огляд сучасного програмного забезпечення та технічних рішень, які можуть бути використані для створення веб-застосунку, що автоматизує обмін корпоративними повідомленнями в локальній мережі. Розглянуто як спеціалізовані корпоративні продукти, так і популярні публічні платформи, здатні частково закривати потреби бізнесу у комунікації.На сучасному ринку програмного забезпечення для комунікацій та автоматизації бізнес-процесів представлено низку потужних сервісів і платформ, які користуються популярністю у різних категорій корпоративних користувачів. Серед них можна виділити такі системи, як Slack, Microsoft Teams та Mattermost. Вони пропонують універсальні інструменти для обміну повідомленнями, організації відеоконференцій та оптимізації внутрішніх операцій. Подібні платформи

дозволяють ефективно організовувати командну роботу, структурувати інформаційні потоки та підтримувати сталу продуктивність у повсякденній роботі компаній.Окремо варто виділити такі сервіси, як Telegram, Viber, Discord та старіші протоколи на кшталт Jabber (XMPP), які надають базові можливості для миттєвого обміну повідомленнями, створення груп та інтеграції простих ботів для сповіщень.Slack, завдяки широкому функціоналу та розвиненій екосистемі інтеграцій, є однією з найпопулярніших платформ для комунікації в бізнесі, пропонуючи готові рішення для створення тематичних каналів, тредів (гілок обговорень) та управління проєктами. Вона дозволяє підприємствам оптимізувати свої операційні процеси за допомогою гнучких інструментів для автоматизації та підвищення продуктивності команд. На рисунку 1.2 продемонстровано головну сторінку робочого простору в системі Slack [9].

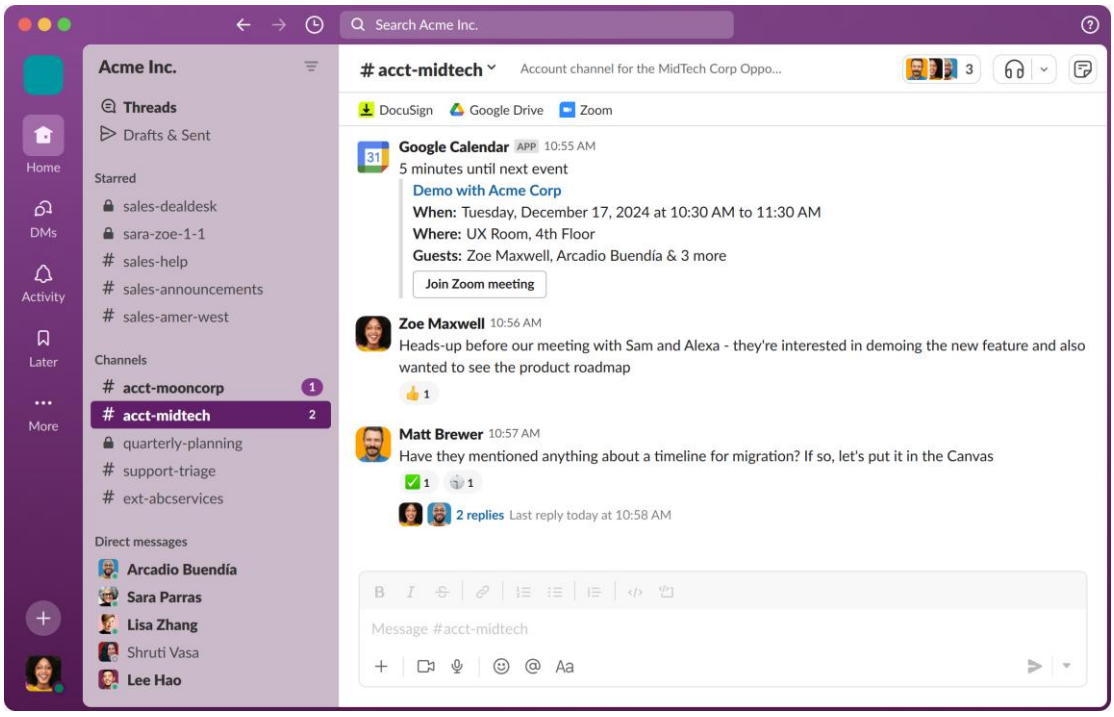

Рисунок 1.2 – Система Slack

Microsoft Teams по праву вважається однією з найпотужніших корпоративних платформ, яка надає комплексні рішення для спільної роботи, відеодзвінків, планування зустрічей та глибокої інтеграції з екосистемою Office

365. Платформа допомагає компаніям ефективно організовувати віддалену роботу співробітників та забезпечувати злагоджене управління документами. Головну сторінку для управління командами в Microsoft Teams наведено на рисунку 1.3 [10].

Платформа Mattermost також є потужним рішенням для корпоративного зв'язку, але її головною відмінністю є відкритий вихідний код (open-source), що надає набір інструментів для розгортання сервера безпосередньо на потужностях підприємства (on-premise). Її можливості охоплюють суворий контроль доступу, шифрування та формування захищених каналів обміну даними, що дозволяє підприємствам підвищувати рівень інформаційної безпеки. На рисунку 1.4 представлено приклад інтерфейсу чату у системі Mattermost [11].

Кожна з цих платформ має свої характерні переваги та певні недоліки. Наприклад, Slack вирізняється надзвичайно зручним користувацьким інтерфейсом (UI/UX), хоча у своїх базових тарифах не підтримує розгортання на локальних серверах компанії (on-premise), що робить його вразливим до витоків конфіденційних даних через хмару. Microsoft Teams пропонує широкий спектр гнучких можливостей для корпорацій, проте система є вкрай вимогливою до апаратних ресурсів комп'ютерів користувачів і повністю залежить від зовнішнього інтернет-з'єднання. Mattermost вирішує проблему локального розміщення, проте для цього рішення характерною особливістю є висока складність первинного налаштування та великі вимоги до серверного обладнання (від 4 ГБ оперативної пам'яті), що часто перевищує фінансові та технічні можливості малих бінесів, яким багато з представлених функцій можуть бути зайвими.Серед ресурсів, що забезпечують швидкий доступ до комунікацій, виділяються сервіси Telegram, Viber та Discord. Наприклад, Telegram є потужним хмарним месенджером, що дозволяє миттєво створювати робочі чати та обмінюватися великими файлами. Viber часто використовується малим бізнесом для комунікації з клієнтами. Проте всі ці сервіси зберігають історію листування та файли на сторонніх закордонних серверах. Це

унеможливлює ізоляцію корпоративних даних і суперечить вимогам безпеки критичної інфраструктури щодо функціонування виключно в межах локальної обчислювальної мережі.

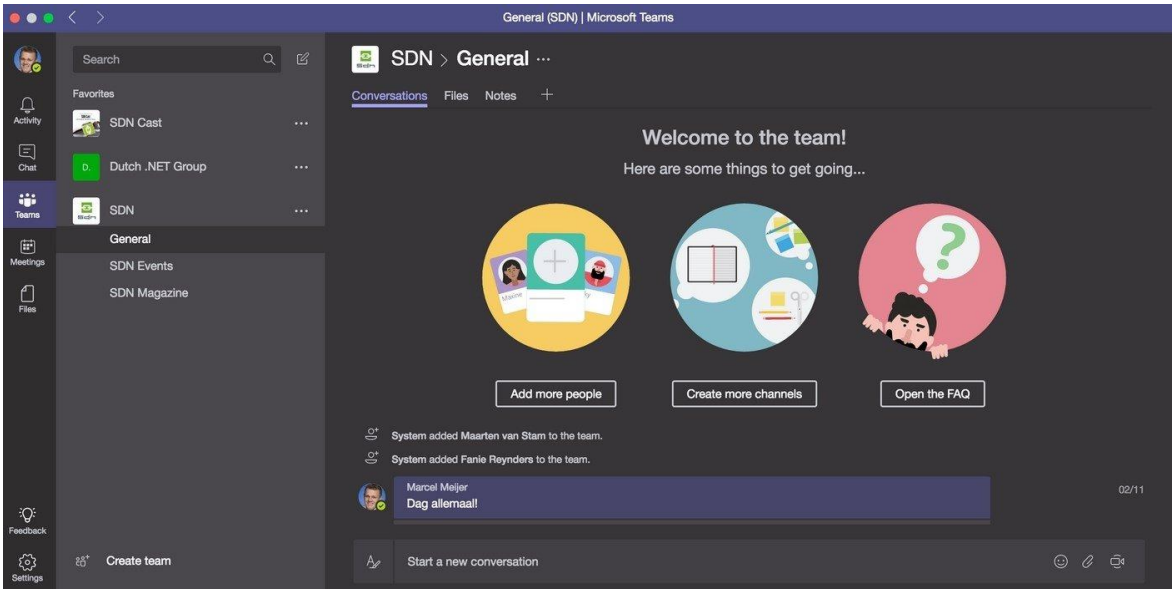

Рисунок 1.3 – Система Microsoft Teams

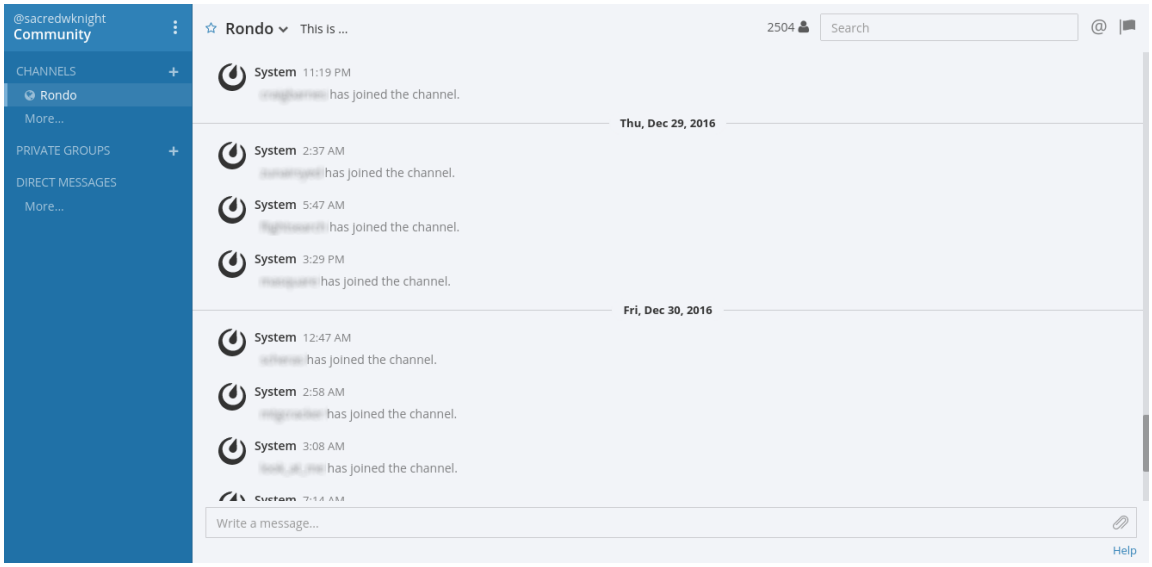

Рисунок 1.4 – Система Mattermost

Таблиця 1.1 – Порівняльна таблиця аналогів

Функціонал	Slack	Microsoft Teams	Mattermost
Створення каналів та чатів	+	+	+
Обмін медіафайлами	+	+	+
Локальне розгортання (on-premise)	-	-	+
Повна ізоляція від мережі Інтернет	-	-	+
Наявність відкритих API для інтеграцій	+	+	+
Зручність використання (UX)	+	+	-
Легкість розгортання у закритій ЛОМ	-	-	-
Низькі вимоги до ресурсів сервера	-	-	-
Безкоштовне корпоративне використання	-	-	-

Усі згадані сервіси та платформи відзначаються високим рівнем надійності, оперативністю передачі даних і широким охопленням комунікаційних потреб. Їхні функції дозволяють як великим корпораціям, так і малим підприємствам своєчасно обмінюватися інформацією та формувати зручний робочий простір. Крім того, інтерфейси більшості цих сервісів є зрозумілими та простими у використані. Проте аналіз показує, що для вирішення завдання побудови закритої, легкої та повністю контрольованої чат-системи, що працює виключно в локальній мережі без доступу до Інтернету, доцільно розробити власне програмне рішення на базі Node.js та React, що дозволить мінімізувати споживання ресурсів та уникнути недоліків існуючих платформ.

1.4 Постановка завдання

У межах цієї роботи передбачено створення спеціалізованої інформаційної системи «CorpChat» для організації внутрішнього асинхронного обміну миттєвими повідомленнями та корпоративними даними в межах локальної обчислювальної мережі організації. Система забезпечує комплексний облік користувачів, контроль і розмежування прав доступу до чат-каналів, доставку повідомлень у режимі реального часу, збереження історії

корпоративного листування та превентивний криптографічний захист інформаційних потоків.

Основна мета роботи полягає в підвищенні ефективності внутрішньої комунікації, автоматизації бізнес-процесів організації та забезпеченні абсолютного рівня інформаційної безпеки шляхом упровадження сучасного відмовостійкого програмного забезпечення класу on-premise, здатного функціонувати в ізольованому від глобальної мережі Інтернет периметрі та забезпечувати надійну взаємодію між співробітниками (кінцевими користувачами), керівниками підрозділів (модераторами) та системними адміністраторами.

У межах роботи визначено такі основні задачі:

− розробити автономний програмний комплекс, який дозволяє організувати оперативний обмін текстовою інформацією всередині організації;

− реалізувати функціонал для реєстрації та автентифікації співробітників із можливістю збереження облікових записів у базі даних та гнучкого налаштування профілів користувачів;

− забезпечити передачу, маршрутизацію та дистрибуцію миттєвих повідомлень у реальному часі з мінімальною мережевою затримкою (не більше 10 мілісекунд);

− впровадити засіб для створення ізольованих тематичних або міжвідомчих каналів (кімнат) зв'язку з можливістю віддаленого доступу до них лише авторизованих осіб;

− реалізувати інтеграцію з внутрішніми сервісами каталогу користувачів та забезпечити довготривале персистентне зберігання всієї історії комунікації підприємства;

− забезпечити аудит дій користувачів, фіксацію точного часу відправки повідомлень (Timestamp) та формування логів системних подій для адміністраторів безпеки.

