МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

КАФЕДРА КОМП'ЮТЕРНИХ НАУК

ВИПУСКНА РОБОТА

на тему:

«Інформаційна система обміну повідомленнями між розробниками програмного забезпечення»

Завідувач випускаючої кафедри — **Довбиш А.С.**

Керівник роботи — **Шелехов І.В.**

Студент гр. ІН–63 — **Кіптенко Є.А.**

СУМИ 2020

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

Кафедра комп'ютерних наук

Затверджую _____

Зав. кафедрою Довбиш А.С.

“_____” _____2020 р.

Завдання

до випускної роботи

Студента четвертого курсу, групи ІН-63 спеціальності “Комп’ютерні науки” денної форми навчання Кіптенко Євгена Андрійовича.

Тема: «Інформаційна система обміну повідомленнями між розробниками програмного забезпечення»

Затверджена наказом по СумДУ

№_____ від _____2020 р.

Зміст пояснювальної записки: 1) аналіз проблеми. постановка задачі дослідження; 2) дослідження алгоритмів навчання інтелектуальної системи; 3) розробка інформаційного та програмного забезпечення інтелектуальної системи.

Дата видачі завдання “_____”_____2020 р.

Керівник випускної роботи _____Шелехов І.В.

Завдання прийняв до виконання _____ Кіптенко Є.А.

РЕФЕРАТ

Записка: 58 стр., 10 рис., 5 табл., 1 додаток, 23 джерела

Об’єкт дослідження: процес розробки інформаційного та програмного забезпечення спеціалізованої системи обміну повідомленнями

Мета роботи: розробити та програмно реалізувати інформаційну веб-орієнтовану систему обміну повідомленнями для розробників програмного забезпечення

Методи дослідження: методи проектування інформаційних систем, методи формування та оптимізації структури баз даних, методи тестування програмного забезпечення

Результати: Було проведено розробку та програмну реалізацію інформаційної системи для обміну повідомленнями між розробниками програмного забезпечення в реальному часі. Основними етапами розробки були: вибір методології проектування інформаційної системи, проектування серверної та клієнтської частин інформаційної системи, визначення структури бази даних інформаційної системи, вибір програмного середовища для реалізації інформаційної системи та реалізація модулів аутентифікації користувача, обміну текстовими повідомленнями, обміну частинами програмного коду інформаційної системи, тестування системи

AGILE, SCRUM, КОНТЕКСТНА ДІАГРАМА, BACK-END, FRONT-END, БАЗА ДАНИХ, АУТЕНТИФІКАЦІЯ, ПОВІДОМЛЕННЯ, ЧАТ

ЗМІСТ

ВСТУП

В умовах сучасного інформаційного суспільства комп'ютерні технології настільки вкоренилися в нашому житті, що повністю змінили способи обміну інформацією, які ми використовуємо для спілкування з друзями, членами сім'ї і діловими партнерами. Незважаючи на те, що електронна пошта стає все більш мобільною, ділові люди все більше і більше звертаються до тих же самих засобів комунікації, які звичайні користувачі застосовують давно і з величезним успіхом: до миттєвого обміну повідомленнями (Instant Messaging, IM). Система миттєвого обміну повідомленнями - це служби для обміну повідомленнями в режимі реального часу. Щоб використовувати подібні сервіси, необхідні лише вихід в інтернет і відповідна клієнтська програма (IM-клієнт). Традиційно в корпоративній системі використовувалася електронна пошта, але з розвитком Instant Messaging вона поступово відійшла на другий план. Різниця між месенджерами і електронною поштою полягає в тому, що повідомлення передаються миттєво, а також є можливість бачити, чи знаходиться абонент в мережі.

В роботі розв’язується задача проектування та реалізації спеціалізованої системи обміну повідомленнями для розробників програмного забезпечення, що дозволяє обмінюватися не тільки текстовому, аудіо і відео даними, але і частинами програмного коду.

1 АНАЛІТИЧНИЙ ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

1.1 Сучасні платформи для обміну повідомленнями

На теперішній час існує декілька додатків для обміну повідомленнями та кілька додатків для чатів, які підходили розробникам. Однак раніше не проводився глибокий аналіз їхніх інструментів, щоб з'ясувати, чи достатньо вони хороші для розробників. Аналіз показує, що у деяких з них були відсутні функції, які вважаються вирішальними, а інші не мали можливості для подальшого вдосконалення. В ході роботи вдалося отримати уявлення про те, що будувати та як, і визначати, які технології та стратегії використовувати на основі свого досвіду. Такі компанії, як Slack, регулярно публікують оновлення розробки (наприклад, огляди продуктивності, порівняння технологій та повідомлення про масштабованість).

Під час аналізу було досліджено такі платформи:

- Flowdock - https://www.flowdock.com
- Gitter - https://gitter.im
- Hangouts - https://hangouts.google.com
- Matrix - http://matrix.org
- Messenger - https://messenger.com
- Rocket.chat - https://rocket.chat
- Skype - https://web.skype.com
- Slack - https://slack.com
- Telegram - https://web.telegram.org
- Whatsapp - https://web.whatsapp.com

Gitter і Slack - це ті продукти, що найбільше зосереджені на розробниках. Хоча основні речі, такі як обмін кодом, були можливі і в деяких інших, таких як Rocket.chat, було зрозуміло, що вони не орієнтуються на розробників, що часто призводить до недостатньої кількості інструментів для них.

Для розробників програмного забезпечення потрібне професійне середовище, яке робить обмін кодом ефективним, а також надає можливість інтегрувати його у сховище вихідного коду.

У цей момент було зрозуміло, що слід звузити повний аналіз до Gitter і Slack. Інші платформи все ще були корисними для отримання декількох загальних концепцій, але вони були практично не пов’язані з темою роботи. Slack і Gitter - це популярні платформи, які зосереджені на продуктивності та розробниках.

У Slack є система на основі кімнат (називається "команди"), але вона не має можливості створювати чати, пов'язані з git або сховищем GitHub, що потребує вдосконалення. Крім того, існує інтеграція з ботами (які можуть надсилати повідомлення будь-якого типу, наприклад, канал GitHub), але це призводить до дуже захащеного чату, якщо сховище активно використовується. Більше того, жоден з них не має можливості розгортати розмови на основі попереднього повідомлення. На даний момент, як на Gitter, так і на Slack, розгалуження для чату потрібно робити вручну, що передбачає набагато більше роботи для користувача, ніж створення чатів і їх знищення одним натисканням кнопки.

В другому розділі використовувалася методологія Agile, щоб визначити параметри ідеальної платформи для обміну повідомленнями між розробниками на базі конкурентного аналізу. Прикладом таких параметрів можуть бути такі особливості платформи:

- Новачки можуть ідентифікувати себе, створивши новий обліковий запис (локальна автентифікація) або автентифікувавшись через соціальний акаунт, наприклад, Facebook, Google або Twitter.
- Сайт буде заохочувати відвідувача до автентифікації через GitHub, оскільки надання дозволу GitHub дає декілька спеціальних інструментів розробника. Сюди входить відображення їх списків сховищ, релізи, розгалуження, запити та коментарі.

- Вибір унікального імені користувача є обов'язковою умовою. В роботі буде використовуватися обране ім’я користувача для візуальної ідентифікації на різних сервісах, таких як групові бесіди або професійні сторінки, замість їх справжнього імені.
- Зареєстрований користувач може створити власні кімнати, які можуть бути публічними або приватними. "Кімната" - це не що інше, як контейнер для кількох "чатів" (віртуального місця, де відбуваються розмови). Адміністратори проекту можуть захотіти створити власні кімнати для кожного з своїх проектів, маючи один або декілька чатів для обговорення кожного проекту.
- Будь-який користувач може читати та приєднуватися до кімнат інших учасників, якщо вони загальнодоступні, або за запрошенням одиного із учасників, якщо вони приватні.
- Адміністратор кімнати може створювати та видаляти свої чати. Чати можуть бути як загальними, так і на основі Git. Учасники повинні вибрати Git, якщо вони мають віддалений вихідний код.
- Сховище, наприклад, GitHub. Чати на основі Git мають тісну інтеграцію з платформою віддаленого постачальника Git, наприклад, можливість дивитися записи чи задачі в режимі реального часу. При цьому можна робити закладки, що дозволяє отримати доступ до них пізніше з домашньої сторінки, замість того, щоб шукати їх у пошуковій системі сайту або отримувати доступ до них за допомогою прямого посилання.
- Крім того, існує система репутації на основі ідентифікатору користувача, яка сприяє груповим дискусіям та продуктивності праці. Користувачі можуть отримати репутацію, спілкуючись в чаті або працюючи (оскільки діяльність GitHub зараховується до репутації).
- Що стосується функцій чату, їх багато. Кілька з них дуже добре відомі, наприклад, file sharing. Інші, частково, тому що вони дуже специфічні, як, наприклад, обмін кодом або форматування MarkDown.

1.2 Комунікаційні протоколи

Для більшості веб-додатків протоколи зв’язку не є предметом обговорення. AJAX через HTTP - це надійний спосіб комунікації. Однак не в нашому випадку. В роботі потрібен, хоча і не в кожній ситуації, надзвичайно швидкий спосіб спілкування для надсилання / отримання повідомлень у режимі реального часу. Для обміну повідомленнями в Інтернеті доступно декілька протоколів зв'язку. Найпопулярніші з них - AJAX, WebSockets та WebRTC.

AJAX - це повільний підхід. Не тільки через заголовки, які потрібно надсилати в кожному запиті, але також через те, що немає можливості отримувати повідомлення про нові повідомлення у чаті. Використовуючи AJAX, нам доведеться робити запити і обробляти нові повідомлення з сервера кожні кілька секунд, що призводить до появи на екрані нових повідомлень з затримкою в кілька секунд, не кажучи про численні зайві запити, що така процедура породжуватиме.

WebSockets - кращий підхід. Для встановлення з'єднань WebSockets може знадобитися до декількох секунд, але завдяки повнодуплексному каналу зв’язку повідомленнями можна швидко обмінюватися (в середньому затримка в кілька мілісекунд на повідомлення). Крім того, і клієнт, і сервер можуть отримувати повідомлення про нові запити через один і той же канал зв'язку, що означає, що на відміну від AJAX, клієнту не потрібно надсилати сервер запит для отримання нових повідомлень, а лише чекати, поки сервер їх надійшле.

WebRTC - це новий протокол зв'язку, доступний для найсучасніших браузерів (Chrome, Firefox та Opera). Він розроблений для високопродуктивних якісних систем відео-зв’язку, передачі аудіо та довільних даних [1]. WebRTC не вимагає жодного сервера в якості проксі-сервера для обміну даними, крім сервера синхронізації, який необхідний для спільного використання метаданих мережі та медіа (це часто робиться через

WebSockets). Той факт, що потокові дані можуть обмінюватися між клієнтами безпосередньо, часто означає швидший обмін повідомленнями та менше навантаження на сервер.

WebRTC може працювати через TCP і UDP, але він часто працює з UDP за замовчуванням. Незважаючи на те, що UDP може призвести до втрати пакету, він дає кращу продуктивність, що може призвести до більш рівномірного голосового чи відеодзвінка.

Враховуючи переваги та недоліки трьох технологій, було вирішено використовувати WebSockets для обміну повідомленнями в режимі реального часу, що гарантує нам доставку пакетів (на відміну від кадрів під час відеодзвінка, бажано не пропускати жодне текстове повідомлення).

WebRTC буде використовуватися як для голосових, так і для відеодзвінків [2], які в результаті стали частиною майбутньої роботи. У більшості випадків WebRTC зробить розмову більш вигідною завдяки більш швидкому часу доставки пакетів. Що стосується втрачених пакетів голосу чи відео, ми насправді не переймаємось цим, доки їх лише декілька.

Якщо мова йде про такі запити, як аутентифікація, створення кімнати чи лістинг, AJAX - хороший варіант. Він не вимагає постійного підключення до сервера, що призводить до меншої витрати ресурсів як для клієнта, так і для сервера, і час відповіді на запити буде достатнім. Однак жоден із цих запитів не вимагає надзвичайно швидкого реагування.

