

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»
(назва навчального закладу)

Кафедра ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ

Дисципліна «Технології розроблення програмного забезпечення»

Курс 3 Група ІА-12 Семестр 5

ЗАВДАННЯ

на курсову роботу студента

Кузнецов Євген Олександрович

(прізвище, ім'я, по батькові)

1. Тема роботи: «Office communicator»
2. Строк здачі студентом закінченої роботи:
3. Вихідні дані до роботи: тема «Office communicator», опис програми та основних можливостей:
Мережевий комунікатор для офісу повинен нагадувати функціонал програми Skype з можливостями голосового / відео / конференц-зв'язку, відправки текстових повідомлень, веденням організованого списку груп / контактів.
4. Зміст розрахунково – пояснювальної записки (перелік питань, що підлягають розробці): огляд існуючих рішень, загальний опис проекту, вимоги до застосунку проекту, сценарії використання системи, концептуальна модель системи, вибір бази даних, вибір мови програмування та середовища розробки, проектування розгортання системи, структура бази даних, архітектура системи та інструкція користувача.

Додатки:

Додаток А - діаграма класів; Додаток Б - код проекту.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

Діаграма класів, діаграма розгортання, діаграма прецедентів, скріншоти фрагментів коду, скріншоти графічного інтерфейсу системи.

6. Дата видачі завдання: 22.09.2023

КАЛЕНДАРНИЙ ПЛАН

№, п/п	Назва етапів виконання курсової роботи	Строк виконання етапів роботи	Підписи або примітки
1.	Видача теми курсової роботи	22.09.2023	
2.	Загальний опис проекту	25.09.2023	
3.	Огляд існуючих рішень	03.10.2023	
4.	Визначення вимог до системи	10.10.2023	
5.	Визначення сценаріїв використання	15.10.2023	
6.	Концептуальна модель системи	30.10.2023	
7.	Вибір БД	07.11.2023	
8.	Вибір мови програмування та IDE	07.11.2023	
9.	Проектування розгортання системи	15.11.2023	
10.	Структура БД	18.11.2023	
11.	Визначення специфікації системи	23.11.2023	
12.	Вибір та обґрунтування патернів проектування	25.11.2023	
13.	Реалізація проекту	30.12.2023	
14.	Захист курсової	01.01.2024	

Студент _____
(підпис)

_ Євген Кузнецов_
(Ім'я ПРІЗВИЩЕ)

Керівник _____
(підпис)

Валерій Колеснік _____
(Ім'я ПРІЗВИЩЕ)

«____» _____ 20__ р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»
Кафедра інформаційних систем та технологій

Тема Office communicator

Курсова робота

З дисципліни «Технології розроблення програмного забезпечення»

Керівник

доц. Колеснік В. М.

«Допущений до захисту»

(Особистий підпис керівника)

« » _____ 2023р.

Захищений з оцінкою

(оцінка)

Члени комісії:

(особистий підпис)

(особистий підпис)

Виконавець

ст. Кузнецов Є. О.

залікова книжка № ІА –1319

гр. ІА-12

(особистий підпис виконавця)

« » _____ 2023р.

(розшифровка підпису)

(розшифровка підпису)

ЗМІСТ

ВСТУП.....	5
1 ПРОЄКТУВАННЯ СИСТЕМИ	6
1.1. Огляд існуючих рішень	6
1.2. Загальний опис проєкту.....	7
1.3. Вимоги до застосунків системи	8
1.3.1. Функціональні вимоги до системи	8
1.3.2. Нефункціональні вимоги до системи	8
1.4. Сценарії використання системи.....	9
1.5. Концептуальна модель системи.....	13
1.6. Вибір бази даних.....	18
1.7. Проєктування розгортання системи	19
1.8. Проєктування розгортання системи	19
2 РЕАЛІЗАЦІЯ КОМПОНЕНТІВ СИСТЕМИ.....	21
2.1. Структура бази даних.....	21
2.2. Архітектура системи.....	22
2.2.1. Специфікація системи	22
2.2.2. Вибір та обґрунтування патернів реалізації.....	23
2.3. Інструкція користувача	33
ВИСНОВКИ	38
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ	39
ДОДАТКИ	40
ДОДАТОК А.....	40
ДОДАТОК Б	42

ВСТУП

Office Communicator є інноваційним інструментом для внутрішньоофісного комунікування, спроектованим з урахуванням потреб корпоративного середовища. Зокрема, цей мережевий комунікатор розроблений для оптимізації комунікації та обміну інформацією в офісному середовищі, де ефективність, безпека та гнучкість грають важливу роль.