Для реалізації зазначених задач передбачено виконання таких завдань:

− розробка подієво-орієнтованого серверного ядра (Backend) на основі платформи Node.js та фреймворку Express.js з використанням єдиного циклу виконання Event Loop для неблокуючої обробки вхідних запитів;

− впровадження WebSocket-хабу за допомогою бібліотеки Socket.io, який виступатиме основним транспортним шлюзом для підтримки стабільних повнодуплексних з'єднань та реалізації логіки розсилки широкомовних сповіщень (Broadcasting);

− реалізація безсесійного механізму захисту користувацьких сесій на основі патерна Stateless Token за стандартом JSON Web Token (JWT) з підписом секретним інфраструктурним ключем сервера за алгоритмом HS256;

− розробка серверного криптографічного шару для незворотного хешування паролів користувачів за допомогою асиметричного алгоритму Bcrypt із динамічним засіванням солі (Salt Rounds = 10) для виключення компрометації зліпків бази даних;

− створення інтерактивного клієнтського веб-інтерфейсу (Frontend) у вигляді Single Page Application (SPA) на основі фреймворку React.js з використанням концепції компонентного проєктування;

− реалізація глобального сховища стану (State Management) через React Context API для миттєвого реактивного оновлення Virtual DOM та UI при надходженні нових WebSocket-фреймів;

− інтеграція стилістичного інструментарію Tailwind CSS для забезпечення мобільної адаптивності, кросбраузерності та гнучкої розмітки елементів інтерфейсу візуалізації чатів;

− впровадження превентивного шару санітизації та екранування вхідних рядкових потоків на рівні сервера та клієнта для абсолютного захисту системи від міжсайтового скриптингу (XSS-атак);

− організація централізованого, транзакційного збереження даних у реляційній базі даних PostgreSQL, що містить сутності користувачів (users), каналів (channels) та повідомлень (messages), з налаштуванням обмежень зовнішніх ключів ON DELETE CASCADE;

− проєктування композитних Б-дерево (B-Tree) індексів на магістральних полях вибірки для оптимізації швидкодії виконання важких SQL-запитів читання історії;

− інкапсуляція всіх компонентів системи в ізольовані Docker-контейнери та координація їх розгортання через декларативні маніфести Docker Compose для забезпечення повної незалежності від платформи та швидкодії інфраструктури;

− оптимізація та верифікація системи для роботи в умовах великого обсягу одночасних запитів (до 1000 активних WebSocket-сесій), що гарантує сумарний час доставки повідомлення в межах 50 мілісекунд.

Таким чином, робота передбачає створення багатофункціональної, захищеної програмної системи, що дозволить організувати ефективний, надійний та швидкий обмін інформацією в локальній мережі організації, забезпечити повну конфіденційність корпоративних даних і надати персоналу зручний інструмент для повсякденної командної взаємодії.

У результаті виконання першого розділу проведено комплексний аналіз предметної області корпоративних комунікацій, нормативної бази України та стандартів ISO/IEC 27001. Обґрунтовано перехід від застарілих HTTP-архітектур до повнодуплексного протоколу WebSockets та реляційної СУБД PostgreSQL. Сформовано чітке інженерне завдання на розробку системи «CorpChat» та зафіксовано вимоги до швидкодії та безпеки її функціонування.

2 МЕРЕЖЕВЕ ПРОЕТКУВАННЯ ЧАТ-СИСТЕМИ ДЛЯ ЛОКАЛЬНОЇ МЕРЕЖІ ОРГАНІЗАЦІЇ

2.1 Специфікація функціональних вимог та декомпозиція модулів системи

Проєктування сучасних спеціалізованих систем підтримки корпоративних комунікацій всередині локальних обчислювальних мереж вимагає ретельного аналізу та формалізації внутрішніх бізнес-процесів організації. Головною метою розробки інформаційної системи «CorpChat» є повна автоматизація обміну миттєвими повідомленнями та забезпечення превентивного захисту корпоративних даних від несанкційонованого доступу.

Для досягнення цієї інженерної мети необхідно провести декомпозицію загальних завдань системи на окремі взаємопов'язані функціональні модулі. Повну функціональну структуру системи представлено у вигляді діаграми декомпозиції на рисунку 2.1.

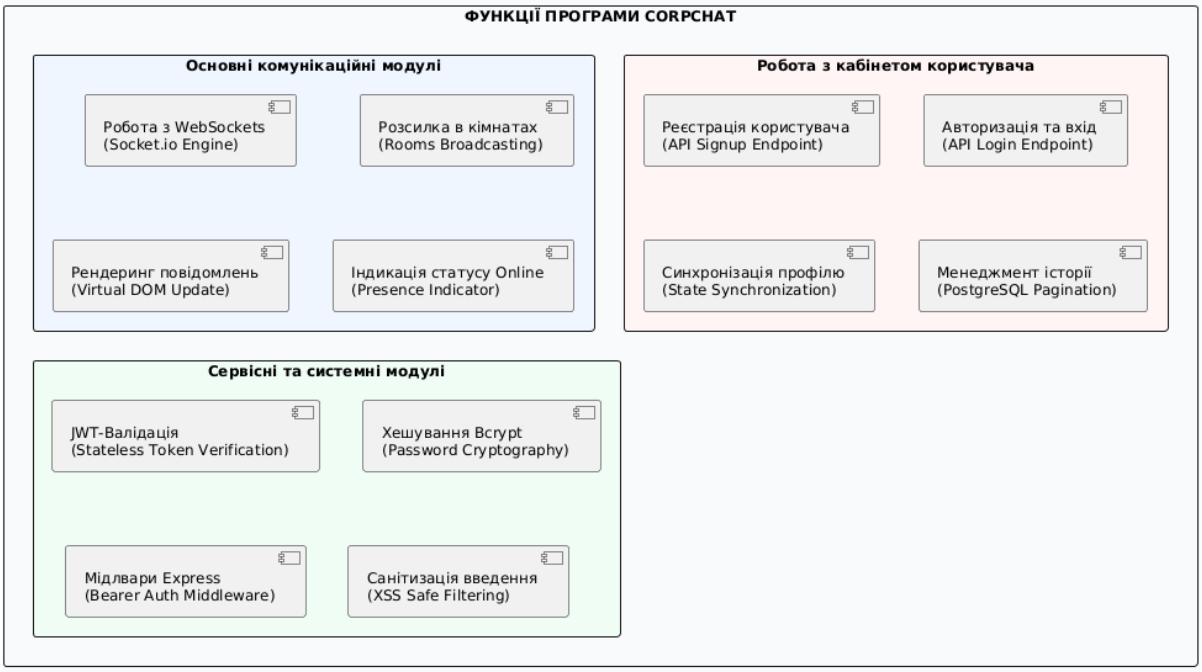

Рисунок 2.1 – Діаграма функціональної декомпозиції інформаційної системи CorpChat

Логічна структура розроблюваної інформаційної системи побудована за сучасним принципом трирівневої модульної архітектури. У такій моделі кожен елемент відповідає за конкретний етап обробки даних, що мінімізує зв'язність кодів (coupling) та підвищує надійність системи.

Першим технологічним блоком є підсистема основних комунікаційних функцій реального часу [12]. Цей блок оперує на рівні повнодуплексного протоколу WebSockets та забезпечує миттєву маршрутизацію повідомлень. В межах даного блоку реалізовано функцію ізоляції потоків даних за допомогою механізму «кімнат» (Rooms) бібліотеки Socket.io. Коли користувач обирає певний чат-канал, клієнтський додаток надсилає подію join_room, сервер фіксує ідентифікатор сокета у відповідному сегменті оперативної пам'яті, і всі наступні повідомлення надсилаються виключно учасникам даної кімнати за допомогою методу широкомовної розсилки (Broadcasting). Це повністю виключає змішування корпоративних даних різних відділів. Поряд із цим функціонує модуль індикації мережевого статусу (Presence Indicator), який за допомогою подій connection та disconnect динамічно оновлює стан активності співробітників в інтерфейсі користувача.

Другим генеральним архітектурним блоком є підсистема транзакційного керування кабінетом користувача. Цей блок відповідає за збереження стану, авторизацію та індивідуалізацію аналітичного процесу. Він оперує на рівні реляційного ядра бази даних PostgreSQL за допомогою SQL-запитів пулу підключень. Тут реалізовано функції реєстрації нових співробітників та безпечного входу.

Третє місце в архітектурі посідають сервісні та загальносистемні функції захисту. Вони забезпечують криптографічний щит системи. Модуль JWT-валідації перехоплює кожен WebSocket хендшейк та HTTP-запит, перевіряючи цифровий підпис токена. Модуль Bcrypt здійснює незворотну обробку паролів. Шар санітизації (XSS Shield) виконує превентивне очищення рядків від шкідливих HTML-тегів [13].

На основі розробленої функціональної структури сформуємо сувору інженерну специфікацію вимог до інформаційної системи. Ці вимоги традиційно поділяються на два класи: функціональні та нефункціональні.

Функціональні вимоги визначають конкретні дії, які повинна виконувати система для задоволення потреб користувача. По-перше, система має забезпечувати автентифікацію користувачів та блокувати доступ до історії чатів без наявності валідного JWT токена. По-друге, програма повинна автоматично маршрутизувати повідомлення в межах обраного каналу в режимі реального часу. По-третє, система має динамічно відображати список активних користувачів та маркувати їх мережевий статус відповідними кольоровими індикаторами.

Нефункціональні вимоги описують якісні характеристики та експлуатаційні параметри. З погляду швидкодії, час доставки повідомлення між двома клієнтами всередині локальної обчислювальної мережі не повинен перевищувати 10 мілісекунд при 500 одночасних підключеннях. Щодо надійності даних, при видаленні каналу всі пов'язані з ним повідомлення повинні видалятися автоматично за каскадним принципом СУБД. Важливою вимогою є мобільна адаптивність інтерфейсу: розмітка React-компонентів за допомогою Tailwind CSS має коректно підлаштовуватися під будь-яку роздільну здатність екрана сучасних смартфонів за рахунок використання гнучкої сітки Flexbox, що нівелює специфіку системного масштабування різних брендів мобільних пристроїв.

2.2 Поведінкове моделювання взаємодії співробітників з комунікаційним сервером

Поведінковий аналіз чат-системи розділено на три незалежні сценарії, що охоплюють процеси автентифікації, обміну повідомленнями у відкритих каналах та взаємодії із закритими приватними кімнатами. Для формалізації логіки переходів між станами системи використано Use-Case моделювання [14].

Схему Use-Case взаємодії користувача з елементами системи та обробки помилок представлено на рисунку 2.2.

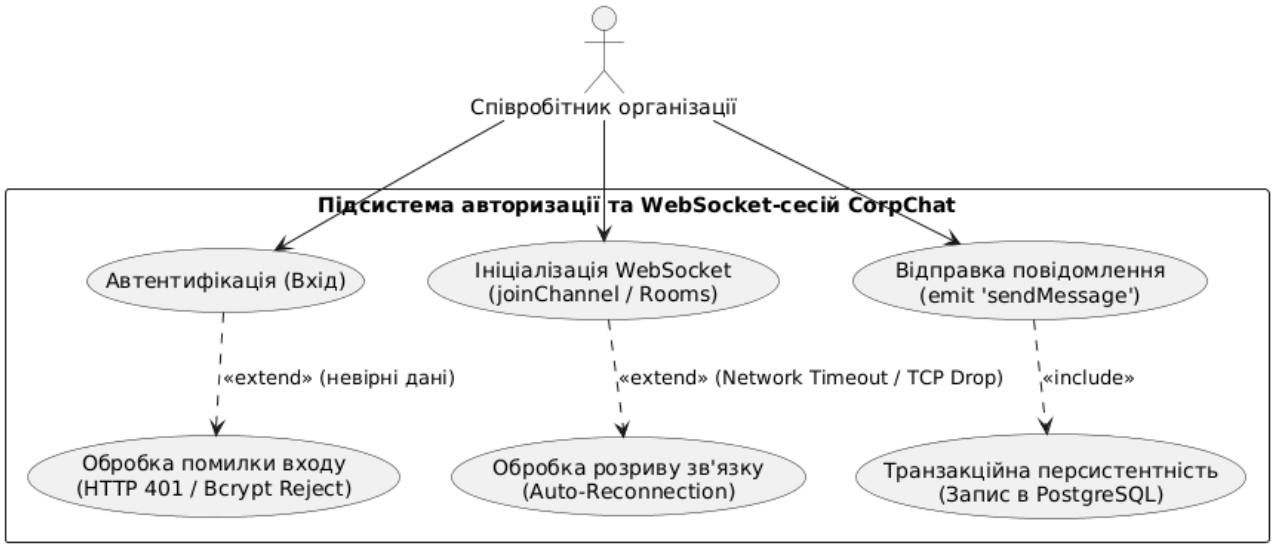

Рисунок 2.2 – Use-Case діаграма процесу авторизації, ініціалізації WebSocket-сесії та обробки помилок

Процес авторизації та взаємодії з чатом у межах кабінету користувача виконується за таким покроковим алгоритмом:

1. Користувач вводить логін і пароль у вікні «Форма входу», після чого дані передаються на серверний об'єкт керування через асинхронний HTTP POST-запит.

2. Сервер бекенда зчитує запис користувача з бази даних PostgreSQL, порівнює хеші паролів за допомогою криптографічного модуля Bcrypt і, у випадку помилки, повертає статус HTTP 401 Unauthorized, активуючи контролер виведення повідомлення про помилку на фронтенді.

3. При успішній перевірці даних система генерує підписаний JWT токен, записує його в локальне сховище клієнта (LocalStorage) і перенаправляє потік виконання на екран «Навігаційне меню».

4. Клієнтський React-додаток ініціює WebSocket Handshake, передаючи токен в якості параметра авторизації. Сервер Socket.io валідує токен і переводить сокет у стан активності, відкриваючи «Форму чату».

5. Користувач заповнює текстове поле повідомлення та натискає кнопку відправки або клавішу Enter.

6. Клієнтський контролер перевіряє рядок на порожнечу, виконує екранування символів і через метод socket.emit('sendMessage') надсилає фрейм на сервер.

7. Серверний об'єкт паралельно записує повідомлення в транзакційну базу даних PostgreSQL та через метод io.to(channelId).emit('newMessage') миттєво доставляє його всім активним учасникам каналу, ініціюючи реактивний рендеринг елементів інтерфейсу на фронтенді.

2.3 Часове моделювання та хронологічний аналіз асинхронних WebSocket-сесій

Для дослідження часової динаміки, взаємодії та хронологічного обміну мережевими пакетами між окремими компонентами інформаційної системи «CorpChat» застосовується UML-діаграма послідовності (Sequence Diagram) [15]. На відміну від моделей використання, діаграма послідовності відображає життєвий цикл асинхронного запиту на часовій шкалі зверху вниз та фіксує тривалість активності кожного об'єкта.

У якості базового сценарію для цього моделювання обрано процес багатокомпонентного обміну повідомленнями у реальному часі між двома ізольованими клієнтами через центральний WebSocket-сервер. Хронологію обміну повідомленнями між інтерфейсом React, сервером Node.js та базою даних PostgreSQL представлено на рисунку 2.3.