1.3 Постановка задачі

Метою роботи є розробка та програмна реалізація інформаційної системи для обміну повідомленнями між розробниками програмного забезпечення в реальному часі. Для досягнення поставленої мети необхідно виконати такі завдання:

1. Обрати методологію проектування інформаційної системи
2. Виконати проектування серверної частини інформаційної системи
3. Виконати проектування клієнтської частини інформаційної системи

4. Визначити структуру бази даних інформаційної системи
5. Обрати програмне середовище для реалізації інформаційної системи
6. Програмно реалізувати такі модулі інформаційної системи:
    - модуль аутентифікації користувача
    - модуль обміну текстовими повідомленнями
    - модуль обміну частинами програмного коду
7. Виконати тестування інформаційної системи

2 ПРОЕКТУВАННЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ

2.1 Методологія проектування

Agile - це набір методів управління проектами з розробки програмного забезпечення. Він має такі особливості:

- систему швидкого реагування на зміни та нові вимоги.
- можливість роботи в команді, і роботи із клієнтом.
- побудова операційного програмного забезпечення з одночасним створенням документації.
- механізм ефективної взаємодії з наявними засобами розробки.

Використовуючи Agile, можливо зосередитись лише на найбільш пріоритетних особливостях інформаційної системи

Scrum

Scrum - один з найпопулярніших фреймворків для розробки програмного забезпечення Agile. Він дозволяє зробити процес розробки ітераційним та поступовим.

Мета Scrum - створити робочі версії продукту за короткий проміжок часу. Кінцевий варіант буде ставати все більш повним у кожній ітерації. Однією з головних особливостей Scrum є те, що зрозуміло, що вимоги можуть змінюватися в будь-який момент і що кілька змін вимог не повинні мати вплив, на час завершення роботи над продуктом.

Працюючи з Scrum, ті, хто бере участь у проекті, починають працювати із відставанням продукту, що відбражається сортованим списком завдань за пріоритетністю. В роботі свій початковий список представлено у наступному розділі, де подано інформацію про особливості використання системи та основні сценарії. Scrum працює зі спринтами – короткими періодами часу, протягом яких передбачається виконати певний обсяг робіт.

Історії користувачів є одним із основних елементів розробки при роботі з методологією Agile. Історія користувача визначає вимоги, що містить

достатньо інформації, щоб розробники могли дати обґрунтовану оцінку необхідності залучення певних засобів для реалізації системи [3].

Оскільки в роботі використовувалась Agile, цей список не повинен був бути повним до початку роботи над проектом, але бажано було мати хоча б кілька пунктів для початку, щоб можна було встановити належні пріоритети.

На початку кожного спринту необхідно проаналізувати всі історії користувачів, оцінювали вартість, яку вони додавали до проекту, і кількість часу, яку необхідно витратити на виконання кожного з них, і відсортувати їх за спадним порядком за важливістю функції системи і часом необхідним для її реалізації. Ці критерії є досить суб'єктивними, тому в роботі найвищий пріоритет надавався функціям, які є найбільш важливими для платформи (наприклад, миттєві текстові повідомлення) або дуже пов'язані з темою чату (наприклад, кодування). Їм надавалась оцінка від 1-10. Часова вартість була оцінкою того, скільки часу буде необхідно витрати на реалізацію функції, враховуючи, що кожен робочий день – це 4 годин. Потім вона переводилась в значення критерію наступним чином:

- 1-2 дні: 1
- 3-4 дні: 2
- 5-6 днів: 3
- 7-9 днів: 4
- 10+ днів: 5

Для сортування списків відставання продукту та спиртів, використовувався третій критерій – пріоритет, який був просто різницею між значенням оцінки важливості і оцінки часу. Тим не менш, у деяких випадках доводилося робити винятки через залежність від історії користувачів. Наприклад, функції входу та реєстрації повинні були бути реалізовані першими, оскільки системі потрібна інформація про користувача, щоб правильно ідентифікувати власника кімнати або відправника повідомлення.

Наш первинний перелік функцій був таким:

Таблиця 2.1 – Первинні вимоги

№	Історія користувача	Значення	Час	Пріоритет
1	Як користувач, я хочу мати змогу зареєструватися за допомогою унікального імені користувача (яке служить ідентифікатором на всій платформі).	1	4	10*
2	Як користувач, я хочу мати змогу ввійти в систему.	1	4	10*
3	Як користувач, я хочу мати можливість вийти з системи, якщо я більше не буду використовувати платформу на даному пристрої.	1	1	10*
4	Як керівник проекту я хочу створити нову віртуальну кімнату для мого проекту.	5	2	10*
5	Як користувач, я хочу мати можливість приєднатися до кімнат, створених раніше менеджером проекту.	7	1	10*
6	Як користувач, я хочу мати можливість створювати дискусійні чати.	6	2	10*
7	Як користувач, я хочу мати можливість вводити чати в кімнаті, в якій я перебуваю.	6	1	10*
8	Як користувач, я хочу мати можливість читати текстові повідомлення в режимі реального часу в чаті.	10	4	8
9	Як користувач, я хочу мати можливість писати текстові повідомлення, які відображатимуться іншим членам кімнати в режимі реального часу.	10	2	8
10	Як користувач, я хочу мати можливість форматувати повідомлення (напівжирний шрифт, курсив, посилання, ...).	8	2	7
11	Як користувач, я хочу мати можливість ділитися зображеннями під час спілкування в чаті.	8	2	7
12	Як користувач, я хочу мати змогу ділитися фрагментами коду.	9	3	7
13	Як користувач, я хочу мати можливість приклеїти повідомлення в чаті поверх чату.	7	2	6
14	Як користувач, я хочу мати можливість редагувати чат.	7	2	6
15	Як користувач, я хочу мати можливість виділити деякі частини повідомлень.	7	2	6
16	Як користувач, я хочу мати можливість зв’язати чат із сховищем GitHub, щоб учасники чату могли дивитись його оновлення в режимі реального часу (формувати запити, задачі, коментарі тощо).	9	6	6
17	Як користувач, я хочу мати можливість отримувати повідомлення про активність в чаті.	8	4	6
18	Як користувач хочу, щоб оновлення інформації про чат відображалися в режимі реального часу.	6	1	5
19	Як користувач я хочу мати змогу об'єднати розгалуження після того, як його обговорення в чаті закінчилося.	7	4	5
20	Як користувач я хочу мати змогу прочитати опис URL-адрес.	6	2	5

Продовження табл. 2.1

№	Історія користувача	Значення	Час	Пріоритет
21	Як користувач, я хочу мати можливість публікувати емоції у чаті	6	1	5
22	Як користувач, я хочу мати можливість надсилати файли будь-якого типу.	7	4	5
23	Як керівник проекту, я хочу мати можливість видаляти учасників кімнати.	6	2	5
24	Як користувач, я хочу мати можливість публікувати опити в чаті в реальному часі.	6	4	4
25	Як розробник я хочу мати доступ до публічного API, щоб мати можливість будувати додаток на його основі (наприклад, інтеграція ботів).	9	30	4
26	Як керівник проекту я хочу мати можливість видалити кімнату.	4	2	3
27	Як користувач, я хочу отримувати сповіщення на робочому столі, коли хтось надіслав повідомлення, а я не був на сторінці чату.	3	1	2
28	Як користувач, я хочу мати змогу змінити свій аватар.	2	2	1
29	Як користувач, я хочу мати змогу змінити свій статус (онлайн, офлайн, відсутній, зайнятий, ...).	2	2	1
30	Як користувач, я хочу мати змогу знаходити контакти, підключаючись до соціальних мереж.	2	2	1
31	Як користувач, я хочу мати можливість приймати участь в групових дзвінках.	6	20	1
32	Я, як користувач, хочу стати частиною репутаційної системи (заснованої на моєму чаті та активності GitHub).	6	15	1

В ході реалізації проекту виникли кілька нових вимог, що за методологією Agile development були включені в відповідну таблицю. Нова таблиця вимог виглядає наступним чином:

Таблиця 2.2 – Нові вимоги

№	Історія користувача	Значення	Час	Пріоритет
1	Як користувачеві, було б зручно мати цільову сторінку, яка вказує на існуючі чати.	8	2	7
2	Як користувач, я хочу мати підтримку MarkDown для повідомлень, що включає принаймні жирний шрифт, курсив, таблиці, списки та гіперпосилання.	8	2	7
3	Як користувач, я хочу, щоб повідомлення в чаті автоматично прокручувались, без необхідності їх вручну прокручувати вниз, як тільки я отримую нове повідомлення, і контейнер для чату заповнений.	7	1	6

Продовження табл. 2.2

№	Історія користувача	Значення	Час	Пріоритет
4	Як користувач, я хочу, щоб повідомлення про активність прокручувались автоматично, без необхідності прокручування вручну, коли я отримую нові оновлення і контейнер для повідомлень заповнений.	7	1	6
5	Як користувач, я хочу, щоб поле для введення тексту чату автоматично змінювало розмір, якщо текстове повідомлення занадто велике і не відповідає типовому.	7	1	6
6	Як користувач, я хочу автоматично підключитися до чату, якщо моє інтернет з’єднання відновиться після зникнення .	5	3	3
7	Як користувач, я б швидше не завантажував усі розмови одразу, а лише тоді, коли мені потрібно.	4	3	2

Як ви вже могли помітити, відставання продукту, а отже, і спринт-журнали, йдуть поруч із функціями ідеальної платформи. Це має сенс, оскільки у нас є безліч історій користувачів, де вони описують характеристики, які хотілося б мати, але це також означає, що деякі з них залишаться не реалізованими. В роботі було завершено всі історії користувачів, які мали пріоритет п'ять і вище.

2.2 Проектування серверної частини

Архітектура програми складається з Back-End та Front-End частин, що для проектування яких використовуються різні засоби та методи.

Front-End – частина, яку бачить кінцевий користувач на сайті.

Back-End містить всі дані та частину логіки, працює на сервері.

2.2.1 Back-End

Back-End реалізовує частину логіки та зберігає дані на стороні сервера. У нашому випадку Back End гарантує, що дані, введені через клієнтську програму (Front-End) будуть збережені та оброблені.

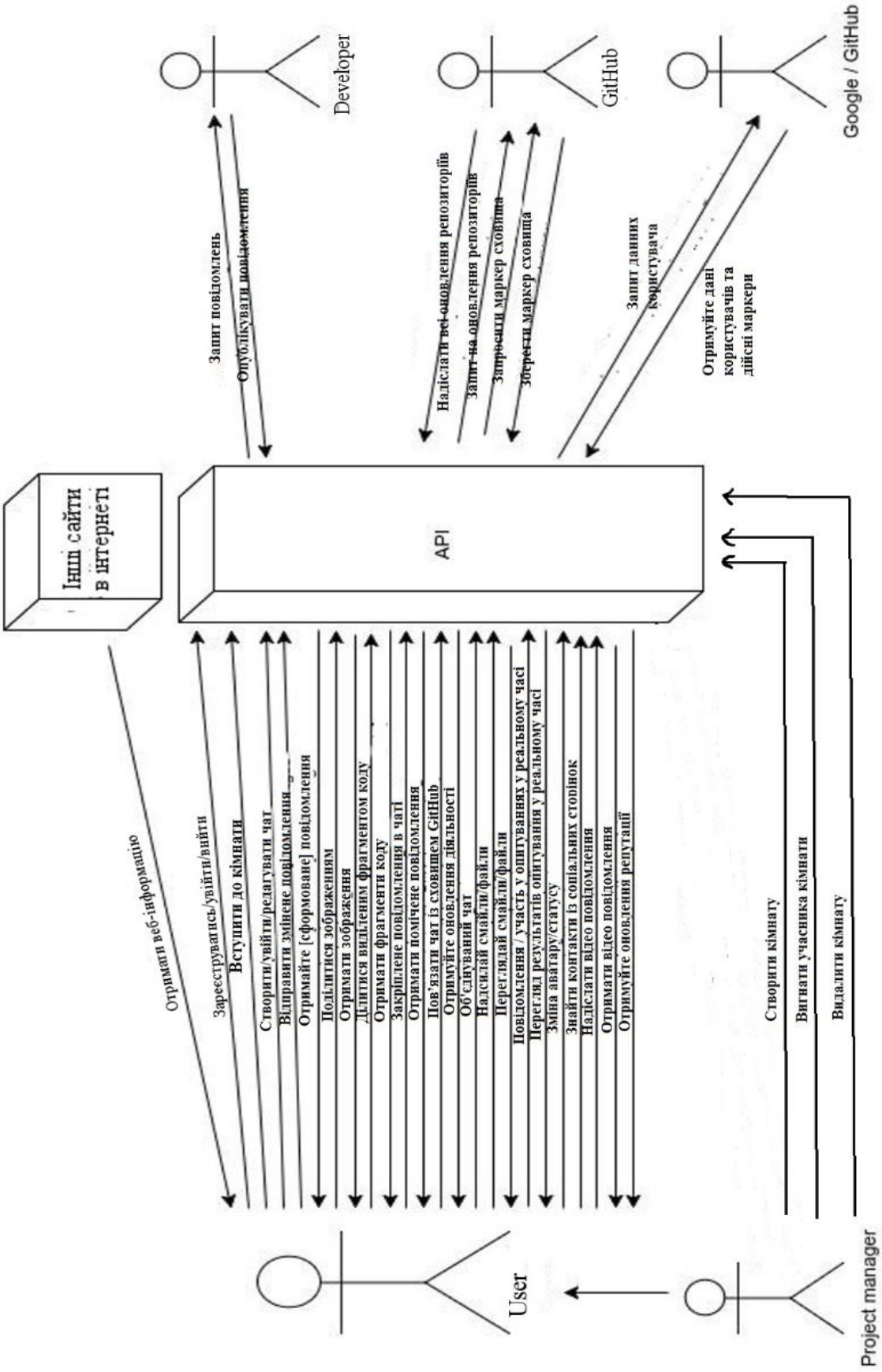


Рисунок 2.1 – Початкова контекстна діаграма

Оскільки Front-End можна змінювати (вихідний код доступний для кінцевого користувача), необхідно переконатися, що всі запити, які будуть отримані, спочатку пройдуть перевірку на валідність сервером, в тому числі, перевірку URI, дозволів користувача тощо. Якщо дані запиту є валідними, система переходить до виконання певної логіки, що супроводжується одним або декількома запитами до бази даних.