Комунікація у реальному часі має включати можливості голосового та відеозв'язку для ефективного спілкування між співробітниками. Це забезпечує невідкладний обмін інформацією та сприяє вирішенню завдань, особливо в ситуаціях, коли особиста зустріч не є можливою.

Окрім голосового та відео зв'язку, мережевий комунікатор повинен мати зручний інтерфейс для обміну текстовими повідомленнями. Це створює можливість швидко передавати короткі повідомлення або запитання, що дозволяє зберегти час та легко здійснювати спілкування.

Мережевий комунікатор повинен дозволяти користувачам організовувати свої контакти в групі. Це спрощує легкий доступ до потрібних контактів та швидке вибирання для викликів або обміну повідомленнями, забезпечуючи систематизацію комунікації.

Високий рівень безпеки є пріоритетною задачею. Мережевий комунікатор повинен гарантувати захист конфіденційної інформації, яка передається через нього, забезпечуючи безпечну комунікацію між співробітниками.

Мережевий комунікатор повинен бути інтегрований з іншими офісними інструментами, такими як електронна пошта, календар, тощо. Це забезпечує зручність використання та обмін інформацією без необхідності переходити між різними платформами.

Система повинна надавати користувачам можливість налаштовувати інтерфейс та функціонал відповідно до їхніх потреб. Це забезпечує індивідуалізацію та зручність використання комунікаційного інструменту для кожного користувача.

Office Communicator, таким чином, створює зручне та безпечне середовище для комунікації в офісі, підтримуючи сучасні стандарти функціональності та забезпечуючи відповідь на потреби корпоративного сектору в ефективній інтернет-комунікації.

1 ПРОЄКТУВАННЯ СИСТЕМИ

1.1. Огляд існуючих рішень

У цьому розділі ми розглянемо існуючі рішення в області офісних комунікаторів, спрямованих на поліпшення внутрішньоофісного спілкування та обміну інформацією.

Microsoft Teams:

Microsoft Teams є одним з провідних офісних комунікаторів, розробленим для покращення комунікації та співпраці в офісному середовищі. Він надає можливості голосового та відеозв'язку, обміну текстовими повідомленнями, проведення онлайн-конференцій, а також інтеграцію з іншими інструментами Microsoft, такими як Word, Excel, та SharePoint.

Slack:

Slack — це інший популярний офісний комунікатор, призначений для полегшення спілкування та обміну інформацією в команді. Він пропонує групові чати, можливість створення каналів для різних проєктів, інтеграцію з багатьма іншими службами та додатками.

Zoom:

Хоча Zoom в першу чергу відомий як платформа для відеоконференцій, він

також має можливості обміну повідомленнями та спрощеного спілкування між користувачами. Він широко використовується для віддалених зустрічей і комунікації в офісному середовищі.

Cisco Webex Teams:

Cisco Webex Teams є іншим інтегрованим рішенням для внутрішньоофісного спілкування. Він пропонує можливості чатів, відеозв'язку, обміну файлами та інтеграцію з іншими бізнес-інструментами Cisco.

Telegram for Business:

Telegram також входить до категорії офісних комунікаторів і має власний варіант для бізнесу. Він пропонує групові чати, канали, голосовий та відеозв'язок, а також можливість надсилати файли.

Враховуючи існуючі рішення, слід відзначити, що багато з них акцентують увагу на зручності використання, інтуїтивності інтерфейсу та інтеграції з іншими інструментами для покращення продуктивності в офісному середовищі.