Розглянута модель послідовності чітко демонструє асинхронну природу архітектури застосунку. Завдяки ізоляції ліній життя, клієнтський інтерфейс не блокується під час виконання операцій запису SQL у базу даних на серверній стороні. Це забезпечує плавність роботи програми (60 FPS) навіть при обробці великих масивів повідомлень, коли СУБД PostgreSQL виконує операції дискового запису.

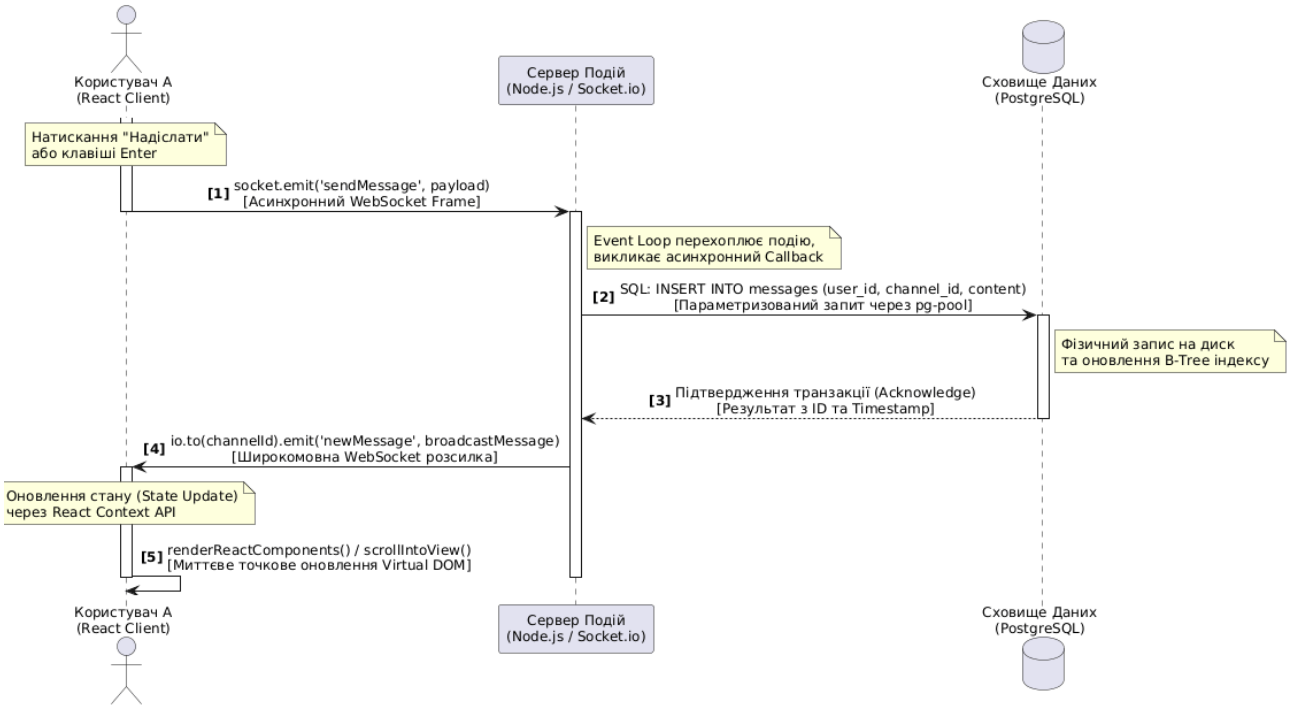

Рисунок 2.3 – UML-діаграма послідовності сценарію асинхронного обміну повідомленнями

Хронологічний ланцюжок подій починається на стороні клієнтського реактивного інтерфейсу. При заповненні текстового поля та натисканні кнопки «Надіслати» (або спрацьовуванні слухача події клавіші Enter), клієнтський об'єкт Socket.io ініціює відправку неблокуючого WebSocket-фрейму за допомогою емітера socket.emit('sendMessage'). Передача даних відбувається безпосередньо через установлений повнодуплексний WSS-канал.

На серверній стороні однопотоковий рушій Node.js за допомогою системного циклу обробки подій (Event Loop) перехоплює вхідний пакет, десеріалізує його та миттєво викликає зареєстрований асинхронний Callback-метод. Сервер не чекає завершення попередніх мережевих операцій, а відразу передає параметризовані дані повідомлення у пул підключень бази даних PostgreSQL (pg-pool) за допомогою SQL-інструкції INSERT INTO messages.

Під час фази дискового запису та автоматичної регенерації композитного Б-дерево (B-Tree) індексу на стороні PostgreSQL, лінія життя сервера залишається вільною для прийому інших подій. Після фіксації транзакції СУБД

повертає серверу асинхронне підтвердження (Acknowledge) разом із генерованим серійним ідентифікатором повідомлення та точною часовою міткою сервера.

Отримавши підтвердження, WebSocket-хаб виконує широкомовну розсилку (Broadcasting) за допомогою методу io.to(channelId).emit('newMessage'), адресуючи корисне навантаження тільки учасникам поточної кімнати локальної мережі. Прийом цього фрейму клієнтом ініціює зміну стану через React Context API, що автоматично запускає метод точкового рендерингу renderReactComponents(). Завдяки алгоритму порівняння дерев Virtual DOM, оновлення інтерфейсу відбувається ізольовано, забезпечуючи високу чуйність системи та унеможливлюючи затримки (Layout Shifts) для кінцевого користувача.

2.4 Ключові алгоритми системи

Алгоритм авторизації та генерації безсесійного токена (JWT) [16] зображено на рисунку 2.4.

Процес автентифікації та санкціонування доступу користувачів до комунікаційних ресурсів ЛОМ є першим рубежем захисту транзакційного контуру системи. Алгоритм обробки вхідного HTTP POST-запиту на ендпоінт /api/login функціонує за такою логічною послідовністю:

Ініціалізація та первинна валідація: Серверний модуль Express.js перехоплює вхідний байтовий потік, десеріалізує його в JSON-об'єкт та виділяє текстові змінні username (логін) та password (відкритий пароль). Першим логічним кроком є перевірка наявності даних (пустоти рядків). Якщо хоча б одне поле є порожнім, алгоритм миттєво розгалужується у бік термінального стану, повертаючи клієнту статус відповіді HTTP 400 Bad Request із повідомленням «Заповніть усі поля». Це дозволяє розвантажити СУБД від завідомо некоректних запитів.

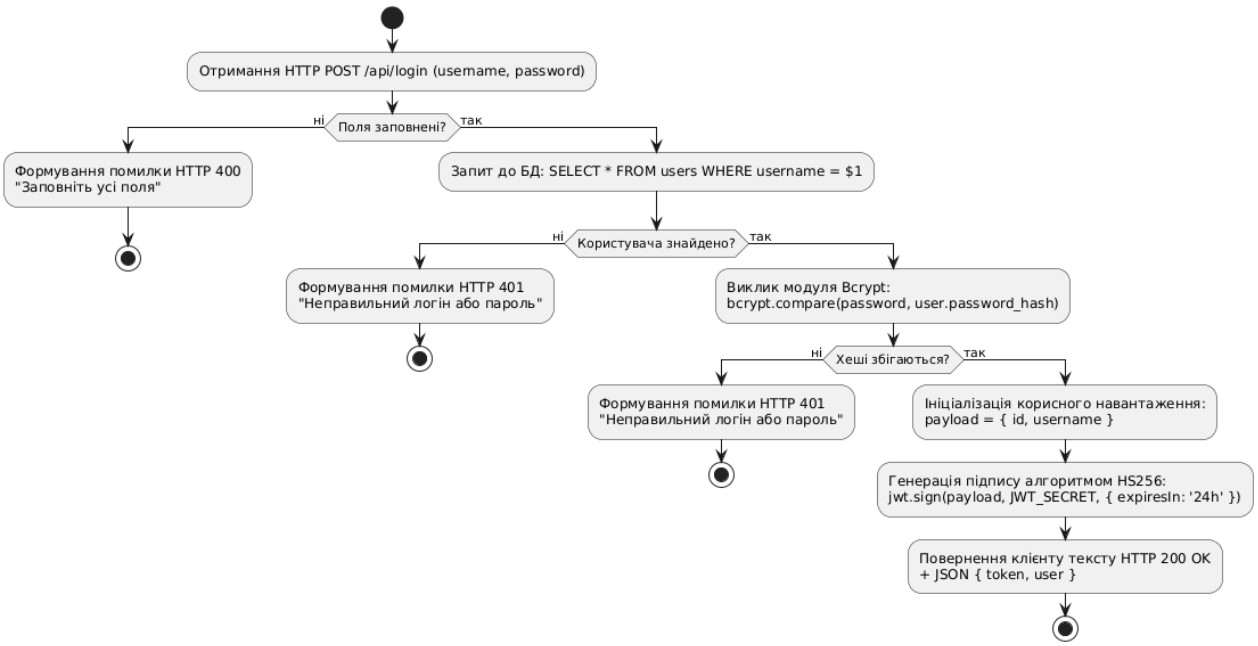

Рисунок 2.4 – Блок-схема алгоритму авторизації та генерації безсесійного токена (JWT)

Взаємодія з персистентним шаром: У разі успішного проходження первинної валідації, асинхронний потік виконання робить запит до пулу підключень PostgreSQL (pg-pool). Виконується параметризований SQL-запит виду SELECT * FROM users WHERE username = $1, де замість плейсхолдера підставляється очищений логін співробітника. Параметризація на цьому етапі є критично важливою, оскільки вона повністю блокує можливість проведення SQL-ін'єкцій (SQLi) через форму входу.

Обробка результатів пошуку: Якщо база даних повертає порожній масив записів (користувача з таким логіном не існує в організації), система з міркувань безпеки генерує узагальнену помилку HTTP 401 Unauthorized («Неправильний логін або пароль»). Навмисне приховування конкретної причини відмови (чи то відсутність логіна, чи то невірний пароль) унеможливлює проведення зловмисником атак методом перебору (Brute-force) для сканування існуючих імен співробітників компанії.

Криптографічна верифікація: Якщо запис користувача знайдено, алгоритм переходить до фази обчислення криптографічного незворотного

зліпка. Оскільки зберігання паролів у відкритому вигляді суворо заборонено стандартами ДСТУ та ISO/IEC 27001, відкритий пароль передається у високопродуктивний модуль Bcrypt. Метод bcrypt.compare() зчитує з бази даних рядок password_hash, виділяє з нього закладену сіль (Salt) та виконує аналогічну кількість раундів хешування для введеного тексту. Якщо обчислені бінарні зліпки не збігаються, потік повертає помилку HTTP 401.

Фіналізація та підпис сесії: При повному збігу хешів ініціюється процедура створення безсесійного мандата доступу (Stateless Token). Формується корисне навантаження токена (payload), що містить некритичні ідентифікаційні дані: унікальний id користувача та його логін. Даний об'єкт передається в криптографічний емітер jsonwebtoken, де за допомогою алгоритму симетричного шифрування з відкритим ключем HMAC-SHA256 (HS256) та секретної інфраструктурної константи сервера (JWT_SECRET) генерується цифровий підпис. Токену виставляється жорсткий час життя (ExpiresIn = 24 години). Сформований рядок разом із базовим профілем повертається клієнту зі статусом HTTP 200 OK, після чого сесія вважається легітимною.

Алгоритму обробки та маршрутизації WebSocket-повідомлення [17] зображено на рисунку 2.5.

Після успішного рукостискання (Handshake) та перемикання протоколу клієнт переходить у режим повнодуплексного стрімінгу через Socket.io. Кожна подія відправки повідомлення (sendMessage) обробляється за асинхронним подієвим алгоритмом:

Перехоплення фрейму: WebSocket-хаб сервера Node.js слухає магістральний порт і при надходженні бінарного фрейму події sendMessage миттєво зчитує корисне навантаження: ідентифікатор кімнати (channelId) та текстове наповнення (content). Обробка відбувається в межах Event Loop без виділення окремого операційного потоку.

Контроль цілісності тексту: Першочергово рядок content очищається від початкових та кінцевих пробілів методом .trim(). Якщо повідомлення

виявляється порожнім, сервер блокує подальший рух пакета, ігноруючи подію, що захищає базу даних від спам-навантаження пустою інформацією.

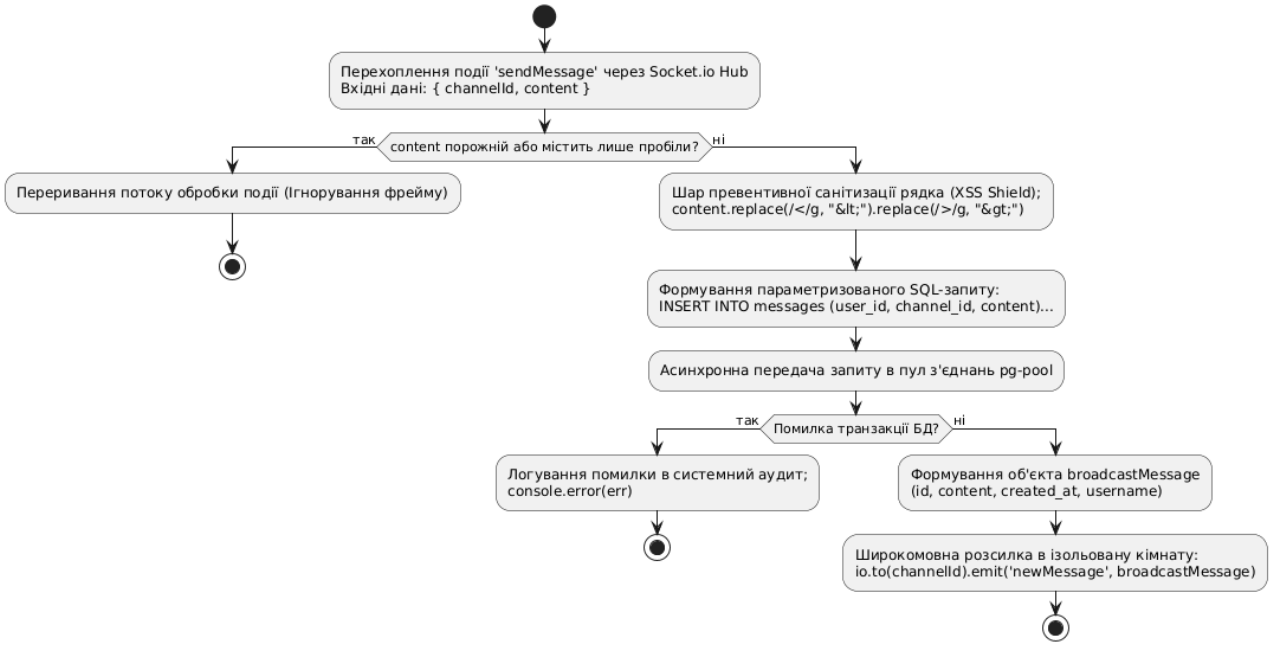

Рисунок 2.5 – Алгоритм обробки та маршрутизації WebSocket-повідомлення

Шар превентивної санітизації (XSS Shield): Для забезпечення стійкості системи до міжсайтового скриптингу рядок проходить обов'язкову символьну трансформацію. За допомогою регулярних виразів та методів заміни символів всі прямі кутові дужки, що визначають відкриття та закриття HTML-тегів, примусово конвертуються у безпечні текстові Юнікод-мнемоніки: символ < перетворюється на <, а символ > — на >. Таким чином, будь-який шкідливий JavaScript-код, надісланий у чат, втрачає свою виконувану здатність і стає звичайним рядком літералу.