2.2.2 API

Система, що розробляється, фактично є системою вводу-виводу даних. Для її реалізації необхідно було програмне середовище, яке здатне реалізувати обробку безлічі запитів в секунду і не є вимогливим щодо ресурсів робочих станцій і серверу. Таким чином, необхідно було зробити вибір між PHP, Python, Java, Go або Node.js. При цьому Node.js був найбільш ефективним для обробки запитів вводу / виводу через асинхронну обробку в одному потоці.

Середовищем для розробки серверної частини був обраний Node.js не лише завдяки продуктивності, а й тому, що воно дозволяло швидко реалізувати необхідні компоненти системи, в порівнянні з іншим мовам, таким як Java. Для розробки клієнтської частини можна в такому випадку застосувати Express, який також використовує Node.js, що дозволило підвищити оперативність розробки веб-контенту.

Можливою альтернативою Node.js був Go, який стає популярним сьогодні через дещо швидший ввод / вивід, ніж Node.js завдяки спеціальним підпрограмам Go, і безперечно кращою продуктивністю під час інтенсивних обчислень [4].

Тим не менш, Go означав більш низьку швидкість розробки. У ньому бракувало бібліотек, оскільки він не був настільки розвинутим, як Node.js, і громіздке управління JSON зробило його не дуже ефективним для нашого додатка (оскільки клієнт JavaScript постійно використовував JSON).

Основні фреймворки та/або бібліотеки, які використовуються при розробці програми:

- Express: фреймворк Node.js, який дозволяє робити швидку розробку веб-сторінок. Він автоматизує більшу частину Back-End розробки. Він також може надавати представлення даних (свого роду HTML-шаблони зі змінними) для Front-End. Використовуючи Express, можна зосередитись на логіці кожного запиту, а не на самому запиті.
- Mongoose: Бібліотека високого рівня MongoDB. Використовуючи об'єкти як моделі баз даних, які згодом стануть даними всередині наших таблиць, вона реалізує базові функції вставки, оновлення та видалення, а також перевірку для кожного з полів.
- Passport: бібліотека аутентифікації для Node.js. Використовуючи різні модулі входу (по одному модулю на провайдера), він приховує всю складність аутентифікації за OAuth, OAuth2, OpenID, GitHub та Google, а також аутентифікації з використанням електронної пошти.
- Sinon: велика бібліотека для тестування, яка містить набір корисних утиліт [5]. Під час тестування системи виникає необхідність знати, чи була викликана певна функція, чи викликана вона з правильними параметрами чи навіть підробити зовнішні вхідні дані, щоб гарантувати тестування функції в специфічних умовах або сценаріях.
- Socket.io: бібліотека JavaScript, яка обробляє з'єднання WebSocket. Вона автоматизує більшу частину роботи з WebSockets, а також забезпечує резервні методи, які працюють без будь-якої спеціальної конфігурації. Socket.io дозволяє виконувати оновлення в реальному часі в системі, такі як надсилання або отримання повідомлень.

2.2.3 База даних

В роботі використовуються бази даних NoSQL, оскільки

- Вони більш гнучкі [6]: ви можете отримати доступ до вкладених даних без необхідності будь-якого з'єднання.
- Вони швидші [6]: вкладені дані зберігаються там же і з ними можна звертатися без додаткових запитів.

- Вони краще масштабуються [7] при розподілі даних по різних вузлах.
- Існує багато типів баз даних NoSQL, які підходять для різних видів роботи, наприклад, Key-Value для сесій або Document-based для складних даних [8].

Спочатку в роботі було застосовано MongoDB, але згодом додатково використовувався Redis, а також зіставлялися дані сеансу з ідентифікаторами користувачів.

MongoDB - це безсхемна документаційно-орієнтована база даних.

Це дає нам можливість без особливих зусиль зберігати складні дані та отримувати їх відразу, без додаткових запитів. Ми використовуємо MongoDB для зберігання будь-яких постійних даних, таких як реквізити та налаштування користувачів, інформацію про кімнати та повідомлення в чаті.

Redis - це структура даних з ключовими значеннями, яка використовує сховище пам'яті для швидкого пошуку за заданим ключем. Запити виконуються швидше, ніж у MongoDB, але також існує більший ризик втрати даних, і він не може обробити складні значення (наприклад, вкладені документи).

Хоча Redis працює краще, ніж MongoDB, не можна застосовувати його для критичних даних. З цієї причини його використовують лише для зберігання сеансів користувачів, що у випадку втрати означало б лише те, що користувачеві доведеться повторно входити в систему, щоб продовжувати користуватися нашою платформою.

2.3 Проектування клієнтської частини

Для реалізації клієнтської частини був обраний шаблон SPA (Single-Page Application). SPA-динамічно отримують дані з API, коли користувач переглядає сайт, уникаючи оновлення всієї сторінки кожного разу, коли користувач заповнив форму або перейшов до іншої частини сайту. При цьому перше завантаження часто займає більше часу, через те, що потрібно завантажувати більший фрагмент файлу JavaScript, але після першого

завантаження затримка між операціями мінімальна, що призводить до більш оперативного обміну повідомленнями та меншого використання пропускної здатності мережі в більшості випадків.

Реалізація масштабованого додатка за допомогою JavaScript Vanilla потребувала значної кількість часу, оскільки у нього немає жодної утиліти високого рівня, яка спростила б розробку такого типу, як HTML-рендер високого рівня, дозволяє будувати елементи на льоту.

Отже, для цього було доцільно вибрати окремий фреймворк та/або бібліотеку. В результаті аналізу існуючих фрейворків: Angular, React та Vue, було обрано React як дуже потужну бібліотека з величезною екосистемою (ви можете знайти безліч утиліт, які мали використовуватись з React). Він відомий завдяки його швидкій продуктивності та малому споживанню пам'яті, що особливо корисно при орієнтації на мобільні пристрої. Більше того, на його офіційному сайті та в Інтернеті є безліч документації.

Основні функції бібліотеки:

- Tree Structure

Сторінка React завжди починається з одного кореневого компонента (деревного вузла), відображеного в попередньому HTML-елементі на сторінці.

Кожен компонент може мати одного або декількох нащадків (відомих як композиції [9]).

```
function RootComponent(props) {
return <h1>It works!</h1>;
}
ReactDOM.render(<RootComponent />, document.
getElementById('main'));
```

- Custom DOM Elements

React не працює безпосередньо з компонентами HTML. Натомість він використовує компонент, який згодом буде перекладений у компоненти HTML.

```
<input className="foo" />
<textarea value="123" />
is transpiled to
<input class="foo" />
<textarea>123</textarea>
```

Інформація передається вниз по дереву за допомогою властивостей компонента Кореневий компонент може не мати доступу до зовнішньої інформації. Однак компонентам-нащадкам може знадобитися такий доступ. Наприклад, компонент, який відповідає за відображення загального текстового поля введення разом з міткою, повинен мати доступ до назви поля.

```
Parent component
<form>
<TextInput name="email" />
<TextInput name="address" /> ...
</form>
TextInput component
function textInput(props) {
return (
<fieldset>
<label for="props.name">{props.name}</label>
<input type="text" id="{props.name}" placeholder=
"{props.name}" /> </fieldset>
);
}
```

Інформація передається вгору по дереву за допомогою посилань на функції. У якийсь момент батьківському компоненту може знадобитися доступ до інформації, яка змінилася в компоненту-нащадку, щоб певним чином відреагувати на такі зміни.

Наприклад, у фрагменті вище, можливо, знадобиться дізнатися, коли значення змінилося на вході, щоб пізніше обробити інформацію форми.

```
Parent component
function inputChanged(event) { ... }
function ParentComponent(props) {
return (
<form>
<TextInput name="email" onChange={inputChanged} />
<TextInput name="address" onChange={inputChanged} />
...
    </form>
);
}
TextInput component
function textInput(props) {
return (
<fieldset>
<label for="props.name">{props.name}</label>
<input type="text" id="{props.name}" placeholder=
"{props.name}" onChange={props.onChange(...)} />
</fieldset>
);
}
```

Крім того, нам потрібні додаткові бібліотеки.

- Babel: Користувачі, які заходять на наш сайт, можуть використовувати старі версії браузера. Щоб переконатися, що всі браузери можуть зрозуміти наш код, використовується Babel, який перетворює сучасний код JavaScript в код JavaScript, який може зрозуміти більшість браузерів.
- Redux: сховище в пам'яті для JavaScript. Сховище типу Redux дозволяє уникнути необхідності передавати дані вгору та вниз по дереву React, оскільки Redux зберігає все це в одному місці, до якого можна отримати доступ будь-коли. Він також є модульним, що робить його ідеальним для нашого застосування, оскільки допомагає досягти масштабованості. Тим не менш, Redux спрощує функції системи особливо такі, що працюють з глобальними змінними, такими як користувач, який на даний момент має автентифікацію. Redux спочатку був побудований для React. Сховища можна легко підключити до компонентів React, які матимуть доступ до будь-яких збережених даних, а також зможуть відправляти нові дії для додавання / оновлення даних у ньому.

- Enzyme: бібліотека, яка полегшує тестування компонентів React. Ця бібліотека була створена спеціально для React і дозволяє нам моделювати компоненти, якби вони були надані в DOM. Він часто використовується для тестування елементів управління, які повинні з’являтися лише за певних умов.
- Sinon: бібліотека тестування JavaScript. Буде використана для тестування Front-End.
- Express: Експрес для надання візуалізації на стороні сервера. SPA-програми не містять контенту у файлі HTML, який надсилається клієнту. Весь контент (і логіка) знаходиться в одному або багатьох фрагментах JavaScript, які може отримати сайт, а це означає, що браузери повинні завантажувати ці файли та виконувати їх, перш ніж вони зможуть відобразити контент на екрані користувача. Це може зайняти до декількох секунд, залежно від розмірів основних фрагментів JavaScript. Надаючи першу запитувану сторінку на сервері, можна заздалегідь завантажити HTML (і навіть CSS), і хоча навігація буде дуже обмежена, поки частини не завантажаться повністю, користувачі зможуть прочитати контент цієї сторінки так, ніби сторінка була повністю завантажена.

Крім того, деякі пошукові системи обмежуються читанням HTML. Оскільки весь наш контент знаходиться у файлах JavaScript, вони його не побачать. Отже, вони не братимуть до уваги жоден змістовий заголовок чи інші типи ключових слів при його індексації.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ

3.1 Структура бази даних

Основним визначальним аспектом будь-якої програми, що використовує бази даних, є її структура бази даних. Вона може змінюватись в залежності від багатьох різних факторів, таких як кількість зчитувань запису або поля, з якими користувач найбільше працює.

В роботі використовується MongoDB, яка здатна оперувати складними за структурою та значними за обсягом даними.

Наша структура даних Redis обмежується відображенням сеансів за ідентифікаторами користувачів текстового типу. Ось так працює веб-запит: Node.js запитує Redis, використовуючи ідентифікатор сеансу користувача, щоб визначити, чи існує обліковий запис користувача та його ідентифікатор. Якщо виявлено ідентифікатор облікового запису, Node.js запитує MongoDB, щоб дізнатися решту інформації про користувача.