1.2. Загальний опис проєкту

Метою даної курсової роботи є створення Office communicator клієнт-серверної системи з використанням веб-сокетів для обміну повідомленнями в реальному часі між веб-браузером і сервером. Це дозволить користувачам вступати в чат, тим самим його створюючи, налаштовувати свої імена, та спілкуватися, відправляючи прості повідомлення.

Проект передбачає використання таких патернів програмування, як strategy, adapter, abstract factory, bridge, composite, client-server, які допоможуть створити гнучку та масштабовану систему, але в ході розробки було прийнято рішення замінити деякі патерни на інші. В основі проєкту лежить використання Spring Framework та WebSocket для реалізації взаємодії клієнт-сервер.

Структура буде поділена на кілька основних рівнів:

Взаємодія з клієнтом: Використання веб-сокетів для забезпечення двостороннього зв'язку між клієнтом та сервером у реальному часі. Клієнтська частина буде реалізована з використанням сучасних технологій веб-розробки.

Бізнес-логіка: Обробка повідомлень, управління станом сесії, виконання команд та інші функції бізнес-логіки.

Кожен з цих рівнів буде ретельно спроектований для забезпечення надійності, високої продуктивності та зручності користувача. Система буде розроблена таким чином, щоб легко адаптуватися до змінних вимог та масштабуватися залежно від потреб користувачів.

Таке рішення дозволить створити потужний інструмент для комунікації в офісі, який буде корисним для тих, які прагнуть ефективно обмінюватися інформацією у реальному часі.

1.3. Вимоги до застосунків системи

1.3.1. Функціональні вимоги до системи

Система Office communicator клієнт-сервер повинна відповідати наступним функціональним вимогам:

- Автентифікація користувачів: Користувачі повинні мати можливість реєструватися і входити в систему, вказуючи своє ім'я або псевдонім.
- Створення чатів: Користувачі можуть створювати чати.
- Відправлення та отримання повідомлень: Система має дозволяти користувачам відправляти та отримувати повідомлення у реальному часі.
- Особиста переписка: Створення особистих переписок між користувачами.
- Відображення мережевого статусу: користувачі бачать тільки тих хто знаходиться онлайн.

1.3.2. Нефункціональні вимоги до системи

Нефункціональні вимоги до клієнт-серверної системи включають:

- Зручний та інтуїтивний інтерфейс: Інтерфейс користувача повинен бути

чітким, зрозумілим і простим у використанні.

- Висока надійність та стабільність системи: Система повинна бути стабільною та надійною у використанні, з мінімальними збоями та помилками.
- Гнучкість та масштабованість: Система повинна бути спроектована таким чином, щоб легко адаптуватися до змінних вимог і збільшення кількості користувачів.
- Безпека даних користувачів: Забезпечення конфіденційності та безпеки персональних даних користувачів, а також зберігання історії чатів та повідомлень.

1.4. Сценарії використання системи

Таблиця 1.1 – Сценарій використання «Реєстрація користувача»

Назва	Реєстрація користувача
Передумови	Користувач зайшов на сервер, бажає увійти в акаунт
Постумови	Користувач використовує форму входу
Сторони, що взаємодіють	Користувач, система
Опис	Цей сценарій описує реєстрацію користувача до office communicator
Основний потік подій	1. Система запитує ім'я користувача 2. Користувач вводить ім'я 3. Система перевіряє, чи ім'я було введено, після чого користувач є зареєстрований

Виняткові ситуації	Відсутні
Примітки	Відсутні

Таблиця 1.2 – Сценарій використання «Створення та приєднання до чату»

Назва	Створення та приєднання до чату
Передумови	Користувач зареєструвався
Постумови	Користувач успішно створив та приєднався до чату
Сторони, що взаємодіють	Користувач, система
Опис	Цей сценарій описує створення та приєднання користувачем до чату
Основний потік подій	1. Користувач натискає на іншого користувача що онлайн 2. Система створює чат рум для обох користувачів та відкриває форму переписки
Виняткові ситуації	Відсутні