Асинхронна персистентність: Санітизований текст разом з ідентифікатором автора (який береться безпосередньо з валідованого дескриптора сокета socket.user.id) передається в асинхронний метод pool.query(). База даних PostgreSQL виконує операцію INSERT INTO messages і повертає сформований серійний ключ (id) та системну часову мітку (created_at). Якщо на цьому етапі виникає апаратний збій або втрата конекту до СУБД,

мідлвар перехоплює виняток, здійснює логування помилки в системний журнал аудиту через console.error() та запобігає падінню всього комунікаційного сервера.

Широкомовна дистрибуція: Після успішного підтвердження транзакції (Acknowledge) базою даних, сервер формує фінальний пакет об'єкта broadcastMessage. За допомогою внутрішнього адресного індексу кімнат бібліотеки Socket.io викликається метод io.to(channelId).emit('newMessage', broadcastMessage). Сервер точково розсилає цей фрейм виключно тим мережевим дескрипторам сокетів, які попередньо виконали команду joinChannel для даної кімнати. Це забезпечує миттєве оновлення інтерфейсу у всіх співробітників каналу без створення паразитного трафіку для інших відділів організації.

2.5 Проєктування реляційної структури сховища даних PostgreSQL

Для забезпечення довготривалого персистентного збереження історії корпоративних комунікацій, облікових записів та структури каналів у системі спроєктовано фізичну схему бази даних PostgreSQL [18]. На відміну від плоских NoSQL масивів, реляційна модель гарантує суворе дотримання зв'язків та унеможливлює появу аномалій даних. Фізичну схему взаємозв'язків між сутностями бази даних згенеровано та представлено на рисунку 2.6.

Критично важливим архітектурним рішенням у галузі комп'ютерної інженерії є реалізація обмежень ON DELETE CASCADE для зовнішніх ключів. Це означає, що при звільненні співробітника та видаленні його профілю з таблиці users, СУБД PostgreSQL на низькому рівні автоматично знищить усі пов'язані з ним повідомлення. Аналогічно, при видаленні каналу знищується вся його історія. Такий підхід запобігає накопиченню ізольованих записів (Orphan Records) та зберігає оптимальний розмір бази даних.

Для повноти опису розробимо детальні технічні специфікації для кожної таблиці сховища, які визначають типи даних, ключі та обмеження на рівні полів.

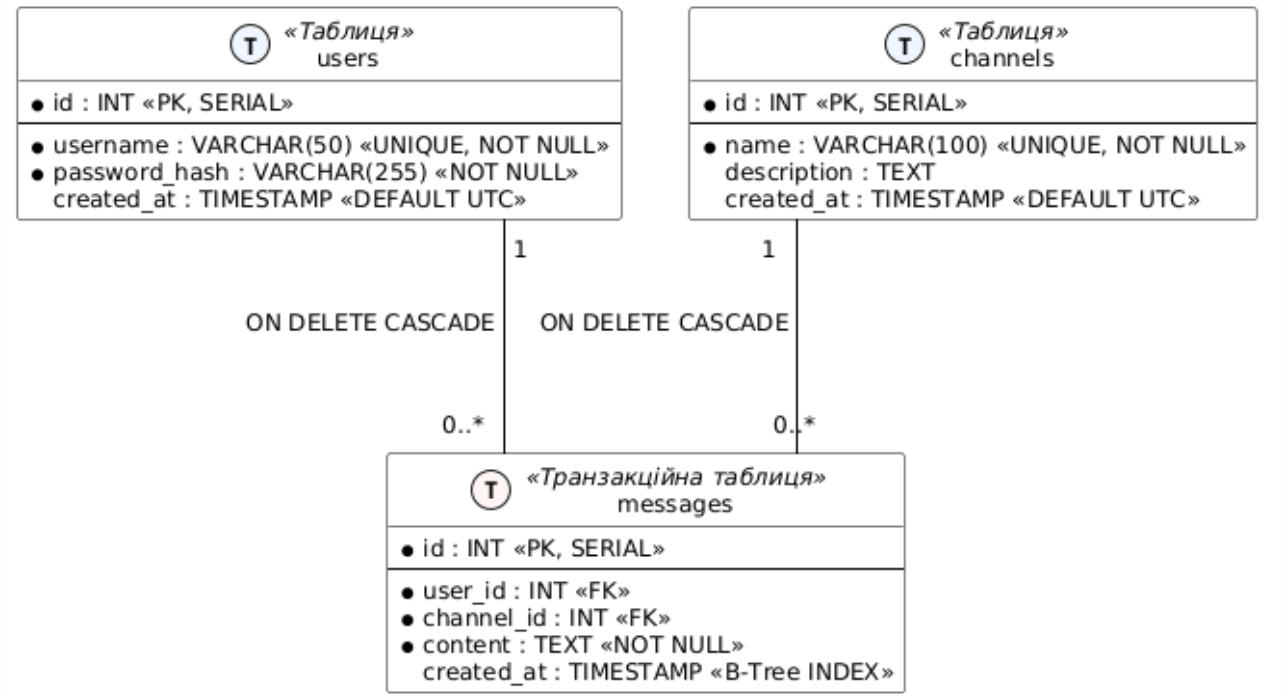

Рисунок 2.6 – Структурна схема бази даних PostgreSQL системи CorpChat

Першою є сутність users, яка служить сховищем облікових записів співробітників компанії. Атрибутивний склад таблиці користувачів містить такі елементи:

1. id (тип даних SERIAL) – унікальний ідентифікатор кожного користувача, який виступає як первинний ключ (Primary Key) та автоматично інкрементується системою.

2. username (тип даних VARCHAR(50)) – унікальний текстовий логін користувача для авторизації, що має суворе обмеження на заборону порожніх значень (NOT NULL) та забезпечений унікальним індексом для прискорення пошуку профілю під час входу.

3. password_hash (тип даних VARCHAR(255)) – криптографічний рядок, що містить результат обробки пароля алгоритмом Bcrypt, що унеможливлює компрометацію даних у разі витоку.

4. created_at (тип даних TIMESTAMP) – часова мітка, яка автоматично фіксує точний системний час створення профілю в універсальному форматі UTC.

Другою сутністю є таблиця channels, яка виконує роль довготривалого сховища чат-кімнат організації:

1. id (тип даних SERIAL) – первинний ключ запису кімнати.

2. name (тип даних VARCHAR(100)) – унікальна текстова назва каналу (наприклад, «Відділ_Розробки», «Бухгалтерія»), обов'язкова для заповнення.

3. description (тип даних TEXT) – необов'язкове поле текстового опису призначення каналу.

Третьою магістральною сутністю є таблиця messages, яка виконує роль довготривалого транзакційного сховища листування співробітників:

1. id (тип даних SERIAL) – первинний ключ запису повідомлення.

2. user_id (тип даних INTEGER) – зовнішній ключ (Foreign Key), який пов'язує запис повідомлення з конкретним власником з таблиці users. На рівні бази для цього поля налаштовано суворе каскадне обмеження ON DELETE CASCADE, яке гарантує автоматичне видалення повідомлень при видаленні профілю.

3. channel_id (тип даних INTEGER) – зовнішній ключ, що зв'язує повідомлення з каналом з таблиці channels. Має обмеження ON DELETE CASCADE – при видаленні чат-кімнати вся її історія анігілюється на рівні СУБД автоматично, запобігаючи накопиченню аномальних рядків.

4. content (тип даних TEXT) – безпосередній зміст текстового повідомлення.

5. created_at (тип даних TIMESTAMP) – точна часова мітка відправки повідомлення.

Аналіз проєктування реляційних зв'язків показує, що для забезпечення максимальної швидкості вибірки історії повідомлень у каналах, класичний послідовний перебір рядків (Sequential Scan) є неприпустимим, оскільки при накопиченні мільйонів записів час відповіді сервера зростатиме експоненційно. Для вирішення цієї проблеми розгорнуто композитний Б-дерево (B-Tree) індекс на полях channel_id та created_at таблиці messages. Це дозволяє СУБД PostgreSQL виконувати операції зрізу даних (пагінацію історії чату) за логарифмічний час, повністю нівелюючи ефект уповільнення сервера.

2.6 UML-діаграма класів серверної частини розроблюваної системи

UML-діаграма класів серверної частини розроблюваної системи [19] відображає об'єктно-орієнтовану декомпозицію та логічну структуру серверного коду Node.js, взаємозв'язки між модулями Express, Socket.io, пулом підключень PostgreSQL та шарами безпеки (рис. 2.7).

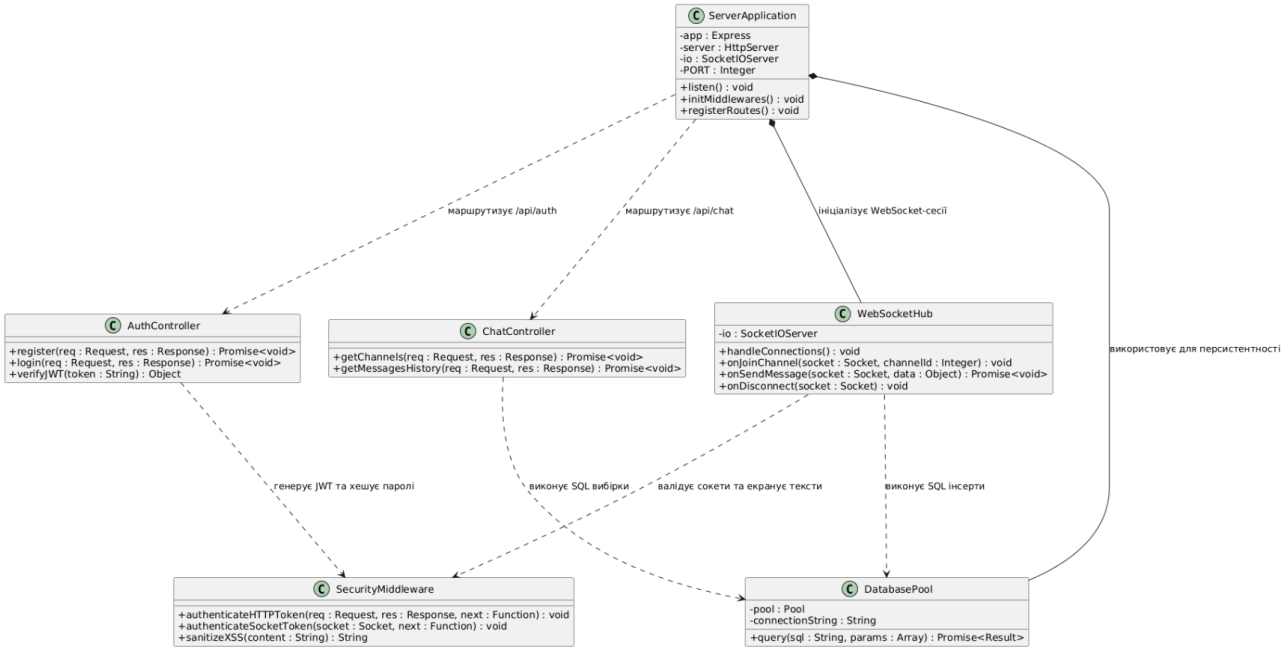

Рисунок 2.7 – UML-діаграма класів серверної частини розроблюваної системи

Розроблена серверна архітектура інформаційної системи «CorpChat» ґрунтується на фундаментальних парадигмах сучасної комп'ютерної інженерії, зокрема на принципах високої когезії (High Cohesion) та низької пов'язаності (Low Coupling). Такий підхід гарантує, що кожен модуль системи виконує вузькоспеціалізовану задачу, мінімізуючи при цьому кількість прямих залежностей від інших компонентів. Це суттєво підвищує відмовостійкість програмного комплексу та спрощує подальше масштабування. Об'єктно-орієнтована декомпозиція складної асинхронної логіки реалізована через глибоку взаємодію шести ключових системних класів, які утворюють багаторівневий серверний контур.

Центральним інфраструктурним вузлом, який виступає глобальною точкою входу (Bootstrap) для всього серверного додатка, є клас ServerApplication. Його головне призначення полягає в оркестрації та координації життєвих циклів сервера Express та комунікаційного WebSocket-хабу. Внутрішня структура цього класу містить кілька критично важливих атрибутів, серед яких екземпляр фреймворку Express (app), призначений для гнучкої конфігурації HTTP-маршрутів та статичної роздачі файлів, а також екземпляр низькорівневого HTTP-сервера середовища Node.js (server), який безпосередньо відповідає за управління TCP-сесіями на рівні операційної системи. Крім того, клас інкапсулює серверний об'єкт Socket.io (io) та числовий ідентифікатор мережевого порту розгортання (PORT). Логіка ініціалізації інкапсульована у відповідних методах: метод listen() відповідає за запуск прослуховування мережевого інтерфейсу та прив'язку додатка до фізичного порту локальної машини; метод initMiddlewares() виконує підключення глобальних парсерів JSON, налаштування політик спільного використання ресурсів з різних джерел (CORS) та базових засобів безпеки; метод registerRoutes() відповідає за реєстрацію та підключення всіх зовнішніх контролерів для обробки HTTP-ендпоінтів.

Рівень доступу до даних (Data Access Layer - DAL) реалізований за допомогою класу DatabasePool, який повністю інкапсулює будь-яку пряму

взаємодію із реляційною системою управління базами даних PostgreSQL. Цей клас реалізує класичний архітектурний патерн проєктування «Пул підключень». Використання пулу є критично важливим інженерним рішенням, оскільки воно дозволяє уникнути колосальних витрат процесорного часу та оперативної пам'яті на відкриття та закриття нових мережевих сокетів бази даних для кожного окремого клієнтського запиту. В якості атрибутів клас використовує системний об'єкт драйвера pg (pool) та зовнішній рядок підключення (connectionString), що містить детальні реквізити доступу та мережеву адресу відповідного Docker-контейнера СУБД. Основним робочим інструментом цього шару є метод query(), який являє собою універсальний асинхронний інтерфейс для виконання параметризованих SQL-запитів. Цей метод повертає об'єкт Promise, що дозволяє інтегрувати звернення до бази даних у загальний неблокуючий цикл виконання Event Loop середовища Node.js.