База даних MongoDB зберігає все інше: інформацію про користувачів, кімнати, чати та повідомлення.

В результаті оптимізації база даних містить чотири сутності: користувачів (Users), кімнати (Rooms), чати (Chats) та повідомлення (Messages). Використовуючи бібліотеку Mongoose на Node.js, також було визначено гнучкі обмеження для кожної з сутностей, що задає вміст її полів у певному форматі без додаткової перевірки структури даних до або після її введення в базу даних.

3.1.1 Користувачі (Users)

Для початку системі потрібно було зберігати дані користувачів. Оскільки очікується значна кількість записів, було створено окрему таблицю для самих користувачів, де зберігається інформація, що використовується при їх автентифікації, та інші особисті дані.

Їх кімнати, чати та повідомлення повинні зберігатися іншому місці. Зважаючи на те, що очікується багато кімнат, чатів та повідомлень на кожного користувача, у цій таблиці відповідні поля відсутні.

Поля таблиці:

- _id: ідентифікатор.
- username: ім'я користувача.
- email: електронна адреса.
- password: зашифрований пароль.
- passwordResetToken: маркер для скидання пароля.
- passwordResetExpires: дата закінчення терміну дії символу скидання пароля.
- github: ідентифікатор профілю GitHub
- google: ідентифікатор профілю Google.
- tokens: перелік пов’язаних токенів послуг.
  - kind: назва послуги (тобто github)
  - accessToken: маркер доступу, наданий службою.
- profile: особисті дані
  - – name: повне ім'я.
  - – gender: стать.
  - – location: розташування
  - – website: персональна URL-адреса веб-сторінки або блогу.
  - – picture: URL-адреса аватара.
- updatedAt: останнє оновлення.
- createdAt: дата створення.

Таблиця користувачів індексується полями _id, username, email, github, and google. Вони охоплюють більшість пошукових запитів, що робляться найчастіше: користувачів шукають багато разів, тоді як їх дані практично не змінюються протягом життєвого циклу системи.

3.1.2 Кімнати (Rooms)

Зважаючи на те, що дані про кімнати не зберігалися як вкладені дані в таблиці користувачів, а в системі використовуються прямі посилання на них, було створено окрему таблицю для кімнат. Крім того, очікувалось, що буде створено багато кімнат, напевно, навіть більше, ніж користувачів.

Поля таблиці:

- _id: ідентифікатор.
- title: назва.
- slug: ідентифікатор URL-адреси кімнати.
- description: опис.
- owner: _id користувача-власника.
- isPrivate: тип кімнати – приватна чи загальна.
- members: масив user _id, які є учасниками цієї кімнати.
- updatedAt: дата зміни.
- createdAt: дата створення.

Зауважте, що в цій таблиці знову не зберігається жодних даних про чати кімнати. Інші програми для чату, які встановлюють обмеження в 10-20 чатів на кімнату, повинні розглянути можливість вбудовування всього об’єкта чату у їхній кімнаті або принаймні зберегти посилання на них. З іншого боку, в таблиці зберігаються посилання на учасників кімнати. Це тому, що не очікується більше декількох сотень користувачів на кімнату, і вони також не є чіткою сутністю, і дисковий простір, який ці посилання займають, не є значним. Інше поле, яке також зберігається за посиланням, - це власник кімнати. Причина, чому ці дані не вбудовуються в таблицю, в цьому випадку полягає в тому, що дані профілю користувача можуть часто оновлюватися, що означитиме необхідність оновлення всіх належних йому кімнат, окрім відповідного Користувача.

Індексація таблиці Кімнати проводиться за полями _id, slug, власником та учасниками.

3.1.3 Чати (Chats)

Дані про чати зберігаються в окремих таблицях. Можуть бути кімнати, в яких їхні учасники мають кілька чатів, але інші можуть мати сотні (навіть якщо це призводить до того, що у вас є кілька неактивних). Ще раз довелося проаналізувати, чи варто вбудовувати або пересилати повідомлення у таблицю чатів чи тримати їх ізольованими в іншій. Очікувалися тисячі повідомлень у будь-якому чаті, який швидко перевищить 16 Мб, що може вмістити будь-який документ MongoDB, навіть якщо будуть зберігатися тільки посилання.

Поля таблиці:

- _id: ідентифікатор
- room: ідентифікатор кімнати, до якої належить чат.
- title: назва.
- description: опис.
- github: ім'я сховища GitHub, яке враховується при створенні GitHub Конкретні чати.
- firstMessageAt: дата першого надісланого повідомлення. Вона використовується для визначення, чи користувач уже отримав усі повідомлення чату.
- lastMessageAt: дата останнього надісланого повідомлення.
- updatedAt: дата зміни.
- createdAt: дата створення.

Індексація таблиці чати виконується за полями _id та room. _id

3.1.4 Повідомлення (Messages)

В системі очікується тисячі повідомлень, тому правильний спосіб їх зберігання, згідно з офіційною документацією MongoDB, це окремі таблиці.

У ході розробок повідомлень можуть бути навіть мільйони, тому можливо, надалі доведеться розглянути можливість зміни структури даних, щоб уникнути вузьких місць в роботі системи.

Цей випадок схожий на таблиці «Кімнати» або «Чати» [11], але необхідно не просто зберігати тисячі записів в довгостроковій перспективі, а враховувати те, що їх кількість буде швидко зростати.

Можна виділити такі основні характеристики повідомлень:

1) Можливість зберігання сотень повідомлень на годину.
2) Можливість завантаження тисяч повідомлень на годину.
3) Можливість отримання повідомлень у хронологічному порядку (найновіший перший).

Зауважте, що кілька чатів одночасно, ймовірно, отримують останні повідомлення не один раз, і хоча повідомлення зберігається лише один раз, кілька учасників, ймовірно, прочитають їх декілька раз. Більше того, користувачі ніколи не хотіли б одразу отримати всі повідомлення. Користувач не тільки не міг би їх прочитати, але й система не змогла б впоратися з навантаженням, якби це було б реалізовано для чатів, що мають багато повідомлень.

Поля таблиці:

- _id: ідентифікатор.
- chat: ідентифікатор чату, до якого він належить.
- owner: ідентифікатор відправника (користувача).
- content: текст.
- type: тип вмісту.
  - language: мова коду, якщо тип повідомлення - код.
  - highlight: рядки для виділення, якщо тип повідомлення - код.
  - chat: посилання на батьківський чат, якщо повідомлення є пересланим.
- deletedAt: дата видалення. Вміст буде видалено при видаленні, але буде відомо про те, що Користувач надіслав повідомлення.
- updatedAt: дата зміни.
- createdAt: дата створення.

Ця структура була розроблена для до 1 000 000 одночасних з'єднань [12]. Індексація повідомлень виконувалася за полями _id та chat + createdAt. Перше поле допомагає шукати певне повідомлення, тоді як друге добре працює при пошуку минулих повідомлень.

3.1.5 Webhooks

Під час реалізації програми виникла необхідність зберігати дані з GitHub WebHooks.

GitHub WebHooks надсилає інформацію лише один раз на сховище, тому необхідно переконатися, що дані зберігаються таким чином, щоб будь-який чат, пов'язаний із цим сховищем GitHub, мав доступ до оновлень GitHub.

Крім того, інформацію довелося швидко отримувати та зберігати, оскільки оновлення GitHub, як правило, численні, і система повинна повідомити про них користувачів чату в режимі реального часу. Цей випадок нагадує повідомлення, хоча цього повідомлення надіслаються ботами.

Поля таблиці:

- type: тип webhook, якщо було декілька постачальників.
- uid: унікальний ідентифікатор повідомлення webhook (надіслане збереженням / оновленням).
- github: специфічні поля GitHub, такі як назва сховища чи дії.
- updatedAt: дата зміни.
- createdAt: дата створення.

Ми проіндексували _id та _id + github.repository.

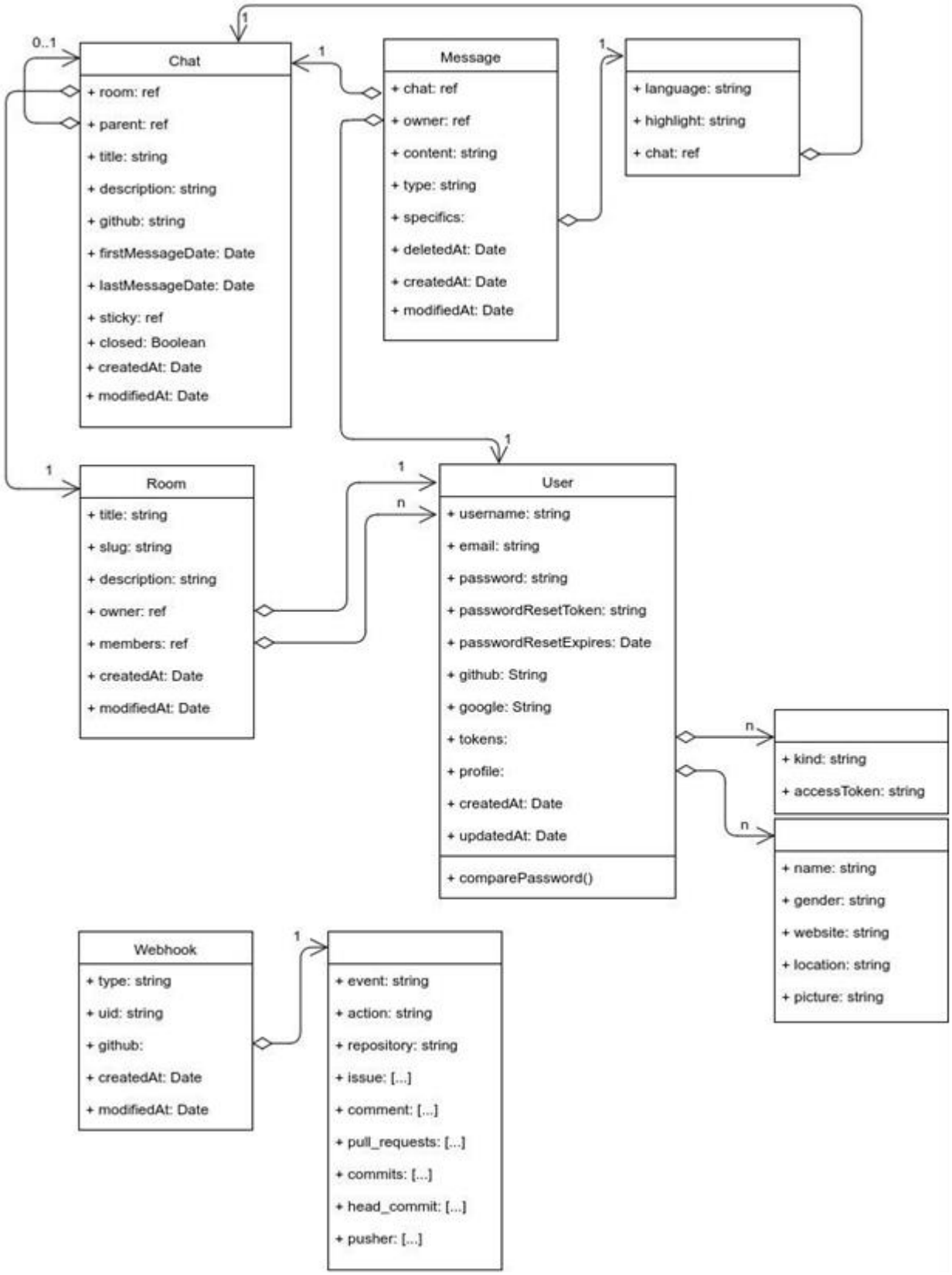


Рисунок 3.1 - Підсумкова діаграма UML бази даних MongoDB

3.2 Вибір та конфігурування програмного середовища

3.2.1 Node

Перш ніж приступити до конфігурації Node, необхідно визначити механізм обміну різними даними з клієнтом. Зважаючи на те, що клієнтська програма не є частиною Back-End, наші дані повинні мати універсальний формат. Традиційно для цього використовувався XML, але в роботі було обрано JSON як формат передачі даних з таких причин:

- Система постійно працюєз JavaScript, і об’єктні структури JavaScript дуже схожі на JSON, що дозволяє їх читати та обробляти без додаткових перетворень.
- JSON набирає все більшої популярності у веб-контексті.
- Express може працювати з JSON.