Примітки	Відсутні
----------	----------

Таблиця 1.3 – Сценарій використання «Надсилання повідомлення»

Назва	Надсилання повідомлення
Передумови	Користувач приєднався до створеного чату
Постумови	Користувачем було успішно надіслано повідомлення
Сторони, що взаємодіють	Користувач, система
Опис	Цей сценарій описує надсилання повідомлення користувачем
Основний потік подій	1.Користувач вводить повідомлення та відправляє
Виняткові ситуації	Відсутні
Примітки	Відсутні

Таблиця 1.4 – Сценарій використання «Створення групи»

Назва	Створення групи
Передумови	Користувач входить в форму зі створенням групи

Постумови	Користувач створив групу
Сторони, що взаємодіють	Користувач, система
Опис	Цей сценарій описує створення групи
Основний потік подій	1.Користувач знаходиться на формі зі створенням груп 2.Вводить назву та вибирає тих хто має входити до групи 3.Натискає зберегти
Виняткові ситуації	Відсутні
Примітки	Відсутні

Діаграма прецедентів в UML є інструментом для ілюстрації функціональності, пропонованої системою, і взаємодії між системою та її користувачами. Ця діаграма служить основою для розуміння вимог та очікувань від програмного продукту і використовується для визначення зовнішнього вигляду та поведінки системи з точки зору користувача.

У складі діаграми прецедентів UML ключовими елементами є прецеденти (use cases), які описують стандартні шляхи користувачів через систему, актори (actors), що представляють користувачів або зовнішні системи, і відносини між ними, які показують, як актори залучаються до прецедентів.

Для моделювання та візуалізації діаграм прецедентів я застосував інструмент PlantUML , який дозволяє легко перетворювати текстові описи на структуровані візуальні представлення. Це забезпечує простий, але потужний спосіб документування вимог, що є важливим аспектом процесу розробки програмного

забезпечення.

У процесі проектування системи було побудовано діаграму прецедентів представлену на рис 1.1.

Актором є користувач системи: звичайний користувач.

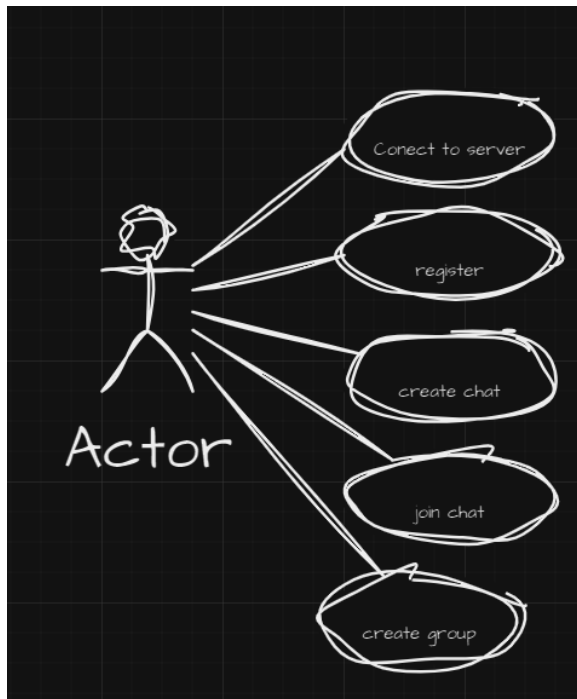


Рис 1.1 – Діаграма прецедентів

1.5 Концептуальна модель системи

У своєму застосунку я буду використовувати клієнт-сервер архітектуру. Проект можна розділити на декілька рівнів:

Клієнтська частина:

Візуальне відображення: Клієнтська частина містить графічний інтерфейс для користувача, де він може вводити повідомлення, бачити чат та інші елементи.

Логіка обробки дій користувача: Ця частина обробляє події користувача, такі як відправка повідомлень, підключення до сервера та інші дії.