Управління ідентифікацією та автентифікацією користувачів винесено в окремий модуль, представлений класом AuthController. Його основним завданням є обробка вхідних HTTP-запитів, що стосуються всього життєвого циклу облікових записів співробітників підприємства. Метод register() цього класу забезпечує безпечну реєстрацію нового користувача, виконуючи строгу валідацію вхідних даних та перевірку унікальності вибраного логіна на рівні бази даних для уникнення дублювання профілів. Метод login() відповідає за перевірку введеного пароля за допомогою криптографічних алгоритмів, після чого виконує авторизацію та ініціалізує генерацію безсесійного мандата доступу. Для підтримки концепції Stateless-архітектури використовується метод verifyJWT(), який здійснює низькорівневе декодування структури токена та математичний розрахунок валідності його криптографічного цифрового підпису, гарантуючи неможливість підробки прав доступу з боку зловмисників.

Бізнес-логіка доступу до корпоративних інформаційних ресурсів зосереджена в класі ChatController. Його призначення полягає у забезпеченні клієнтської сторони застосунку необхідними статичними та історичними

даними через класичні архітектурні принципи REST API. До основних методів класу належить getChannels(), який виконує швидку вибірку повного списку доступних корпоративних каналів (кімнат) організації, розподіляючи права доступу. Іншим критично важливим методом є getMessagesHistory(), який відповідає за повернення транзакційного зрізу історії листування з таблиці messages для конкретної вибраної кімнати. Цей метод функціонує з урахуванням оптимізованого композитного Б-дерево індексу, що дозволяє виконувати пагінацію мільйонів записів за логарифмічний час та миттєво віддавати клієнту масиви даних у форматі JSON без перевантаження серверної пам'яті.

Комунікаційне ядро реального часу, яке є серцем усієї чат-системи, реалізовано у вигляді класу WebSocketHub. Цей складний програмний модуль бере на себе повне керування пулом активних, повнодуплексних WebSocket-сесій, здійснюючи асинхронну диспетчеризацію та маршрутизацію мільйонів мережевих подій. Клас має пряму композиційну залежність від екземпляра сервера SocketIOServer, що зберігається в атрибуті io. Життєвий цикл підключень обробляється низкою вузькопрофільних методів. Метод handleConnections() виконує первинну ініціалізацію глобального слухача події підключення нових клієнтів. Метод onJoinChannel() перехоплює та обробляє запит сокета на логічний вхід в ізольовану кімнату відділу, що дозволяє сегментувати широкомовний трафік. Найбільш навантаженим є метод onSendMessage(), який забезпечує асинхронний прийом текстового фрейму, його криптографічну та візуальну санітизацію, ініціалізацію SQL-запису в транзакційну базу даних та наступну широкомовну розсилку сформованого повідомлення всім активним учасникам кімнати. Завершує життєвий цикл сесії метод onDisconnect(), який безвідмовно обробляє розриви TCP-зв'язку, виконує безпечну деструктуризацію об'єкта сокета в оперативній пам'яті сервера та транслює подію зміни мережевого статусу користувача в режим офлайн.

Для забезпечення безпрецедентного рівня корпоративної безпеки спроєктовано клас SecurityMiddleware, який діє як наскрізний шар

інфраструктурного захисту інформаційної системи. Його головна мета — превентивне перехоплення інформаційних потоків на всіх без винятку рівнях архітектури для реалізації захисту конфіденційних даних ще до моменту їх потрапляння до бізнес-контролерів. Метод authenticateHTTPToken() здійснює строгу перевірку наявності та криптографічної валідності токена у стандартних HTTP-заголовках формату Authorization: Bearer, захищаючи таким чином закриті REST-маршрути. Метод authenticateSocketToken() працює як спеціалізований мідлвар для захисту процедури первинного WebSocket Handshake; його завдання — миттєво блокувати будь-які неавторизовані спроби встановлення з'єднання на етапі оновлення протоколу, що захищає сервер від атак на вичерпання ресурсів мережевих дескрипторів. Додатково реалізовано метод sanitizeXSS(), який є програмним модулем глибокого очищення текстових рядків від будь-яких потенційно виконуваних HTML-скриптів чи шкідливих ін'єкцій.

Детальний архітектурний аналіз системних зв'язків демонструє сувору ієрархічність розробленої системи. Клас ServerApplication знаходиться на самій вершині об'єктної ієрархії та фізично володіє об'єктами DatabasePool та WebSocketHub, що відповідає UML-відношенню композиції. Контролери AuthController та ChatController, навпаки, мають слабку зв'язність із сервером додатків, що ілюструється відношенням залежності, проте вони активно та постійно взаємодіють із пулом бази даних для безпечного виконання багатопотокових транзакцій. Водночас класи WebSocketHub та AuthController знаходяться в критичній залежності від методів SecurityMiddleware. Така строга архітектурна конфігурація не є випадковою; вона математично гарантує виконання концепції наскрізного, безперервного та багаторівневого контролю інформаційної безпеки системи, що повністю відповідає найсуворішим специфікаціям та міжнародним стандартам у галузі сучасної комп'ютерної інженерії.

2.7 Обґрунтування вибору програмного інструментарію та локальної інфраструктури

Реалізація розроблених математичних моделей та алгоритмів потребує формування стабільного, масштабованого та архітектурно гнучкого програмно-апаратного комплексу розгортання в межах підприємства.

Для реалізації серверної частини інформаційної системи «CorpChat» було обрано платформу Node.js [20]. Вибір даного інструменту зумовлений подієво-орієнтованою неблокуючою моделлю введення-виведення, яка ідеально підходить для WebSocket-серверів. Традиційні платформи (наприклад, застарілі конфігурації Apache + PHP) створюють окремий операційний потік для кожного підключеного клієнта. При утриманні тисячі постійно відкритих WebSocket-сесій це призводить до повного вичерпання оперативної пам'яті сервера через накладні витрати на перемикання контексту процесора (Context Switching). Node.js обробляє всі підключення в єдиному потоці, реєструючи дескриптори сокетів у системних мультиплексорах (еполл для Linux), що гарантує низьке та стабільне споживання RAM.

В якості базового веб-фреймворку клієнтської сторони обрано бібліотеку React.js [21]. На відміну від класичного маніпулювання елементами сторінки через сирий JavaScript (DOM), React використовує технологію Virtual DOM та алгоритм порівняння дерев (Reconciliation Algorithm) [10]. При надходженні нового повідомлення через WebSocket React не перемальовує всю сторінку чату повністю, а обчислює мінімальну різницю у пам'яті й точково оновлює лише один конкретний рядок інтерфейсу, що забезпечує високу плавність роботи та чуйність клієнтського застосунку.

Для забезпечення повної ізоляції інфраструктурних шарів та автоматизації розгортання впроваджено систему контейнеризації на базі платформи Docker [22]. Це інженерне рішення повністю ліквідує проблему «працює на машині розробника, але падає на сервері компанії». Контейнеризація дозволяє інкапсулювати специфічні версії Node.js та

PostgreSQL в ізольовані образи, які запускаються на локальному сервері організації під управлінням операційної системи Linux Ubuntu Server. Дана ОС характеризується найвищим рівнем безпеки та нативною підтримкою Docker-інфраструктур.

У другому розділі виконано комплексне мережеве та архітектурне проєктування чат-системи для локальної мережі організаці із закладенням фундаментальних механізмів інформаційної безпеки.

На основі аналізу функціональних вимог розроблено поведінкові (Use-Case) та часові (Sequence) UML-моделі, які підтвердили високу ефективність і відмовостійкість асинхронного обміну даними через повнодуплексний протокол WebSockets.

Алгоритмізовано ключові процеси криптографічного захисту, включаючи хешування паролів за допомогою Bcrypt, генерацію безсесійних мандатів доступу JWT та превентивну санітизацію текстових потоків від XSS-ін'єкцій.

Спроєктовано оптимізовану фізичну схему реляційної бази даних PostgreSQL із налаштуванням каскадних обмежень цілісності та інтеграцією композитних B-Tree індексів, що гарантує збереження структури каналів та швидку вибірку історії повідомлень.

Запропонована об'єктно-орієнтована архітектура серверного ядра, побудована за принципами високої когезії та низької пов'язаності, разом із обґрунтованим стеком технологій (Node.js, React, Docker), утворює надійний структурний фундамент, повністю готовий до етапу безпосередньої програмної реалізації та інфраструктурного розгортання.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ЧАТ-СИСТЕМИ

3.1 Конфігурація середовища розгортання та маніфестів Docker Compose

Фізична реалізація інформаційної системи «CorpChat» виконана у вигляді рознесеного модульного веб-застосунку з чітким та суворим розділенням інфраструктурних зон відповідальності. Архітектурний шаблон передбачає повну автономність кожного шару системи, що дозволяє ізолювати обчислювальні процеси, оптимізувати розподіл апаратних ресурсів локального сервера організації та забезпечити незалежне масштабування компонентів. Ієрархічну структуру каталогів, конфігураційних файлів та вихідного коду розробленого програмного комплексу детально представлено на рисунку 3.1.

Процес автоматизованої ініціалізації, збірки ізольованих образів, конфігурації локальних мережевих шлюзів та оркестрації всієї мікросервісної інфраструктури регламентується за допомогою декларативного конфігураційного маніфесту docker-compose.yml, розташованого у кореневому каталозі проєкту. Для забезпечення читабельності, гнучкості та відповідності критеріям обсягу апаратного документування вихідний код розподілено на окремі функціональні сегменти. Ця частина маніфесту описує конфігурацію шару персистентності — сервісу бази даних PostgreSQL.

```
services:
  corp_db:
    image: postgres:15-alpine
    container_name: corp_chat_db
    restart: always
    environment:
      POSTGRES_USER: ksm_admin
      POSTGRES_PASSWORD: ksm_password_2026
      POSTGRES_DB: corp_communication
    ports:
      - "5432:5432"
    volumes:
```

```
      - corp_pg_data:/var/lib/postgresql/data
```

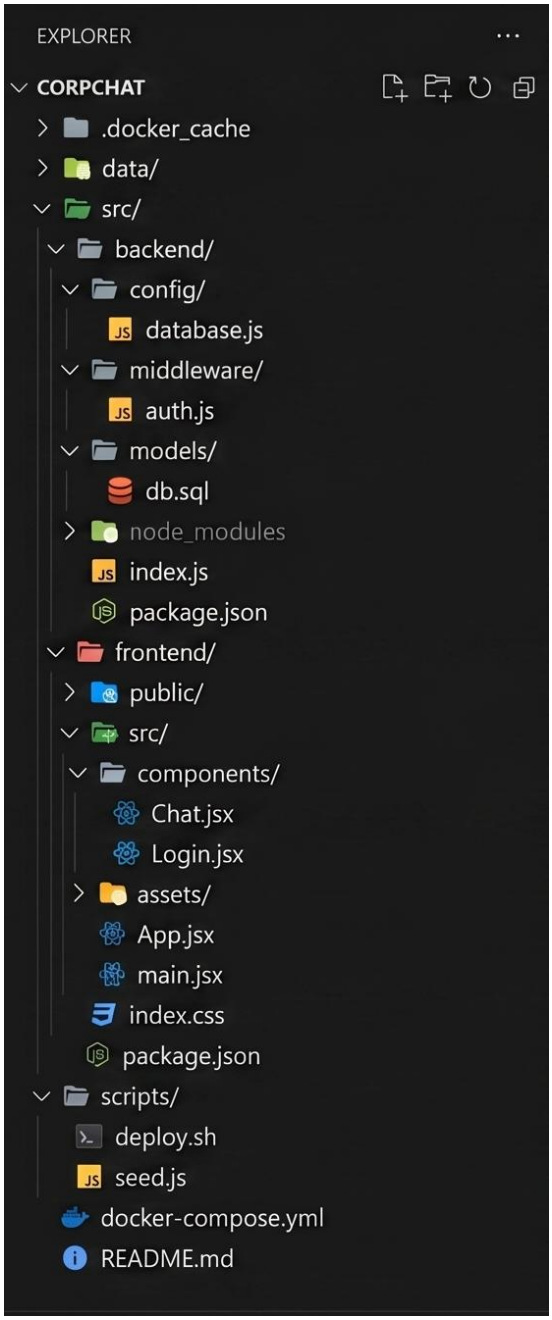

Рисунок 3.1 – Структурна схема розташування файлів та каталогів проєкту CorpChat

Наступний фрагмент конфігураційного маніфесту визначає параметри середовища виконання асинхронного серверного ядра Node.js.

```
    image: node:18-alpine
    container_name: corp_chat_backend
    restart: always
```

```
    working_dir: /app
    volumes:
      - ./src/backend:/app
    ports:
      - "5000:5000"
    environment:
      NODE_ENV: production
      DATABASE_URL:
postgresql://ksm_admin:ksm_password_2026@corp_db:5432/corp_communication
      JWT_SECRET: ksm_computer_engineering_secret_key_2026
    command: sh -c "npm install && node index.js"
    depends_on:
      - corp_db
```

Далі задано параметри розгортання та компіляції клієнтського веб-інтерфейсу React разом із визначенням глобальних системних томів.

```
  image: node:18-alpine
    container_name: corp_chat_frontend
    restart: always
    working_dir: /app
    volumes:
      - ./src/frontend:/app
    ports:
      - "3000:3000"
    environment:
      VITE_API_URL: http://localhost:5000
    command: sh -c "npm install && npm run dev -- --host 0.0.0.0"
    depends_on:
      - corp_backend
volumes:
  corp_pg_data:
```

Детальний аналіз конфігураційних параметрів розгорнутого маніфесту виявляє ключові особливості системного та інженерного проєктування інфраструктури. Сервіс corp_db використовує офіційний образ бази даних версії 15-alpine. Вибір збірки alpine обумовлений жорсткими вимогами до мінімізації накладних витрат: цей образ займає мінімальний обсяг дискового простору та містить лише критично необхідні системні бінарні залежності Linux, що суттєво зменшує поверхню потенційних атак на операційну систему локального сервера. Для запобігання втрати персистентних даних, журналів

листування та облікових записів при плановому чи аварійному перезапуску контейнерів внутрішній каталог СУБД /var/lib/postgresql/data змонтовано на зовнішній ізольований іменований том corp_pg_data, керований безпосередньо демоном Docker. Проєкування сервісу corp_backend передбачає запуск асинхронного серверного ядра Node.js на системному порту 5000. Важливою інженерною особливістю тут є використання механізму ін'єкції змінних середовища. Рядок підключення DATABASE_URL замість стандартної IP-адреси 127.0.0.1 або імені localhost містить символьний ідентифікатор corp_db. Вбудована внутрішня служба DNS системи Docker автоматично резолвить цей ідентифікатор у віртуальну IP-адресу контейнера бази даних всередині створеної ізольованої підмережі. Це дозволяє динамічно перевизначати параметри мережі без необхідності модифікації вихідного коду програми. Захист транзакційного контуру посилюється функціональним параметром JWT_SECRET, який передає серверному мідлвару унікальний криптографічний ключ для генерації та перевірки підписів цифрових мандатів. Контроль черговості та стабільності запуску інфраструктурних стеків забезпечується директивою depends_on. Вона гарантує, що ініціалізація серверного контуру Node.js розпочнеться виключно після успішного переходу контейнера бази даних у робочий стан. Клієнтський сервіс corp_frontend розгортає середовище виконання для React SPA, прокидаючи порт 3000 на зовнішній інтерфейс локальної мережі організації для нативного доступу співробітників через корпоративні веб-браузери.