Як сказано в розділі проектування, було вирішено використовувати фреймворк Express. Express відповідає за керування сеансами користувачів, обмін інформацією з базами даних та обробку HTTP-запитів (аналіз даних, таких як x-www-formurlencoded в об’єкти JavaScript, виконання певних контролерів, коли маршрут узгоджується, і повернення значення).

Express, як повноцінний фреймворк, також може самостійно запустити веб-сервер (з усією його логікою). Але цей функціонал виконувався на сторонній бібліотеці (http), що дозволило запускати і API AJAX, і WebSockets на одному порті.

Система використовує Socket.io для роботи з WebSockets. Що стосується його конфігурації, він має доступ до даних сеансу, а також працює з бібліотекою http фреймворку Express. Пакет http запускає Express і Socket.io на одному порту.

Структура папок сервера виглядала так на самих ранніх етапах:

```
.
|-- bin
|-- config
|-- data
```

```
|-- data-session
|-- docs
|-- node_modules
`-- src
   |-- controllers
   |-- helpers
   |-- middleware
   |-- models
   |-- routes
   `—sockets
```

Врешті-решт вона дещо зросла, але, на щастя, зміни виявилися мінімальними.

`bin`: пакетні файли або інші виконувані файли.

`config`: конфігурація користувачів, які не обов’язково змінюється, наприклад, база даних налаштування або ключі програми.

`data and data-session`: папка, де MongoDB та Redis будуть зберігати свої дані. node_modules: npm-модулі.

`src`: містить всю логіку.

`src/controllers`: обробники запитів.

`src / helpers`: утиліти, які можна використовувати в будь-якій точці програми, які, як правило, використовуватимуться не один раз.

`src/middleware`: node.js express Framework може використовувати проміжне програмне забезпечення для обробки запитів, перш ніж вони надходять до контролера.

`src/models`: моделі баз даних, які були сформовані в попередньому підрозділі.

`src / sockets`: обробник з'єднання з сокетом (включаючи відправників подій та слухачів).

Папка для тести, що були створенні, відсутня. Тести зберігаються поруч із утилітою для тестування, що дозволяє уникнути переписування всього дерева папок.

```
`-- moduleFolder
|-- index.js
`-- index.test.js
```

3.2.2 Створення, читання, модифікація та видалення даних

Запити GET, POST, PUT / PATCH, DELETE є основою API RESTful, і тому вони є в програмі.

GET використовується для отримання списку документів або певного документа. URL-адреси, як правило, доволі очевидні для розрізнення, і в більшості випадків, коли потрібен конкретний ідентифікатор кімнати, чату, повідомлення або користувача. POST використовується для створення документів. Інші сайти, такі як Instagram, використовують цей метод також для оновлення та видалення даних, швидше за все, з міркувань сумісності [15]. DELETE використовується для видалення документа. PATCH використовується для оновлення документів. Вам може бути цікаво, чому використовується PATCH замість PUT. Хоча обидва призначені для оновлення документів, перше може оновити лише кілька полів, а інше - відправити цілий новий документ. Хоча PUT простіший можливість оновлення лише декількох полів моделі дуже цікава під час оновлення інформації чату. У чаті, повному людях, ми очікуємо багато оновлень, і деякі можуть надходити одночасно, тому PATCH дозволяє виконати для додавання та оновлення полів коректно [16-18].

3.2.3 React

Тепер настав час для клієнта, що базується на React. Хоча мова йде не лише про React, а скоріше про React та частину її екосистеми. React сам по собі пропонує занадто мало засобів, щоб обробляти проект такого розміру. Існує необхідність розробки комплексу додатків, які будуть читатись і записуватися

іншими частинами програми, обробляти URL-адреси та зміни шляху, отримувати з API (як HTTP, так і WebSockets) тощо.

Ось так виглядала структура папок на самих ранніх етапах:

```
.
|-- dist
|-- src
|-- components
|-- containers
|-- helpers
|-- redux
|-- redux
|-- middleware
`-- modules
|-- routes
|-- styles
`-- tests
`-- webpack
```

`dist`: файли, які призначені для розповсюдження, в основному генеруються модулем веб-пакету модулів.

Він містить index.html, який буде відправною точкою для всіх, хто має доступ до нашого сайту, та пакети JavaScript, що містять логіку програми (включаючи необхідні зовнішні бібліотеки).

`src`: містить всю логіку програми.

`src/components`: компоненти [19], які не взаємодіють з Redux.

`src/containers`: містить підключені компоненти, які взаємодіють із нашим сховищем Redux.

`src / helpers`: загальний код JavaScript. В даний час в цій папці зберігаються реалізації програми AJAX Fetching і клієнта WebSockets на основі Promise.

`src / redux`: все про Redux.

`src / redux / middleware`: Розширення Redux, наприклад, підтримка спеціальної дії об'єкта для отримання прямо з API.

`src/redux/modules`: Модулі Redux, по одному на кожну окрему тему на сайті.

`src/routes`: маршрути застосування, пов'язані з їх відповідними компонентами.

`src/styles`: містить глобальні стилі застосування.

`src/tests`: Конфігурація глобальних тестів.

`webpack`: Конфігурація веб-пакунків.

Окремі тести розташовані поруч із реалізацією. Наприклад, компонент komponent.js має компонент компонент.test.js у тій же папці.

3.3 Короткий опис програмної реалізації

3.3.1 Аутентифікація

Першою було реалізовано функцію аутентифікації, що включала локальну, GitHub та Google аутентифікації.

На стороні сервера це передбачало створення декількох нових сценаріїв для обробки входу, реєстрації (лише для локальної автентифікації) та виходу та відповідних стратегій для обробки кожного з цих сервісів.

Для локальної аутентифікації було налаштовано наступні два маршрути:

```
/auth/signup
 /auth/signin
```

Вони обробляють регістр та дані форми входу відповідно.

Для аутентифікації OAuth було налаштовано такі маршрути:

```
/auth/github
/auth/github/callback
 /auth/google
/auth/google/callback
```

Система повинна мати кінцеві точки для достовірної автентифікації OAuth. Перший - це запит, який подав користувач на сторінку аутентифікації, а також зворотній виклик від сервісів. Слід зазначити, що оскільки система

обробляє всі типи аутентифікації, навіть зворотні виклики, то JSON не повертається в жодному з маршрутів AOuth. Як виняток, система використовує переадресацію на клієнтські сторінки як тоді, коли аутентифікація пройшла успішно, так і під час помилки (або через проблему з нашої сторони, або через те, що користувач відмовився надавати нам дозволи на сторінці постачальника).

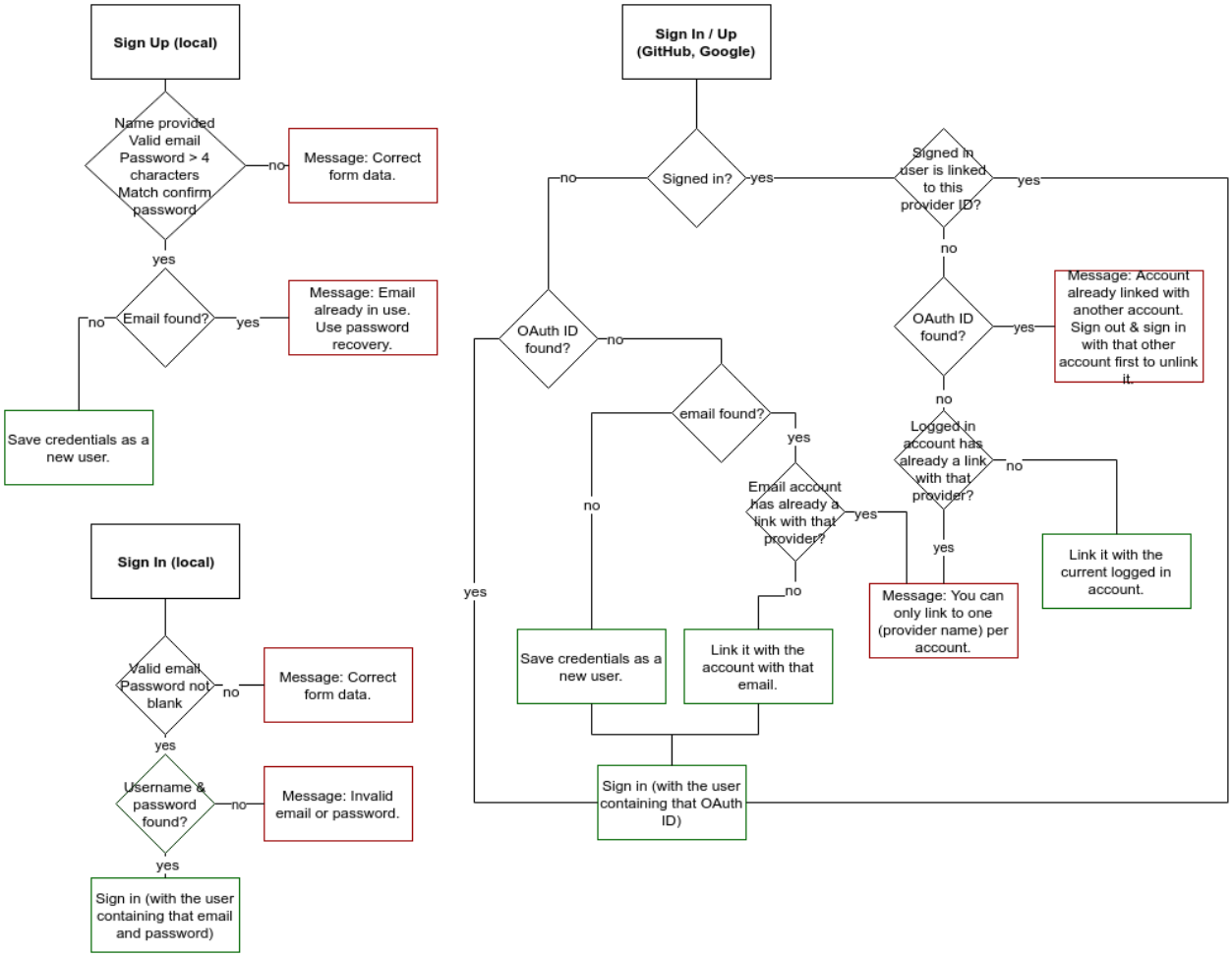


Рисунок 3.2 – Підсумкова діаграма UML бази даних MongoDB

Далі, необхідно було вирішити, як клієнт буде повідомляти сервер про вхід користувача.

Більшість сервісів API включають маркер автентифікації в параметрах запиту, але з іншого боку, більшість веб-додатків, як правило, використовують для цього файли cookie, і вони надсилають ці файли cookie на кожен запит. В роботі використовується другий варіант, оскільки система є веб-орієнтованою.

Клієнтському додатку також знадобиться конкретна функція для перевірки, чи є його поточний ідентифікатор користувача дійсним.

Що стосується логіки автентифікації, система використовує Passport за OAuth та OAuth2 в загальному API.

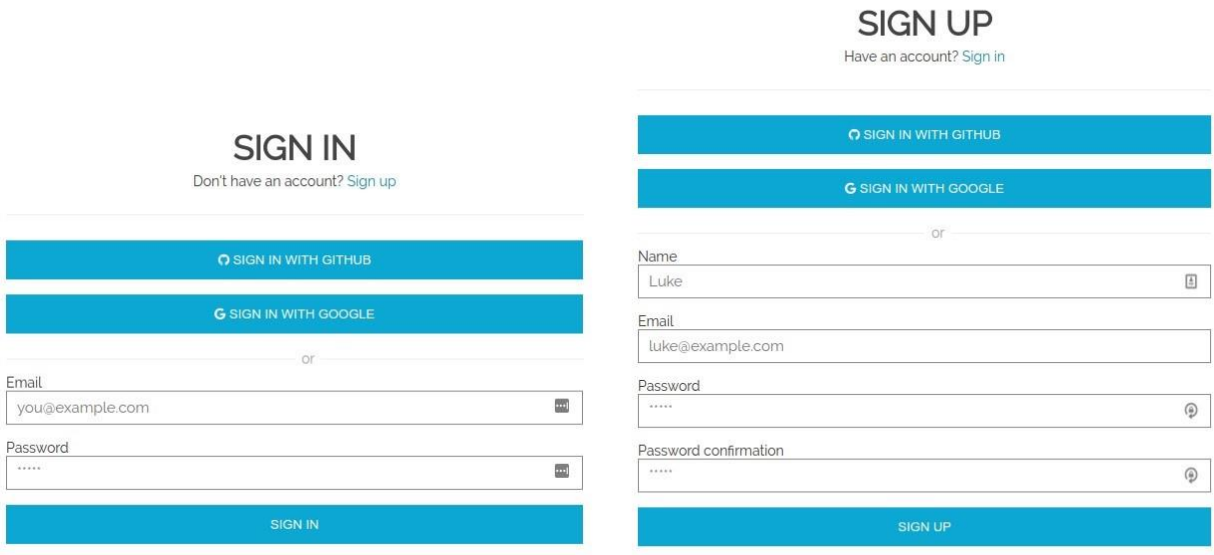
3.3.2 Клієнтська частина

Для обробки локального входу та реєстрації було створено дві різні форми (рис. 3.3), які надсилатимуть свою інформацію на сервер. Сервер перевірить їх і проінформує клієнта про будь-яку помилку, яка могла статися. Для виходу з системи також є відповідна кнопка.