Серверна частина:

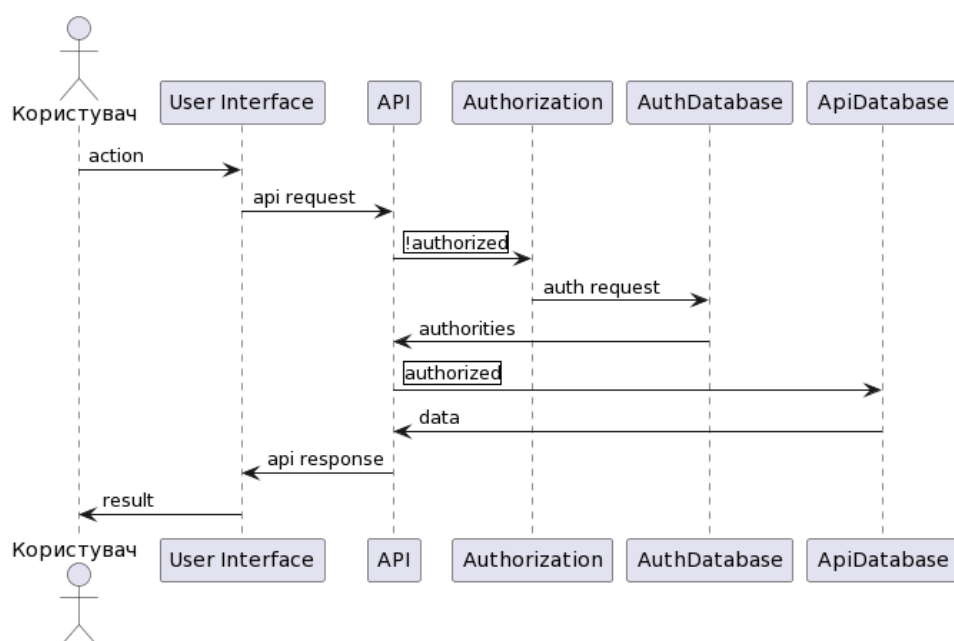
Основна логіка роботи: Серверна частина містить бізнес-логіку для обробки

повідомлень, управління користувачами та іншу логіку, яка відповідає за роботу чату.

Забезпечення зв'язку: Серверна частина відповідає за приймання і обробку повідомлень від клієнтів та надсилення їх іншим користувачам.

Взаємодія між клієнтом і сервером:

Для встановлення зв'язку між клієнтом і сервером використовується WebSocket протокол. Клієнт та сервер можуть обмінюватися повідомленнями в режимі реального часу. Сервер слухає запити від клієнтів і відповідає на них. Він також має можливість сповіщати клієнтів про нові повідомлення в чаті.



Діаграма

послідовностей зображена на рисунку 1.2

Рисунок 1.2 – Діаграма послідовностей

Структура проекту:

Ось як можна класифікувати різні компоненти нашого office communicator client-server застосунку:

Серверна частина (Server-side):

`com.communicator.websocket.config` – містить класи конфігурації WebSocket, які визначають налаштування для веб-сокета сервера.

`com.communicator.websocket..controller` – включає контролери, які

обробляють вхідні повідомлення від клієнтів (наприклад, ChatController для обробки повідомлень чату) і події пов'язані з веб-сокетами (WebSocketEventListener).

com.communicator.websocket.mapper – містить інтерфейси які перетворюють одні класи на інші

com.communicator.websocket.repository – містить репозиторії для зв'язку з базою даних

com.communicator.websocket.service – містить сервіси з бізне-логікою

com.example.websocketdemo.model – визначає структуру даних, з якими працює сервер (наприклад, ChatMessage для зберігання інформації про повідомлення).