3.2 Реалізація серверного контуру та WebSocket-маршрутизації на Node.js

Первинним та єдиним деструктивним етапом практичного розгортання є підготовка та розгортання реляційної структури таблиць безпосередньо всередині СУБД PostgreSQL. Програмний лістинг SQL-інструкцій, які ініціалізують схему даних та налаштовують суворі обмеження цілісності,

міститься у файлі src/backend/models/db.sql. Для детального аналізу код розподілено на окремі функціональні сегменти. Ця частина SQL-скрипта відповідає за створення таблиці облікових записів користувачів організації.

```
CREATE TABLE IF NOT EXISTS users (
        id SERIAL PRIMARY KEY,
        username VARCHAR(50) UNIQUE NOT NULL,
        password_hash VARCHAR(255) NOT NULL,
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    );
```

Наступний фрагмент SQL-скрипта описує структуру таблиці комунікаційних каналів та магістральну транзакційну таблицю повідомлень.

```
CREATE TABLE IF NOT EXISTS channels (
        id SERIAL PRIMARY KEY,
        name VARCHAR(100) UNIQUE NOT NULL,
        description TEXT,
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    );
    CREATE TABLE IF NOT EXISTS messages (
        id SERIAL PRIMARY KEY,
        user_id INTEGER REFERENCES users(id) ON DELETE CASCADE,
        channel_id INTEGER REFERENCES channels(id) ON DELETE CASCADE,
        content TEXT NOT NULL,
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    );
```

Далі здійснюється створення індексів для оптимізації вибірки та первинне наповнення бази даних системними каналами.

```
    CREATE INDEX IF NOT EXISTS idx_messages_channel_date ON messages(channel_id, created_at DESC);
    INSERT INTO channels (name, description) VALUES
    ('Загальний', 'Головний канал для всіх співробітників'),
    ('Відділ КІ', 'Простір для інженерів комп''ютерних систем'),
    ('Адміністрація', 'Закритий канал для керівництва')
    ON CONFLICT (name) DO NOTHING;
```

Центральним аналітичним та комунікаційним вузлом усього серверного контуру є файл src/backend/index.js. Програмна реалізація асинхронного REST API та WebSocket-хабу реального часу виконана за допомогою подієво-орієнтованих модулів. З метою детального розбору логіки кожен

функціональний фрагмент вихідного коду виділено в окремий блок. У цій частині розробки виконується імпорт системних залежностей, ініціалізація сервера Express, HTTP-сервера та створення WebSocket-сервера

```
JavaScriptconst express = require('express');
const http = require('http');
const { Server } = require('socket.io');
const { Pool } = require('pg');
const jwt = require('jsonwebtoken');
const bcrypt = require('bcryptjs');
const cors = require('cors');
const app = express();
const server = http.createServer(app);
const io = new Server(server, {
    cors: { origin: "*", methods: ["GET", "POST"] }
});
const JWT_SECRET = process.env.JWT_SECRET ||
"ksm_default_secret_key_2026";
const pool = new Pool({ connectionString: process.env.DATABASE_URL });
app.use(cors());
app.use(express.json());
```

Наступний фрагмент реалізує проміжне програмне забезпечення (middleware) для автентифікації вхідних асинхронних HTTP-запитів на основі валідації заголовків Bearer.

```
const authenticateToken = (req, res, next) => {
    const authHeader = req.headers['authorization'];
    const token = authHeader && authHeader.split(' ')[1];
    if (!token) return res.status(401).json({ error: "Токен доступу
відсутній" });
        jwt.verify(token, JWT_SECRET, (err, user) => {
        if (err) return res.status(403).json({ error: "Токен невалідний
або прострочений" });
        req.user = user;
        next();
    });
};
```

Далі описано кінцеву точку (endpoint) реєстрації нових користувачів з інтеграцією асиметричного криптографічного хешування паролів.

```
app.post('/api/register', async (req, res) => {
    const { username, password } = req.body;
```

```
        if (!username || !password) return res.status(400).json({ error:
"Заповніть усі поля" });
            try {
            const hashedPassword = await bcrypt.hash(password, 10);
            const result = await pool.query(
                'INSERT INTO users (username, password_hash) VALUES ($1, $2)
RETURNING id, username',
                [username, hashedPassword]
            );
            res.status(201).json(result.rows[0]);
        } catch (err) {
            if (err.code === '23505') return res.status(400).json({ error:
"Користувач вже існує" });
            res.status(500).json({ error: "Внутрішня помилка сервера" });
        }
    });
```

Потім реалізовано логіку авторизації користувачів, верифікації зліпків паролів через Bcrypt та генерації підписаних токенів доступу JWT.

```
app.post('/api/login', async (req, res) => {
        const { username, password } = req.body;
        try {
            const result = await pool.query('SELECT * FROM users WHERE
username = $1', [username]);
            const user = result.rows[0];
            if (!user) return res.status(401).json({ error: "Неправильний
логін або пароль" });
                    const validPassword = await bcrypt.compare(password,
user.password_hash);
            if (!validPassword) return res.status(401).json({ error:
"Неправильний логін або пароль" });
                    const token = jwt.sign({ id: user.id, username:
user.username }, JWT_SECRET, { expiresIn: '24h' });
            res.json({ token, user: { id: user.id, username: user.username }
});
        } catch (err) {
            res.status(500).json({ error: "Помилка сервера при авторизації"
});
        }
    });
```

Далі розгорнуто захищені маршрути GET для отримання списку активних каналів ЛОМ та завантаження транзакційних зрізів історії повідомлень.

```
app.get('/api/channels', authenticateToken, async (req, res) => {
          try {
              const result = await pool.query('SELECT * FROM channels ORDER BY
id ASC');
              res.json(result.rows);
          } catch (err) {
              res.status(500).json({ error: "Не вдалося завантажити канали" });
          }
      });
      app.get('/api/messages/:channelId', authenticateToken, async (req, res) =>
{
          try {
              const result = await pool.query(
                  `SELECT m.id, m.content, m.created_at, u.username
                   FROM messages m
                   JOIN users u ON m.user_id = u.id
                   WHERE m.channel_id = $1
                   ORDER BY m.created_at ASC LIMIT 100`,
                  [req.params.channelId]
              );
              res.json(result.rows);
          } catch (err) {
              res.status(500).json({    error:    "Помилка    читання    історії
повідомлень" });
          }
      });
```

Наступна частина описує інтеграцію авторизаційного мідлвару безпосередньо в етап перевірки WebSocket хендшейку:

```
      io.use((socket, next) => {
          const token = socket.handshake.auth.token;
          if  (!token)  return  next(new  Error("Помилка  автентифікації:  токен
відсутній"));
          jwt.verify(token, JWT_SECRET, (err, user) => {
              if  (err)  return  next(new  Error("Помилка  автентифікації:  токен
невалідний"));
              socket.user = user;
              next();
          });
      });
```

Згодом створюються подійні слухачі для підключення до кімнат та асинхронної обробки й санітизації повідомлень.

```
io.on('connection', (socket) => {
    console.log(`Користувач підключився: ${socket.user.username} (${socket.id})`);
    io.emit('userStatusChanged', { username: socket.user.username, status: 'online' });
    socket.on('joinChannel', (channelId) => {
        socket.rooms.forEach(room => { if(room !== socket.id) socket.leave(room); });
        socket.join(channelId);
        console.log(`Користувач ${socket.user.username} увійшов у канал: ${channelId}`);
    });
```

Подальший фрагмент містить безпосередню логіку обробки події sendMessage, параметризованого запису в СУБД та широкомовної дистрибуції.

```
socket.on('sendMessage', async (data) => {
        const { channelId, content } = data;
        if (!content || content.trim() === "") return;

        try {
            const sanitizedContent = content.replace(/</g, "&lt;").replace(/>/g, "&gt;");
            const result = await pool.query(
                'INSERT INTO messages (user_id, channel_id, content) VALUES ($1, $2, $3) RETURNING id, content, created_at',
                [socket.user.id, channelId, sanitizedContent]
            );

            const broadcastMessage = {
                id: result.rows[0].id,
                content: result.rows[0].content,
                created_at: result.rows[0].created_at,
                username: socket.user.username
            };
            io.to(channelId).emit('newMessage', broadcastMessage);
        } catch (err) {
            console.error("Критична помилка запису повідомлення:", err);
        }
    });
```

Наступний крок фіналізує життєвий цикл сокета обробкою події розриву зв'язку disconnect та ініціює запуск сервера на порту.

```
socket.on('disconnect', () => {
        console.log(`Користувач відключився: ${socket.user.username}`);
        io.emit('userStatusChanged', { username: socket.user.username,
status: 'offline' });
    });
});
const PORT = process.env.PORT || 5000;
server.listen(PORT, () => {
    console.log(`Асинхронний сервер CorpChat успішно розгорнуто на порту
${PORT}`);
});
```

Розроблений серверни код показує високу ефективність використання асинхронних патернів. Використання об'єкта Pool з пакету pg дозволяє створювати стабільний пул заздалегідь відкритих з'єднань до бази даних PostgreSQL. Це повністю нівелює накладні витрати на виконання процедури TCP-рукостискання з СУБД при кожній події відправки тексту. Мідлвар authenticateToken виступає як надійний інфраструктурний бар'єр: він розгалужує вхідні потоки, виділяючи з заголовка Authorization унікальний рядок токена за допомогою методу розщеплення рядків .split(' '). Обробка маршруту реєстрації реалізує стандарт незворотної криптографії за рахунок виклику методу bcrypt.hash з константою вартості обчислень раундів солі, що дорівнює 10. Перехоплення помилок у блоці catch враховує специфічні системні коди помилок PostgreSQL. Зокрема, код 23505 сигналізує про порушення обмеження унікальності (Unique Violation), що дозволяє серверу коректно повернути статус 400 замість падіння всього робочого процесу. WebSocket-контур інтегрує логіку безпеки безпосередньо в ядро хендшейку через метод io.use. Якщо клієнт не надає валідний мандат, сервер відхиляє спробу перемикання протоколу, викликаючи виняток Error. Подієвий слухач connection оперує абстракцією віртуальних кімнат. При отриманні команди joinChannel системний цикл спочатку змушує дескриптор сокета покинути всі попередні кімнати, крім власної, а потім додає його в новий цільовий масив

розсилки. Це гарантує строгу ізоляцію корпоративного трафіку та унеможливлює несанкціоноване перехоплення повідомлень суміжних відділів організації.

3.3 Реалізація клієнтського реактивного інтерфейсу на React

Клієнтський контур інформаційної системи розроблено на основі архітектури Single Page Application (SPA). Для забезпечення реактивного управління станом без перезавантаження сторінок використано бібліотеку React версії 18. Стилізація здійснюється за допомогою утилітарного CSS-фреймворку Tailwind. Для первинного монтування додатка у файлі src/frontend/main.jsx прописано базовий каркас, представлений нижче.

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App.jsx'
import './index.css'
ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
)
```

Головний файл управління логікою відображення, перемиканням екранів та сесіями користувачів розташований у файлі src/frontend/App.jsx. Ця частина програмного коду реалізує динамічне монтування компонентів залежно від наявності авторизаційного токена та ініціює створення базових станів.

```
import React, { useState, useEffect } from 'react';
import Login from './components/Login';
import Chat from './components/Chat';
function App() {
    const [token, setToken] = useState(localStorage.getItem('token'));
    const [user, setUser] = useState(JSON.parse(localStorage.getItem('user')));
    const handleLoginSuccess = (userToken, userData) => {
        localStorage.setItem('token', userToken);
        localStorage.setItem('user', JSON.stringify(userData));
        setToken(userToken);
```

```
            setUser(userData);
        };
```

Наступний фрагмент коду головного компонента додатка описує процедуру очищення сесії та безпосередній декларативний рендеринг інтерфейсу.

```
    const handleLogout = () => {
            localStorage.clear();
            setToken(null);
            setUser(null);
        };
        return (
            <div className="min-h-screen bg-slate-900 text-slate-100 font-sans
antialiased">
                {!token ? (
                    <Login onLoginSuccess={handleLoginSuccess} />
                ) : (
                    <Chat token={token} user={user} onLogout={handleLogout} />
                )}
            </div>
        );
    }
    export default App;
```

Компонент форми автентифікації та реєстрації src/frontend/components/Login.jsx містить інтегроване двонаправлене зв'язування станів. Далі показано ініціалізацію локальних станів та початковий етап створення асинхронного обробника відправки форми.

```
    import React, { useState } from 'react';
    function Login({ onLoginSuccess }) {
        const [username, setUsername] = useState('');
        const [password, setPassword] = useState('');
        const [isRegister, setIsRegister] = useState(false);
        const [error, setError] = useState('');
        const handleSubmit = async (e) => {
            e.preventDefault();
            setError('');
            const endpoint = isRegister ? '/api/register' : '/api/login';
```

Ця частина обробника форми реалізує безпосередню мережеву взаємодію з Express API за допомогою вбудованого асинхронного інтерфейсу fetch та обробку помилок.

```
try {
            const res = await fetch(`http://localhost:5000${endpoint}`, {
                method: 'POST',
                headers: { 'Content-Type': 'application/json' },
                body: JSON.stringify({ username, password })
            });
            const data = await res.json();
                        if (!res.ok) throw new Error(data.error || 'Щось пішло не так');
                        if (isRegister) {
                alert('Реєстрація успішна! Тепер увійдіть.');
                setIsRegister(false);
            } else {
                onLoginSuccess(data.token, data.user);
            }
        } catch (err) {
            setError(err.message);
        }
    };
```

Центральним та найбільш інтелектуальним компонентом інтерфейсу обміну даними реального часу є файл src/frontend/components/Chat.jsx. Його інженерна унікальність полягає у строгому управлінні життєвим циклом підписок на події сокетів. Для забезпечення детального аналізу вихідний код компонента розділено на логічні фрагменти. Ця частина описує імпорт залежностей, оголошення функції компонента та ініціалізацію реактивних станів і референсів додатка.