Зберігання даних виконується Redux, що дозволяє проводити оновлення з будь-якого компонента системи в режимі реального часу.

Наприклад, для користувача щойно увійшов із локальної форми автентифікації, система оновить панель навігації з його іменем без будь-якого додаткового запиту.

Нижче є схема того, як виглядає наш модуль Authentication Redux.



а *б*

Рисунок 3.3 – Форми клієнтської частини: а) авторизація, б) реєстрація

Отримати всі користувацькі дані, щоб заповнити таку форму автоматично, можна за маршрутом `/ whoami`. Наша локальна аутентифікація,

яка працює з наданими користувачем електронною поштою та паролем, працює з формою, і оновлює його стан відразу після перевірки облікових даних за допомогою API

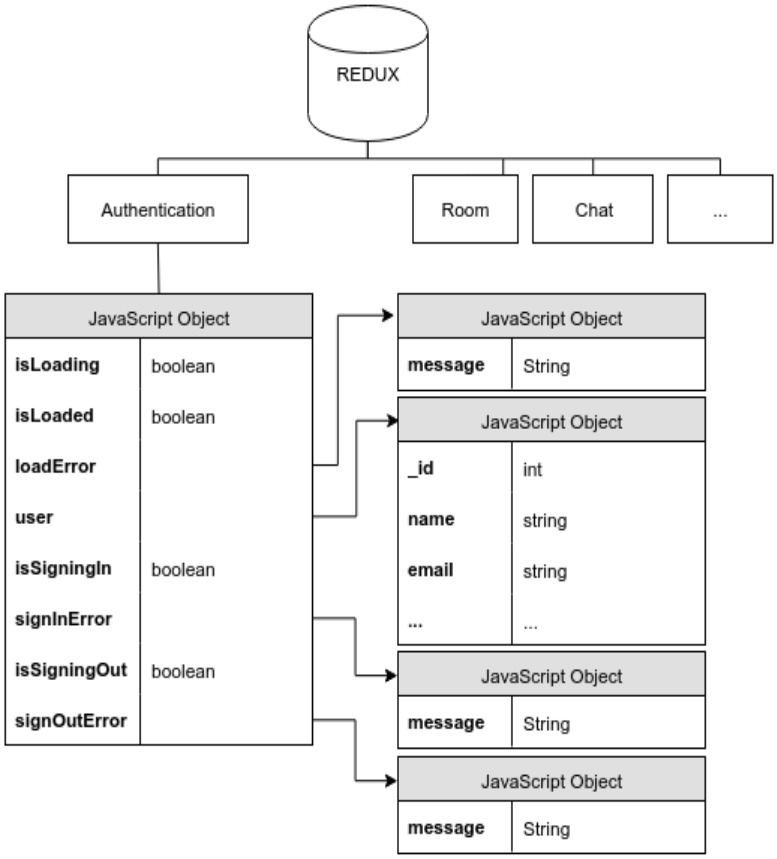


Рисунок 3.4 – Схема модуля аутентифікації Redux

3.3.3 Методи класів кімнати (Rooms), чати (Chats), повідомлення (Messages)

Після того, як частина аутентифікації була готова, настав час перейти до кімнат, чатів та повідомлень, які користувачі створюватимуть, читатимуть, оновлюватимуть та видалятимуть. Кожна операція включала в себе операції з базою даних, запит клієнта на сервер, перевірку, інтерфейс користувача (форма або сітки даних) та відображення результату кожної операції.

На стороні сервера система підтримує ці операції за допомогою методів GET, POST, PATCH та DELETE. Щоб зробити це ефективно, необхідно абстрагувати найпоширеніші операції, особливо валідаційні: перевірити, чи користувач увійшов у систему, які кімнати, чати та повідомлення йому доступні.

Таблиця 3.1 – Методи класу кімнати (rooms)

Метод	Шлях	Опис
GET	`/rooms`	Надає список кімнат
POST	`/rooms`	Створює нову кімнату
GET	`/rooms/search`	Виконує фільтрацію кімнат з необов'язковими параметрами запиту: `_id`, `slug` або `title`.
GET	`/rooms/_id`	Надає ідентифікатор конкретної кімнати
PATCH	`/rooms/:_id`	Оновлює конкретну кімнату, її ідентифікатор та інші поля кімнати.
POST	`/rooms/:_id/join`	Приєднує користувача до конкретної кімнати з наявним ідентифікатором кімнати.
POST	`/rooms/:_id/leave`	Видаляє користувача до конкретної кімнати з наявним ідентифікатором кімнати
DELETE	`/rooms/:_id`	Видаляє конкретну кімнату.
GET	`/rooms/:_id/chats`	Надає перелік чатів конкретної кімнати
POST	`/rooms/:_id/chats`	Створює новий чат конкретної кімнати
GET	`/chats/:_id`	Надає ідентифікатор конкретного чату
PATCH	`/chats/:_id`	Оновлює конкретний чат
DELETE	`/chats/:_id`	Видаляє конкретний чат
POST	`/chats/:_id/fork`	Створює розгалуження чату
POST	`/chats/:_id/fork-merge`	Об'єднує розгалуження чату
POST	`/chats/:_id/fork-upgrade`	Створює новий чат з розгалуження
GET	`/chats/:_id/messages`	Надає повідомлення конкретного чату
POST	`/chats/:_id/messages`	Додає повідомлення в конкретний чат

Таблиця 3.3 – Методи класу повідомлення (Messages)

Метод	Шлях	Опис
PATCH	`/messages/:_id`	Редагує повідомлення за наявним ідентифікатором.
DELETE	`/messages/:_id`	Видаляє повідомлення за наявним ідентифікатором.

Клієнт використовує ці методи API та запитує або відображає відповідні дані для створення чи читання різних об'єктів системи, використовуючи форми або таблиці даних.

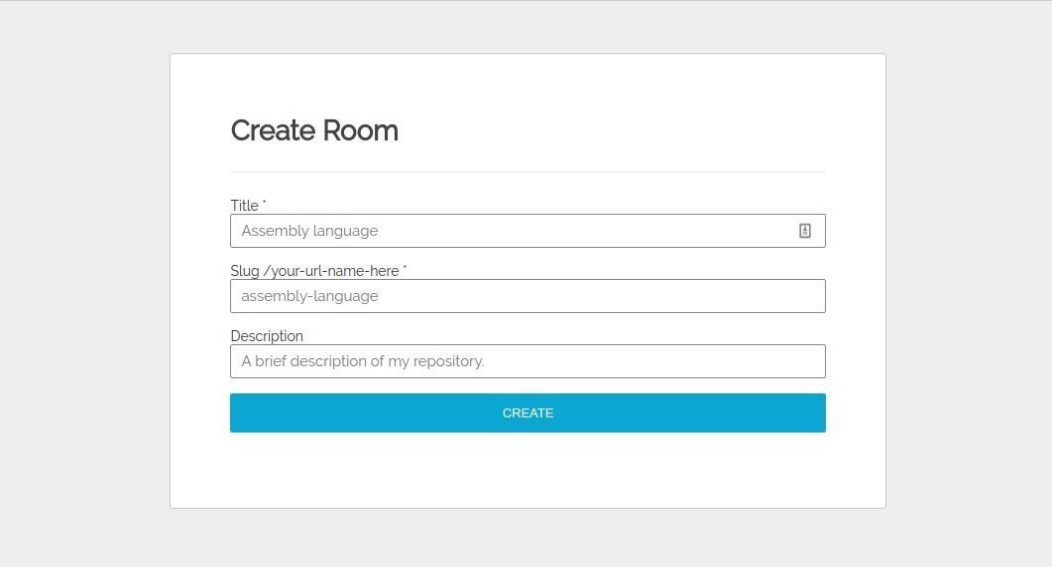


Рисунок 3.5 – Форма створення кімнати

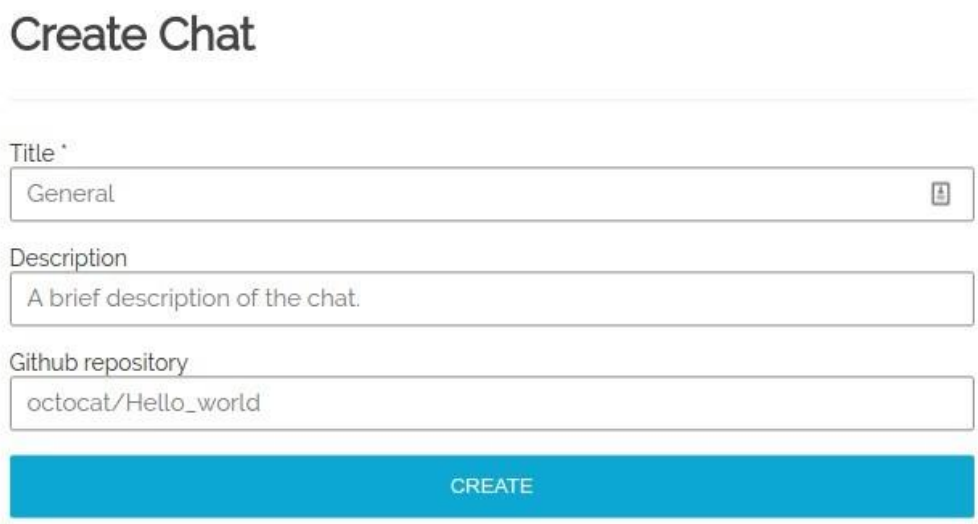


Рисунок 3.6 – Форма створення чату

3.3.4 Обмін повідомленнями

Система обміну повідомленнями використовує WebSockets. WebSockets дозволяють підтримувати в режимі реального часу, низької затримки та двонаправленого зв’язку між клієнтом та сервером. Стійке двостороннє з'єднання - це те, що дає змогу отримувати серверні повідомлення в будь-який

час. Це також дозволяє уникнути необхідності надсилати зайві заголовки HTTP з кожним запитом, що може помітно змінити використання пропускної здатності мережі [21].

З боку сервера використовується Socket.io - бібліотека WebSockets високого рівня. Вона пропонує обмежений функціонал у порівнянні з тим, що робить AJAX, і в основному призначена для використання для надсилання / отримання оновлень у реальному часі. Принцип DRY [20] допоміг позбутися дублюваної логіки між нашими WebSockets та AJAX. Дублювання відбувалося з подій WebSockets, які виконували так само, як і аналогічні маршрути AJAX, наприклад, надсилання повідомлень [22].