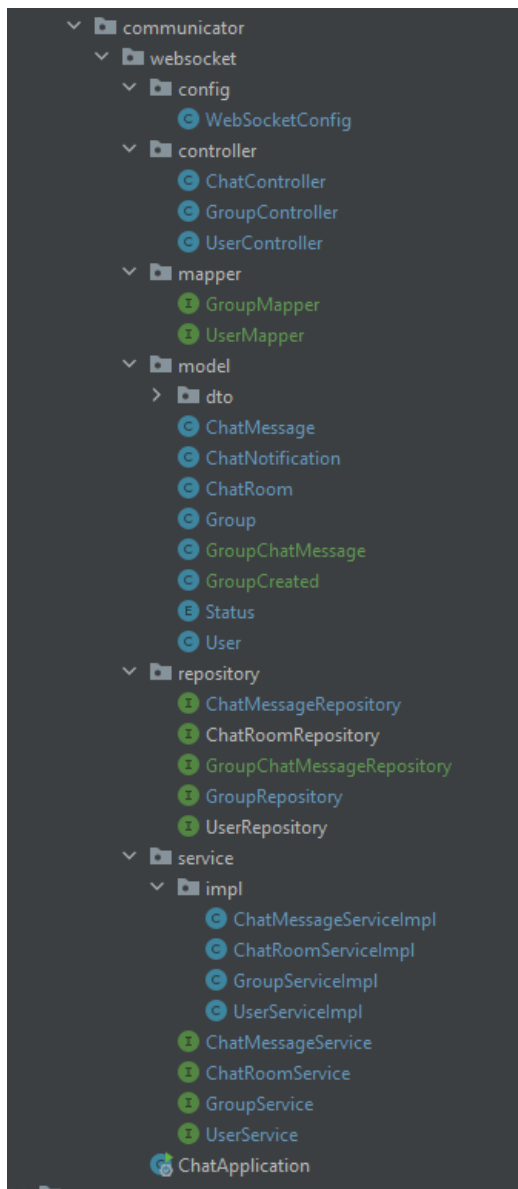


Рисунок 1.3 – Структура серверної частини

Клієнтська частина (Client-side):

resources/static – ця папка зазвичай містить статичні ресурси для клієнтської частини застосунку, такі як HTML (index.html), CSS (main.css), та JavaScript (main.js) файли, які виконуються в браузері користувача.

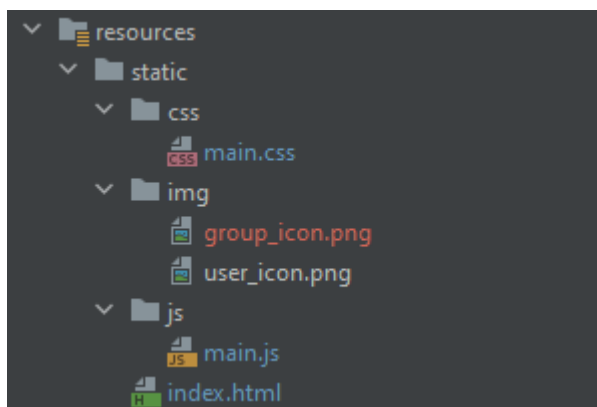


Рисунок 1.4 – Структура клієнтської частини

Бізнес логіка (Business logic):

Частково реалізована в `com.communicator.websocket.controller` та `com.communicator.websocket.service`. Контролери обробляють запити і відповіді, тоді як `sthdsdb` можуть займатися валідацією чи модифікацією повідомлень.

`resources/application.properties` – файл конфігурації, який може містити налаштування для застосунку, такі як порт сервера, параметри бази даних та інше.

Основні етапи клієнт серверної архітектури:
З'єднання: Клієнти (наприклад, веб-браузери або мобільні додатки) з'єднуються з сервером через мережу, часто використовуючи протоколи, такі як HTTP.

Відправлення запиту: Коли користувач взаємодіє з клієнтською програмою (наприклад, натискає кнопку), програма формує та відправляє запит на сервер.

Обробка сервером: Сервер приймає запит, обробляє його (може включати доступ до бази даних, виконання обчислень тощо) та створює відповідь.

Відправлення відповіді: Сервер відправляє відповідь назад до клієнта.

Обробка відповіді клієнтом: Клієнтське програмне забезпечення отримує відповідь, обробляє її та відображає користувачеві результати.

Загальна діаграма класів показана в додатку А.