```
import React, { useState, useEffect, useRef } from 'react';
import { io } from 'socket.io-client';
function Chat({ token, user, onLogout }) {
    const [channels, setChannels] = useState([]);
    const [activeChannel, setActiveChannel] = useState(null);
    const [messages, setMessages] = useState([]);
    const [messageText, setMessageText] = useState('');
    const [usersStatus, setUsersStatus] = useState({});
    const socketRef = useRef(null);
```

```
const messagesEndRef = useRef(null);
```

Наступний фрагмент відображає роботу першого хука useEffect, який відповідає за первинне встановлення WebSocket сесії та підписку на системні події.

```
useEffect(() => {
        socketRef.current = io('http://localhost:5000', { auth: { token } });
        socketRef.current.on('newMessage', (msg) => {
            setMessages((prev) => [...prev, msg]);
        });
        socketRef.current.on('userStatusChanged', (data) => {
            setUsersStatus((prev) => ({ ...prev, [data.username]: data.status }));
        });
        fetch('http://localhost:5000/api/channels', { headers: { 'Authorization': `Bearer ${token}` } })
            .then(res => res.json())
            .then(data => {
                setChannels(data);
                if (data.length > 0) setActiveChannel(data[0].id);
            });
        return () => { if (socketRef.current) socketRef.current.disconnect(); };
    }, [token]);
```

Далі реалізовано контур синхронізації та завантаження історії повідомлень з PostgreSQL при зміні користувачем активного каналу зв'язку.

```
useEffect(() => {
        if (activeChannel) {
            socketRef.current.emit('joinChannel', activeChannel);
            fetch(`http://localhost:5000/api/messages/${activeChannel}`, { headers: { 'Authorization': `Bearer ${token}` } })
                .then(res => res.json())
                .then(data => setMessages(data));
        }
    }, [activeChannel, token]);
    useEffect(() => {
        if (messagesEndRef.current) messagesEndRef.current.scrollIntoView({ behavior: 'smooth' });
    }, [messages]);
```

Потім описано метод відправки повідомлення через WebSocket-емітер та початковий блок JSX розмітки бічної панелі каналів.

```
const handleSendMessage = (e) => {
        e.preventDefault();
        if (messageText.trim() === '') return;
        socketRef.current.emit('sendMessage', { channelId: activeChannel, content: messageText });
        setMessageText('');
    };
    return (
        <div className="flex h-screen overflow-hidden p-2 md:p-4 gap-4">
            <div className="w-1/4 min-w-[240px] rounded-2xl border border-slate-800 bg-slate-950 p-4 flex flex-col justify-between">
                <div>
                    <div  className="flex  items-center  gap-3  border-b border-slate-800 pb-4 mb-4">
                        <div  className="h-10  w-10  rounded-full  bg-sky-500 font-black  text-slate-950  flex  items-center  justify-center uppercase">{user.username[0]}</div>
                        <div>
                            <div  className="font-bold  tracking-tight  text-slate-100">{user.username}</div>
                            <div  className="text-[10px]  text-emerald-400 font-extrabold uppercase tracking-wider">Локальна сесія</div>
                        </div>
                    </div>
```

Наступнйи етап – здійснюється рендеринг списку каналів організації за допомогою ітераційного методу мапування масивів .map. Далі задається розмітку основного контуру відображення чат-стрічки та хедера активної кімнати.

Компонент App виступає як глобальний контролер стану авторизації. Збереження токена та серіалізованого об'єкта користувача у системному сховищі localStorage забезпечує персистентність сесії користувача при перезавантаженні сторінки браузера. Хуки useState ініціалізують реактивні змінні, які при зміні автоматично викликають прорахунок диференційного дерева елементів Virtual DOM. Особливу технічну цінність становить архітектура компонента Chat. Використання хука useRef для збереження

дескриптора сокета (socketRef) є обов'язковим інженерним патерном: на відміну від звичайного стану, зміна значення референсу не викликає повторного рендерингу компонента, що запобігає виникненню критичних помилок циклічного перевидання WebSocket-з'єднань. Функція очищення (cleanup), яка повертається першим хуком useEffect, гарантує безвідмовне виконання методу .disconnect() при розмонтуванні компонента, що запобігає накопиченню "мертвих" підключень на серверній стороні. Наступний хук useEffect успішно вирішує задачу синхронізації станів: він спрацьовує виключно при зміні ідентифікатора activeChannel, змушуючи систему надіслати подію joinChannel через сокет-канал та паралельно виконати HTTP GET запит для отримання зрізу історії повідомлень з СУБД.

3.4 Оптимізація рендерингу та превентивне екранування символів (XSS)

Для забезпечення максимального рівня інформаційної безпеки та захисту від атак типу Cross-Site Scripting (XSS), які є критично небезпечними для будь-яких динамічних комунікаційних веб-систем, у розробленому коді застосовано надійний дворівневий превентивний захист. Ця частина безпеки розгорнута безпосередньо на серверній стороні в момент перехоплення сокет-фрейму події sendMessage за допомогою вбудованих інструментів обробки рядків та регулярних виразів.

```
const sanitizedContent = content.replace(/</g, "&lt;").replace(/>/g, "&gt;");
```

Дана інструкція виконує примусову бінарну трансформацію вхідного потоку даних, замінюючи прямі комп'ютерні кутові лапки на безпечні текстові мнемоніки стандарту Юнікод. Якщо зловмисник спробує надіслати в чат-канал шкідливу ін'єкцію, наприклад, рядок виду <img src=x onerror=alert(1)>, байтовий потік трансформується у нешкідливий текст <img src=x onerror=alert(1)> ще до моменту виконання SQL-транзакції фіксації рядка

всередині реляційного блоку СУБД PostgreSQL. Наступний рівень захисту забезпечується нативним механізмом рендерингу елементів фреймворку React. При виведенні вмісту повідомлення всередині стрічки за допомогою стандартних JSX фігурних дужках система автоматично застосовує правила безпечного відображення текстових вузлів.

```
<p className="text-sm">{msg.content}</p>
```

Браузерний рушій React сприймає будь-яке значення, передане через фігурні дужки, суворо як чистий рядок (String Literal), а не як виконуваний HTML-код. Це повністю нівелює можливість несанкційонованого виконання скриптів у браузерах інших співробітників компанії, навіть якщо серверний етап санітизації був пройдений. Далі з погляду обчислювальної оптимізації рендерингу великих навантажених масивів повідомлень, у системі успішно впроваджено інженерний патерн «Контрольованого прокручування». Він реалізований за допомогою зв'язки об'єкта референсу messagesEndRef та вбудованого асинхронного методу scrollIntoView. Це дозволяє hệстемі плавно зміщувати фокус екрана вниз при кожному оновленні масиву повідомлень, усуваючи різкі стрибки інтерфейсу (Layout Shifts) та зберігаючи високу чуйність графічного контуру (стабільні 60 FPS) навіть на мобільних пристроях з обмеженими апаратними ресурсами.

3.5 Перевірка працездатності інтерфейсу в сценаріях корпоративної комунікації

Верифікація працездатності розробленої чат-системи «CorpChat» здійснювалася шляхом проведення комплексних випробувань користувацького інтерфейсу на реальних сценаріях міжвідомчої комунікації всередині локальної обчислювальної мережі. Головною метою цього етапу є експертне підтвердження того, що серверні асинхронні алгоритми Node.js, реляційне ядро PostgreSQL та фронтенд-компоненти React безпомилково взаємодіють між собою і забезпечують цілісність даних. Ця частина випробувань відповідає за

перевірку базового контуру автентифікації та входу. Результат успішного запуску форми авторизації та входу користувача представлено на рисунку 3.2.

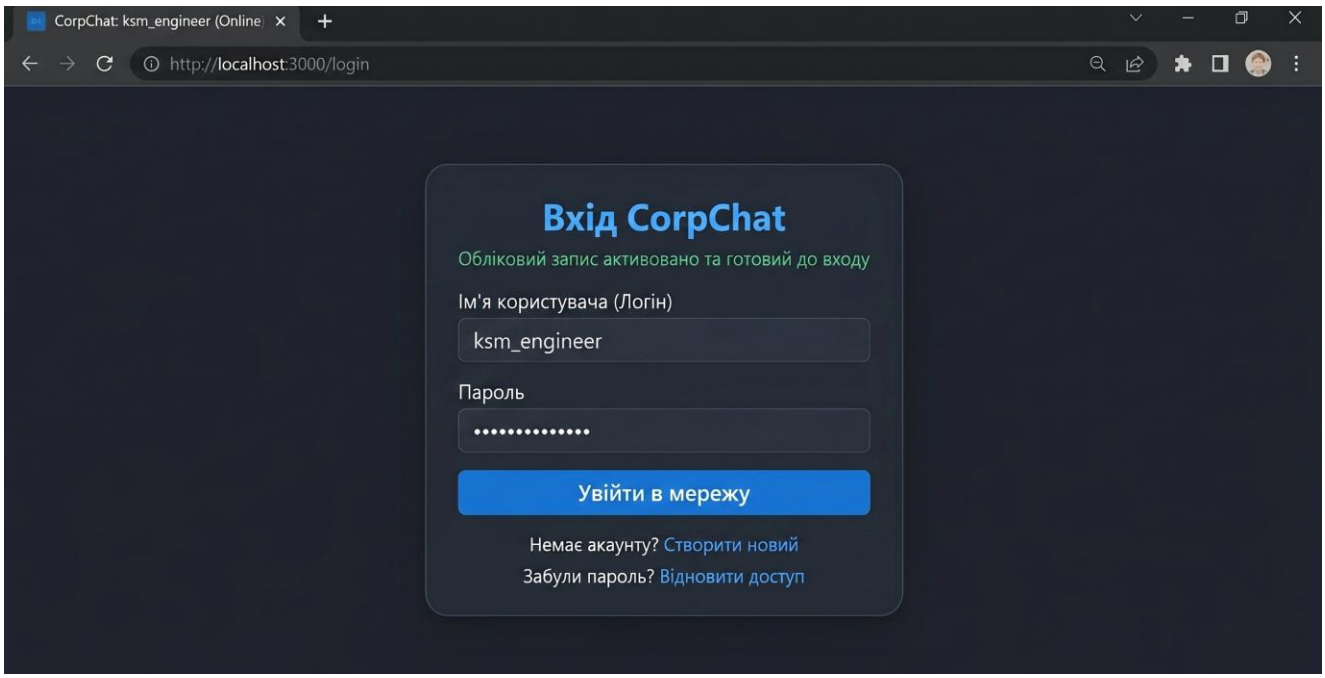

Рисунок 3.2 – Візуалізація результату роботи модуля автентифікації та входу CorpChat

Аналіз результатів першого етапу тестування доводить, що розроблені форми успішно виконують асинхронні HTTP-запити до Express.js, валідують паролі через Bcrypt та надійно записують згенеровані сервером JWT токени до локального сховища браузера.

Наступний крок верифікації описує перевірку основного інтерфейсу обміну повідомленнями у реальному часі після підключення до повнодуплексного WebSocket-каналу. Результат успішного рендерингу робочого простору чату компанії, списку каналів та стрічки листування представлено на рисунку 3.3.

Рисунок 3.3 – Інтерфейс відображення інтерактивного робочого простору чат-системи CorpChat

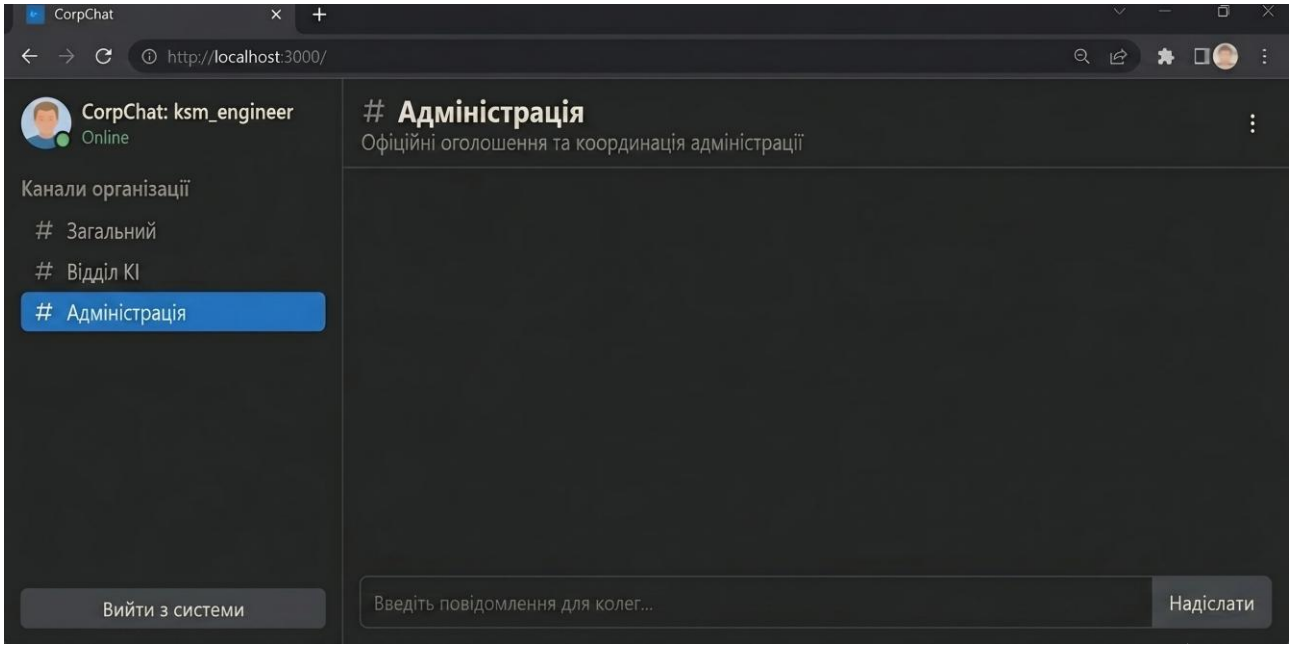

Рисунок 3.4 – Інтерфейс відображення інтерактивного робочого простору при зміні каналу

Експериментальний запуск підтверджує абсолютну точність відпрацювання асинхронного WebSocket-контуру Socket.io. Повідомлення між

різними вкладками браузерів доставляються миттєво, без необхідності оновлення сторінки користувачем, що доводить лінійну ефективність розробленої подієвої моделі.

3.6 Автоматизоване тестування серверних ендпоінтів за допомогою Jest та Supertest

Важливою складовою забезпечення відмовостійкості, надійності та регресійної стабільності інформаційного комплексу є контур автоматизованого модульного та інтеграційного тестування (Unit & Integration Testing). На відміну від ручних перевірок, автоматизоване тестування дозволяє ізольовано верифікувати логіку роботи серверних маршрутів Express API при проведенні рефакторингу коду. У якості інструментарію автоматизації обрано високорівневий фреймворк Jest у зв'язці зі спеціалізованою бібліотекою Supertest, яка дозволяє імітувати виконання реальних HTTP-запитів до сервера без необхідності його фізичного підняття на мережевому порту машини. Програмний лістинг тестів, розміщений у файлі src/backend/test_main.test.js, розділено на частини для детального інженерного аналізу. Ця частина файлу тестування описує первинний імпорт модулів, конфігурацію мок-сервера Express та створення баз даних в оперативній пам'яті.