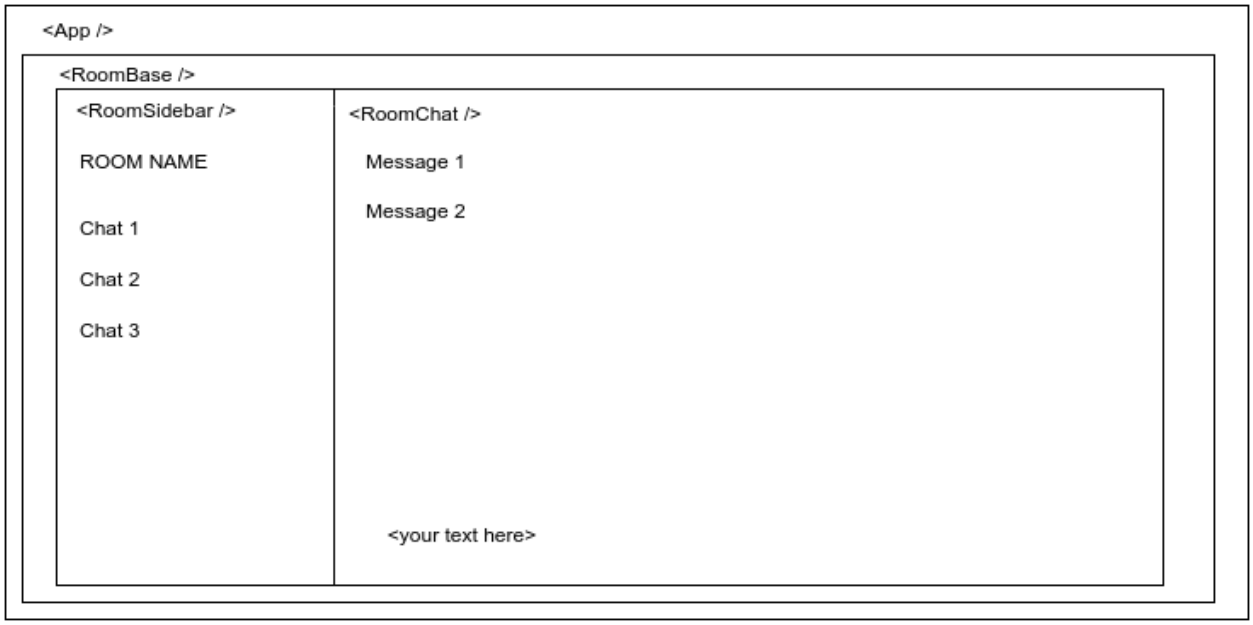


Рисунок 3.7 – Огляд компонентів React

3.3.5 Оновлення кімнат (Room) і чатів (Chat)

Для оновлення кімнат і чатів в системі повторно використовується сокет-з’єднання. В такий спосіб, наприклад, одразу буде отримано нову назву чи оновлення опису чату або кімнати.

Система коритується встановленим з'єднанням, щоб оновити деякі компоненти кімнати або чату в режимі реального часу, а не створює нові для перевірки оновлень час від часу.

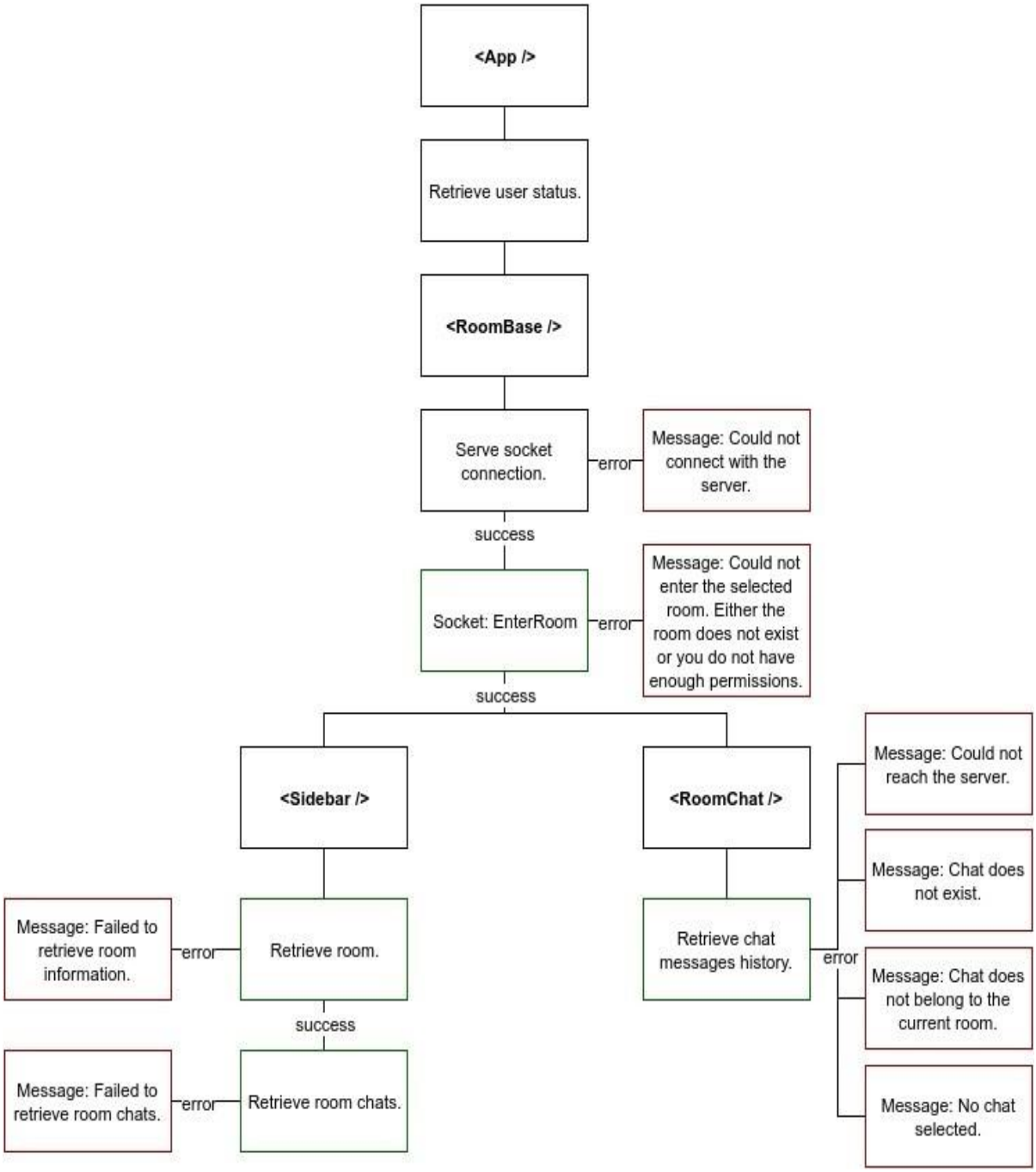


Рисунок 3.8 - Діаграма дій реагування в кімнаті

3.3.6 Закріплене повідомлення

Закріплене повідомлення - це повідомлення, яке завжди відображається у верхній частині чату для підвищення його видимості. Для розробників це особливо важливо, оскільки надає можливість обговорювати конкретний фрагмент, не потребуючи постійно прокрутки повідомлення в чаті вгору та вниз. Використовуючи закріплене повідомлення, вони можуть обговорювати

його, маючи весь час на екрані відповідну коду. На сервері інформація про закріплене повідомлення зберігалася як окреме поле у базі даних чату.

Клієнтська частина дозволяла визначити повідомлення, яке необхідно було закріпити та місце, де його слід відображати. Для реалізації було створено додатковий компонент React, ScrollContainer, що діяв незалежно від компонента ChatHistory.

Клієнтська сторона надсилає та отримує оновлення щодо закріпленого повідомлення в режимі реального часу через WebSockets, використовуючи той же самий механізм, що і для оновлення чату, оскільки, закріплені повідомлення є його частиною.

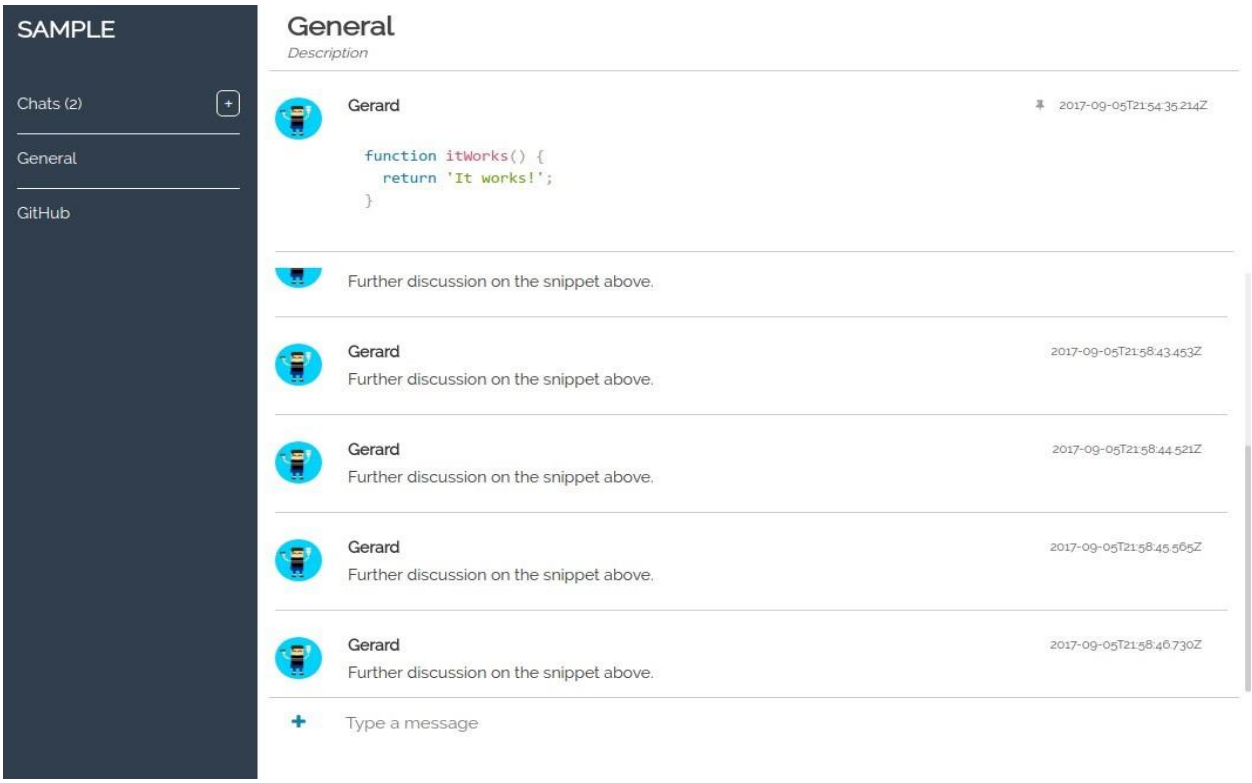


Рисунок 3.9 – Закріплені повідомлення

3.4 Тестування

Тестування вихідного коду, відоме як динамічне тестування, є різновидом методики тестування програмного забезпечення, за допомогою якого аналізується динамічна поведінка коду. Тестування вихідний коду зменшує можливість збоїв після зміни його частин, навіть якщо вони безпосередньо не пов'язані з внесеними нами нещодавно модифікаціями.

Тести є необхідною умовою безпечного рефакторингу, оскільки це часто означає, що в багатьох частинах програми використовується зміна великої кількості коду. Залежності, параметри та навіть реалізація функції можуть змінюватися, що може призвести до того, що програма не працює так, як слід, навіть якщо програма збирається без помилок.

Існує кілька десятків типів тестів, щоб переконатися, що програма повністю працює як очікувалося. В роботі застосовувалися тести Unitar, Integration та E2E, які є найпоширенішими та які виявились достатніми для більшості проектів.

3.4.1 Front end тестування

Front end тести оцінюють чи компоненти та API, які є частиною клієнта, працюють так, як очікувалося. Додаток React-Redux-Router налаштовано багатьма компонентами, але в роботі основну увагу було зосереджено на тести щодо Redux. Він генерує новий стан після обробки об'єкта. В деяких випадках необхідно перевірити, що початковий стан об’єкту порожній, в деяких інших - щоб об’єкт мав певний стан. Крім того тестувалися обробник подій об’єктів, що кінцевому підсумку генерує нові стани Redux. Найпоширеніший - це обробник кліків. Enzyme, бібліотека тестування JavaScript для React, може імітувати дії користувача, відтворюючи вміст React.

3.4.2 Back end тестування

Back end тести полягають у тому, щоб переконатися, що API правильно обробляє, зберігає та повертає відповіді JSON користувачеві (через AJAX або WebSockets). Наша експрес-архітектура розділена на 4 частини: моделі, послуги, контролери та маршрутизатор. Кожен з них мав власний набір одиничних тестів. Крім того, у нас були тести інтеграції E2E.