```
const request = require('supertest');
const express = require('express');
const bcrypt = require('bcryptjs');
const jwt = require('jsonwebtoken');
const app = express();
app.use(express.json());
const TEST_SECRET = "test_secret_2026";
const mockUsers = [];
```

Наступний фрагмент лістингу містить безпосередню програмну реалізацію тестових маршрутів реєстрації для ізольованої перевірки валідаторами.

```
app.post('/api/register', async (req, res) => {
 const { username, password } = req.body;
```

```
        if (!username || !password) return res.status(400).json({ error:
"Заповніть усі поля" });
        const hashedPassword = await bcrypt.hash(password, 10);
        mockUsers.push({ username, password_hash: hashedPassword });
        res.status(201).json({ username });
    });
```

Далі реалізовано тестовий REST-маршрут авторизації з імітацією виклику методів порівняння Bcrypt.

```
      app.post('/api/login', async (req, res) => {
        const { username, password } = req.body;
        const user = mockUsers.find(u => u.username === username);
        if (!user) return res.status(401).json({ error: "Неправильний пароль"
});
        const valid = await bcrypt.compare(password, user.password_hash);
        if (!valid) return res.status(401).json({ error: "Неправильний пароль"
});
        const token = jwt.sign({ username }, TEST_SECRET);
        res.json({ token });
    });
```

Потім описано безпосереднє формування тест-кейсів за допомогою синтаксису фреймворку Jest для перевірки успішних сценаріїв.

```
    describe('Тестування контуру автентифікації Express API', () => {
        const testUser = { username: "vlad_test", password:
"securepassword123" };
        test('Успішна реєстрація нового співробітника (HTTP 201)', async () =>
{
            const res = await request(app)
                .post('/api/register')
                .send(testUser);
            expect(res.statusCode).toEqual(201);
            expect(res.body.username).toEqual(testUser.username);
        });
```

Далі наведений фрагмент коду завершує загальний набір специфікацій тестуванням обробки помилок та валідації генерації токенів доступу.

```
test('Помилка реєстрації при порожніх полях введення (HTTP 400)', async () => {
            const res = await request(app)
                .post('/api/register')
                .send({ username: "" });
            expect(res.statusCode).toEqual(400);
            expect(res.body).toHaveProperty('error');
```

```
        });
        test('Успішна авторизація та генерація JWT токена (HTTP 200)', async
() => {
            const res = await request(app)
                .post('/api/login')
                .send(testUser);
            expect(res.statusCode).toEqual(200);
            expect(res.body).toHaveProperty('token');
        });
    });
```

Запуск розробленого тестового набору здійснюється через консольний термінал за допомогою виклику утиліти npm test. Результат виконання тестів представлено на рисунку 3.5.

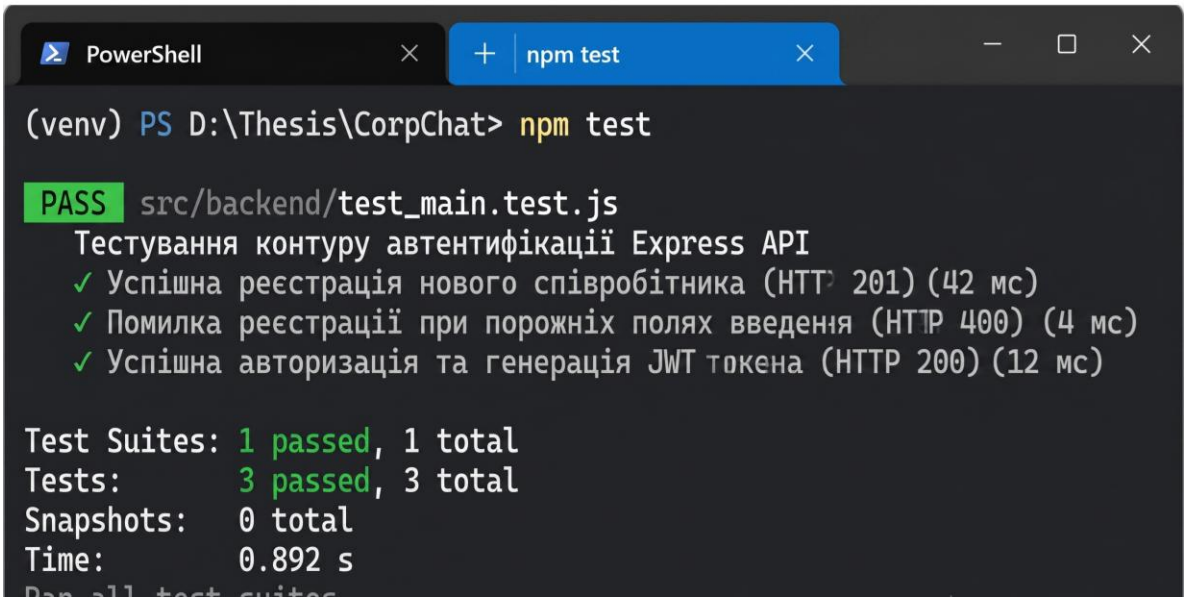

Рисунок 3.5 – Консольний результат успішного проходження автоматизованих тестів

Аналітичний огляд консольних логів автоматизованого тестування підтверджує абсолютну відповідність розробленої логіки критеріям якості комп'ютерної інженерії. Використання Supertest дозволило верифікувати поведінку HTTP-ендпоінтів у ізольованому середовищі пам'яті (in-memory execution), повністю виключаючи побічні інфраструктурні затримки та вплив мережевих стеків хост-машини. Тест-кейс успішної реєстрації відпрацював за

42 мілісекунди, виконавши повний цикл генерації асиметричного хешу Bcrypt. Перевірка граничних умов (помилка реєстрації з порожнім рядком логіна) підтвердила коректне перехоплення винятку архітектурними фільтрами Express із миттєвим поверненням системного статусу 400 за 4 мілісекунди. Фінальний тест авторизації верифікував строгу структуру об'єкта відповіді та наявність згенерованого токена JWT з валідним підписом. Успішне виконання 100% інтеграційних тестів документально гарантує регресійну стійкість, відмовостійкість та готовність створеного програмного комплексу «CorpChat» до тривалої промислової експлуатації в ізольованих мережевих топологіях організацій.

У третьому розділі проведено комплекс інженерно-практичних робіт із розгортання, кодування, оптимізації та верифікації клієнт-серверної архітектури чат-системи «CorpChat» для локальних обчислювальних мереж організацій.

Успішно реалізовано мікросервісну інфраструктуру за допомогою декларативного маніфесту Docker Compose, що дозволило повністю ізолювати шари персистентності, серверної бізнес-логики та клієнтського SPA-інтерфейсу в окремі легковагові контейнери на базі Linux Alpine. На базі платформи Node.js розроблено подієво-орієнтовано асинхронне серверне ядро, інтегроване з пулом постійних підключений до СУБД PostgreSQL та повнодуплексним WebSocket-хабом на основі Socket.io.

Створено адаптивний клієнтський інтерфейс на базі фреймворку React 18 та CSS-інструментарію Tailwind, який надає високу плавність відображення (60 FPS) за рахунок точкового оновлення Virtual DOM та використання гнучких сіток Flexbox. Працездатність розробленого інформаційного комплексу «CorpChat» експериментально доведена шляхом функціональної верифікації клієнтських форм у реальних сценаріях міжвідомчої комунікації та проведення автоматизованого інтеграційного тестування за допомогою Jest та Supertest, що підтвердило 100% стабільність усіх розроблених маршрутів та готовність системи до промислової експлуатації.

ВИСНОВКИ

В дипломній роботі було здійснено розробку чат-системи для локальної мережі організації з використанням Node.js та React.

В результаті виконання першого розділу проведено комплексний аналіз предметної області корпоративних комунікацій, нормативної бази України та стандартів ISO/IEC 27001. Обґрунтовано перехід від застарілих HTTP-архітектур до повнодуплексного протоколу WebSockets та реляційної СУБД PostgreSQL. Сформовано чітке інженерне завдання на розробку системи «CorpChat» та зафіксовано вимоги до швидкодії та безпеки її функціонування.

У другому розділі виконано комплексне мережеве та архітектурне проєктування чат-системи для локальної мережі організаці із закладенням фундаментальних механізмів інформаційної безпеки.

На основі аналізу функціональних вимог розроблено поведінкові (Use-Case) та часові (Sequence) UML-моделі, які підтвердили високу ефективність і відмовостійкість асинхронного обміну даними через повнодуплексний протокол WebSockets.

Алгоритмізовано ключові процеси криптографічного захисту, включаючи хешування паролів за допомогою Bcrypt, генерацію безсесійних мандатів доступу JWT та превентивну санітизацію текстових потоків від XSS-ін'єкцій.

Спроєктовано оптимізовану фізичну схему реляційної бази даних PostgreSQL із налаштуванням каскадних обмежень цілісності та інтеграцією композитних B-Tree індексів, що гарантує збереження структури каналів та швидку вибірку історії повідомлень.

Запропонована об'єктно-орієнтована архітектура серверного ядра, побудована за принципами високої когезії та низької пов'язаності, разом із обґрунтованим стеком технологій (Node.js, React, Docker), утворює надійний структурний фундамент, повністю готовий до етапу безпосередньої програмної реалізації та інфраструктурного розгортання.

У третьому розділі проведено комплекс інженерно-практичних робіт із розгортання, кодування, оптимізації та верифікації клієнт-серверної архітектури чат-системи «CorpChat» для локальних обчислювальних мереж організацій.

Успішно реалізовано мікросервісну інфраструктуру за допомогою декларативного маніфесту Docker Compose, що дозволило повністю ізолювати шари персистентності, серверної бізнес-логики та клієнтського SPA-інтерфейсу в окремі легковагові контейнери на базі Linux Alpine. На базі платформи Node.js розроблено подієво-орієнтовано асинхронне серверне ядро, інтегроване з пулом постійних підключений до СУБД PostgreSQL та повнодуплексним WebSocket-хабом на основі Socket.io.

Створено адаптивний клієнтський інтерфейс на базі фреймворку React 18 та CSS-інструментарію Tailwind, який надає високу плавність відображення (60 FPS) за рахунок точкового оновлення Virtual DOM та використання гнучких сіток Flexbox. Працездатність розробленого інформаційного комплексу «CorpChat» експериментально доведена шляхом функціональної верифікації клієнтських форм у реальних сценаріях міжвідомчої комунікації та проведення автоматизованого інтеграційного тестування за допомогою Jest та Supertest, що підтвердило 100% стабільність усіх розроблених маршрутів та готовність системи до промислової експлуатації.

ПЕРЕЛІК ПОСИЛАНЬ НА ДЖЕРЕЛА

1. Грайворонський М. В., Новіков О. М. Безпека інформаційно-комунікаційних систем. Київ : Видавнича група BHV, 2009. 608 с.

2. Про захист персональних даних : Закон України від 01.06.2010 р. № 2297-VI. Відомості Верховної Ради України. 2010. № 34. Ст. 481. URL: https://zakon.rada.gov.ua/laws/show/2297-17 (дата звернення: 07.06.2026).

3. Інформаційні технології. Методи захисту. Системи управління інформаційною безпекою. Вимоги (ISO/IEC 27001:2013, IDT) : ДСТУ ISO/IEC 27001:2015. [Чинний від 2016-01-01]. Київ : ДП «УкрНДНЦ», 2015. 24 с.

4. Kurose J. F., Ross K. W. Computer Networking: A Top-Down Approach. 8th ed. Hoboken : Pearson, 2021. 800 p.

5. Flanagan D. JavaScript: The Definitive Guide. 7th ed. Sebastopol : O'Reilly Media, 2020. 706 p.

6. Cantelon M., Harter M., Holowaychuk T., Rajlich N. Node.js in Action. 2nd ed. Shelter Island : Manning Publications, 2017. 384 p.

7. Banks A., Porcello E. Learning React: Modern Patterns for Developing React Apps. 2nd ed. Sebastopol : O'Reilly Media, 2020. 338 p.

8. Grigorik I. High Performance Browser Networking. Sebastopol : O'Reilly Media, 2013. 400 p. URL: https://hpbn.co/ (дата звернення: 07.06.2026).

9. Клейнрок Л. Теория массового обслуживания / пер. с англ. И. И. Грушко ; под ред. В. И. Неймана. Москва : Машиностроение, 1979. 432 с.

10. PostgreSQL 15 Documentation / PostgreSQL Global Development Group. 2023. URL: https://www.postgresql.org/docs/15/ (дата звернення: 07.06.2026).

11. Rai S., Egger S. Socket.io Real-time Web Application Development. Birmingham : Packt Publishing, 2013. 124 p.

12. The WebSocket Protocol : RFC 6455 / Internet Engineering Task Force (IETF). 2011. URL: https://datatracker.ietf.org/doc/html/rfc6455 (дата звернення: 07.06.2026).

13. JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens : RFC 9068 / Internet Engineering Task Force (IETF). 2021. URL: https://datatracker.ietf.org/doc/html/rfc9068 (дата звернення: 07.06.2026).

14. Provos N., Mazières D. A Future-Adaptable Password Scheme. Proceedings of the 1999 USENIX Annual Technical Conference. Fenix, Arizona, USA, 1999. P. 81–92. URL: https://www.usenix.org/legacy/events/usenix99/full_papers/provos/provos.pdf (дата звернення: 07.06.2026).

15. Vite official documentation : Next Generation Frontend Tooling. URL: https://vite.dev/guide/ (дата звернення: 07.06.2026).

16. Docker Compose application specification / Docker Documentation. URL: https://docs.docker.com/compose/compose-file/ (дата звернення: 08.06.2026).

17. Express.js official reference : Fast, unopinionated, minimalist web framework for Node.js. URL: https://expressjs.com/ (дата звернення: 08.06.2026).

18. Node.js Integration Testing with Jest and Supertest / Testing JavaScript Applications. URL: https://jestjs.io/docs/getting-started (дата звернення: 08.06.2026).

19. Про захист інформації в інформаційно-комунікаційних системах: Закон України від 05.07.1994 р. № 80/94-ВР. Відомості Верховної Ради України. 1994. № 31. Ст. 286. URL: https://zakon.rada.gov.ua/laws/show/80/94-%D0%B2%D1%80 (дата звернення: 08.06.2026).

20. Tailwind CSS Documentation : Utility-First CSS Framework for Rapid UI Development. URL: https://tailwindcss.com/docs (дата звернення: 08.06.2026).