Далі наведено фрагмент одиничного тесту для перевірки імені користувача:

```
import { expect } from 'chai';
import { isUsername, isPassword } from '../validate';
describe('Model: User (validate)', () => {
it('should be invalid if username length is not
```

```
between 5-20', () => {
expect(isUsername('x')).to.be.false; expect(isUsername
('x'.repeat(21))).to.be.false;  expect(isUsername('x'.
repeat(5))).to.be.true;
});
```

3.4.3 Інтеграція та E2E

Нарешті, ми використовували комбінацію тестів на інтеграцію та E2E для перевірки кожного маршруту, на яких користувач може надсилати запити та очікувати відповіді на нього. Для виконання цих тестів необхідно було запустити екземпляр сервера. Кожен з тестів надсилав один чи більше запитів на цей сервер та аналізував відповідь, імітуючи поведінку користувача (або нашого React-клієнта). Для тестів використовувався AJAX бібліотека SuperAgent, яка працює на Node.js платформі і імітує роботу браузера

Один з наших тестів E2E для аутентифікації виглядає наступним чином:

```
function signin(email = 'demo@example.com', password =
 'password') {
return new Promise((resolve, reject) => {
request
.post(`${server}/auth/signin`)
.send({ email, password })
.end((err, res) => { if (err) return reject(err);
return resolve(res);
});
});
}
it('should log in with the right credentials', (done)
=> {
chain
.then(() => signin())
.then(() => { request
.get(`${server}/users/whoami`)
.end((err, res) => { expect(res.status).to.equal(200);
 done();
});
});
});
```

ВИСНОВКИ

В ході випускної роботи було проведено розробку та програмну реалізацію інформаційної системи для обміну повідомленнями між розробниками програмного забезпечення в реальному часі. При цьому було виконано такі завдання:

1. Обрано методологію проектування інформаційної системи
2. Спроектовано серверну частину інформаційної системи
3. Спроектовано серверну частину клієнтську частину інформаційної системи
4. Визначено структуру бази даних інформаційної системи
5. Обрано програмне середовище для реалізації інформаційної системи
6. Програмно реалізувано модуль аутентифікації користувача, модуль обміну текстовими повідомленнями, модуль обміну частинами програмного коду інформаційної системи:
7. Протестовано інформаційну систему

СПИСОК ЛІТЕРАТУРИ

1. Webrtc vs websockets. – http://stackoverflow.com/a/18825175
2. Webrtc samples.– https://webrtc.github.io/samples/
3. User stories: An agile introduction. http://www.agilemodeling.com/ artifacts/userStory.htm.
4. Making the switch from making the switch from node.js to golang. http://blog.digg.com/post/141552444676/ making-the-switch-from-nodejs-to-golang
5. Sinon - best practices for spies, stubs and mocks. – https://semaphoreci.com/community/tutorials/best-practices-for-spies-stubsand-mocks-in-sinon-js
6. Why is nosql faster than sql?.– http://softwareengineering. stackexchange.com/questions/175542/why-is-nosql-faster-than-sql
7. Why nosql. – http://softwareengineering.stackexchange.com/ questions/175542/why-is-nosql-faster-than-sql
8. Exploring the different types of nosql databases. – https://www. 3pillarglobal.com/insights/exploring-the-different-types-of-nosql-databases
9. Composition vs inheritance - react. – https://facebook.github.io/react/ docs/composition-vs-inheritance.html
10. Mongodb manual 3.4. – https://docs.mongodb.com/manual/reference/ limits/
11. 6 rules of thumb for mongodb schema design. – http://blog.mongodb.org/post/87200945828/ 6-rules-of-thumb-for-mongodb-schema-design-part-1
12. Scaling secret: Real-time chat. – https://medium.com/always-be-coding/ scaling-secret-real-time-chat-d8589f8f0c9b#.m5jigxq6x
13. Analysis of json use cases. – https://blogs.oracle.com/xmlorb/entry/ analysis_of_json_use_cases

14. Crud cycle (create, read, update and delete cycle). – http://searchdatamanagement.techtarget.com/definition/CRUD-cycle
15. About native xmlhttp. – https://msdn.microsoft.com/en-us/library/ms537505(v=vs.85).aspx
16. Please. don’t patch like an idiot.– http://williamdurand.fr/2014/02/14/please-do-not-patch-like-an-idiot/
17. Rfc 5789 - patch method for http. – https://tools.ietf.org/html/rfc5789 Perfomance tips | google cloud platform. – https://cloud.google.com/storage/docs/json_api/v1/how-tos/performance#patch
18. Container vs component? – https://github.com/reactjs/redux/issues/756#issuecomment-141683834,
19. Hunt D. T. A. The Pragmatic Programmer: From Journeyman to Master. Addison-Wesley Professional, 2019.– 352 p.
20. Html5 websocket: A quantum leap in scalability for the web. – http://www. websocket.org/quantum.html
21. Building a scalable node.js express app. – https://medium.com/@zurfyx/building-a-scalable-node-js-express-app-1be1a7134cfd
22. Strategy design pattern. – https://sourcemaking.com/design_patterns/strategy
23. Dynamic testing. – https://www.tutorialspoint.com/software_testing_dictionary/dynamic_testing.htm

ДОДАТОК

app.jsx

```
import React from 'react';
import { render, unmountComponentAtNode } from 'react-dom';
import { Provider } from 'react-redux';
import { Router, browserHistory } from 'react-router';
import { syncHistoryWithStore } from 'react-router-redux';

import SocketClient from './helpers/SocketClient';
import ApiClient from './helpers/ApiClient';
import configureStore from './redux/create';

const socketClient = new SocketClient();
const apiClient = new ApiClient();

const initialState = {};
const store = configureStore(initialState, socketClient, apiClient);
const history = syncHistoryWithStore(browserHistory, store);

const mountPoint = document.getElementById('main');

const renderApp = () => {
  // Import global styles. Specific styles can be found inside ./src/components
  // or ./src/containers.
  // require('../node_modules/font-awesome/css/font-awesome.min.css');
  require('./styles/global.scss');

  const routes = require('./routes').default;

  render((
    <Provider store={store}>
      <Router history={history}>
        {routes}
      </Router>
    </Provider>
  ), mountPoint);
};

// development
if (module.hot) {
  const reRenderApp = () => {
    try {
      renderApp();
    } catch (error) {
      const RedBox = require('redbox-react').default;

      render(<RedBox error={error} />, mountPoint);
    }
  };

  module.hot.accept('./routes', () => {
    setImmediate(() => {
      // Preventing the hot reloading error from react-router
```

```
      unmountComponentAtNode(mountPoint);
      reRenderApp();
    });
  });
}

renderApp();
```

global.scss

```
/* Loaded on application boot, not component / container dependent. */

@import url('https://fonts.googleapis.com/css?family=Raleway|Source+Sans+Pro');
@import 'variables';

html {
  height: 100%;
}

body {
  color: $black3;
  margin: 0;
  padding: 0;
  min-height: 100%;
  font-family: 'Raleway', sans-serif;
  font-size: 14px;
}

h2 {
  text-transform: uppercase;
}

a {
  &, &:link, &:visited, &:active {
    color: $blueMedium1;
    text-decoration: none;
  }

  &:hover {
    color: $blueLight1;
    text-decoration: none;
    cursor: pointer;
  }
}

:global a.no-decorate {
  &, &:link, &:visited, &:active {
    color: $black5;
  }
}

input, textarea, select {
  background-color: $whiteLight1;
  border: 1px solid #888;
  font-family: 'Raleway', sans-serif;
  font-size: 1.1em;
  font-weight: 200;
  padding: 7px 10px 7px 10px;
  width: 100%;
```

```
  transition: border 0.2s ease;
}

textarea:focus, input:focus, select:focus {
  outline: none;
  box-shadow: none;
  border: 1px solid $blueLight1;
}

:global .CodeMirror {
  border: 1px solid #888;
}

button {
  color: $black3;
  background-color: #9ecadd;
  border: 0;
  border-radius: 2px;
  box-shadow: none;
  padding: 5px;
  transition: background-color 0.2s ease;

  &:focus {
    outline: none;
  }

  &:hover {
    background-color: $greenLight1;
    cursor: pointer;
  }
}

:global button.transparent, :global a.transparent-button {
  background-color: $whiteLight1;
  border: 2px solid $black5;

  &:hover {
    border: 2px solid $blueLight1;
  }
}

:global .big-transparent-button {
  background-color: transparent;
  border: 3px solid $whiteLight2;
  border-radius: 10px;
  color: $whiteLight1;
  height: 4em;
  font-size: 0.6em;
  font-weight: 200;
  width: 100%;
  transition: background 0.2s ease,
              color 0.2s ease;

  &:hover {
    background-color: $whiteLight2;
    color: $blueLight1;
  }
}
```

```
:global .full-center {
  position: absolute;
  top: 50%;
  left: 50%;
  transform: translate(-50%, -50%);
}

:global .row {
  display: flex;
  flex-direction: row;
}

:global .modal {
  position: fixed;
  background-color: rgb(0,0,0);
  background-color: rgba(0,0,0,0.4);
  top: 0;
  left: 0;
  height: 100%;
  width: 100%;
  z-index: 42;

  > span.close {

  }

  > div {
    @extend .full-center;
    background-color: $whiteLight1;
    padding: 25px;
    min-width: 30%;
  }
}

:global a.underline {
  text-decoration: none;

  &::after {
    display: block;
    content: "";
    background-color: currentColor;
    height: 2px;
    width: 0;
    margin-top: 1px;
    transition: all 0.2s ease;
  }

  &:hover::after {
    width: 100%;
  }
}

:global ul.horizontal-list {
  display: inline-block;
  list-style-type: none;
  margin: 0;
  padding: 0;
```

```
  li {
    display: inline-block;
    list-style-type: none;
  }

  li + li {
    margin-left: 5px;
  }
}

:global .error {
  color: $redError;
}

:global .form-container {
  padding: 20px;

  header {
    border-bottom: 1px solid $greyExtreme;
    margin-bottom: 25px;
    text-align: center;

    h2 {
      font-size: 2.4em;
      color: $black4;
    }
  }

  header.with-subtitle {
    h2 {
      margin-bottom: 5px;
    }

    padding-bottom: 25px;
  }

  button {
    background-color: $blueLight1;
    color: $whiteLight1;
    border-radius: 0;
    padding: 12px 15px;
    width: 100%;
    text-align: center;
    text-transform: uppercase;
    transition: background-color 0.2s ease;

    &:hover {
      background-color: #2ecc71;
    }
  }

  .field-container {
    margin-top: 15px;
  }

  .error {
    display: block;
  }
}
```

```
:global .form-container-2 {
  @extend .form-container;

  header {
    text-align: left;

    h2 {
      font-size: 2em;
      text-transform: capitalize;
    }
  }

  input, button {
    border-radius: 2px;
  }
}

:global .form-container-modal {
  @extend .form-container-2;

  h2 {
    margin-top: 0;
  }
}

:global .spacer {
  background-color: $greyExtreme;
  width: 100%;
  height: 1px;
  text-align: center;
  margin-top: 25px;

  > span {
    background-color: $whiteLight1;
    color: $black9;
    position: absolute;
    margin-top: -8px;
    padding-left: 7px;
    padding-right: 7px;
  }
}

:global .page {
  padding-top: $headerHeight;
  min-height: calc(100% - 60px);
  .box {
    margin-top: 5%;
  }
}

:global .content {
  margin-left: auto;
  margin-right: auto;
  padding: 15px;
  max-width: 1100px;
}
```

```
:global .box {
  border: 1px solid $grey;
  border-radius: 3px;
  background-color: $whiteLight1;
  padding: 40px;
  width: 60%;

  h2 {
    margin-top: 0;
  }
}

:global .card-list {
  margin: 15px 0 0 0;
  padding: 0;

  &::after {
    content: '';
    display: inline-block;
    width: 100%;
  }

  .card {
    display: inline-block;
    width: 23.5%;
    margin-right: 2%;
    vertical-align: top;

    &:nth-child(4n) {
      margin-right: 0;
    }
  }
}

:global .card {
  text-align: left;
  border-top: 1px solid $black3;
  height: 200px;

  > span {
    display: block;
    word-wrap: break-word;
  }

  .title {
    color: $black3;
    font-size: 2em;
    margin-bottom: 1px;
  }

  .subtitle {
    font-weight: bold;
    margin-bottom: 10px;
  }

  .description {}
}
```

```
:global .vertical-fold {

  &:hover {
    > ul {
      display: inherit;
    }
  }
    padding-left: 0;
    transition: all 0.2s ease;
  }

  li {
    list-style-type: none;
    padding: 15px;
    font-size: 1.5em;
  }
}

:global .react-contextmenu {
  background-color: $grey;
  border: 1px solid $black5;
  border-radius: 1px;
  padding: 4px 4px;

  > .react-contextmenu-item {
    color: $black4;
    padding: 3px 7px;
    border-radius: 3px;

    &:hover {
      background-color: $whiteLight4;
      color: $black3;
      cursor: pointer;
    }
  }

  > .react-contextmenu + .react-contextmenu-item {
    margin-top: 5px;
  }
}
```