1.6 Вибір Архітектури Системи

В процесі розробки системи communicator було прийняте важливе архітектурне рішення - не використовувати PostgreSQL базу даних для зберігання або управління даними. Це рішення було обґрунтоване наступними ключовими факторами:

Реалізація Чату в Реальному Часі зі збереженням повідомлень: Основна функція системи - забезпечення миттєвого обміну повідомленнями між користувачами. Це вимагає мінімальних затримок та високої швидкості передачі даних, яку може гарантувати використання веб-сокетів, також ми маємо зберегти повідомлення для подальшого збереження історії чатів.

Безпека та Приватність: інформація знаходиться в базі даних яка приєднана до серверів тобто ніхто інші користувачі не зможуть зробити запити та дістатись цих повідомлень без дозволу власника.

Масштабованість та Гнучкість: Використання веб-сокетів дозволяє легко масштабувати систему, оскільки вона може обробляти велику кількість одночасних з'єднань без значного навантаження на інфраструктуру.

Простота Розробки та Підтримки: Рішення використовувати postgres також полегшує процес розробки та підтримки системи, зменшуючи кількість компонентів, які потребують оновлення та налагодження.

Таким чином, вибір веб-сокетів та postgres бази даних для communicator client-server системи є свідомим та виправданим рішенням, яке враховує специфіку застосунку та вимоги до його функціональності.

1.7 Вибір мови програмування та середовища розробки

Для реалізації communicator client-server системи було обрано Java як основну мову програмування. Java, відома своєю універсальністю та переносимістю, є ідеальним вибором для створення багатоплатформених серверних додатків. Основні переваги Java, що вплинули на цей вибір, включають її об'єктно-орієнтований підхід, що підтримує такі фундаментальні концепції, як

наслідування, абстракція та поліморфізм. Це сприяє створенню модульного та легко розширюваного коду.

Java відрізняється своєю високою надійністю та простим синтаксисом, що робить її зручною для розробників на різних рівнях досвіду. Код, написаний на Java, компілюється в байт-код, який може виконуватися на будь-якій Java віртуальній машині (JVM), надаючи високу ступінь переносимості на різні операційні системи.

Для середовища розробки було обрано IntelliJ IDEA, яке вважається одним із найпотужніших та найзручніших інструментів для розробки на Java. IntelliJ IDEA вирізняється своїм розумним код-редактором, глибокою інтеграцією з різними фреймворками та підтримкою новітніх технологій. Його інтуїтивно зрозумілий інтерфейс, ефективні інструменти рефакторингу та аналізу коду роблять процес розробки швидким і приємним.

IntelliJ IDEA також надає великий спектр можливостей для оптимізації продуктивності розробника: від швидкого доступу до різних інструментів до розширених можливостей для управління проектом, включаючи підтримку баз даних та інструменти для візуального проектування інтерфейсу.

Обрані мова програмування та середовище розробки ідеально підходять для створення надійної, масштабованої та високопродуктивної системи client-server, яка може ефективно функціонувати в різноманітних середовищах і підтримувати сучасні стандарти розробки.

1.8 Проектування розгортання системи

Діаграма розгортання допомагає моделювати фізичний аспект об'єктно орієнтованої програмної системи. Це структурна схема, яка показує архітектуру системи як розгортання (розповсюдження) програмних артефактів для цілей

розгортання. Артефакти являють собою конкретні елементи у фізичному світі, які є результатом процесу розвитку. Вона моделює конфігурацію часу виконання у статичному вигляді та візуалізує розподіл артефактів у програмі. У більшості випадків це включає моделювання апаратних конфігурацій разом із компонентами програмного забезпечення, що працювали на них.

На діаграмі розгортання ми можемо побачити:

- Вузли: Це можуть бути сервери або комп'ютери, на яких працює програма.
- Компоненти: Це частини програми, як-от база даних, веб-сервер або додатки, які розміщені на цих вузлах.
- Зв'язки: Це показує, як вузли зв'язані між собою, наприклад через інтернет або локальну мережу.

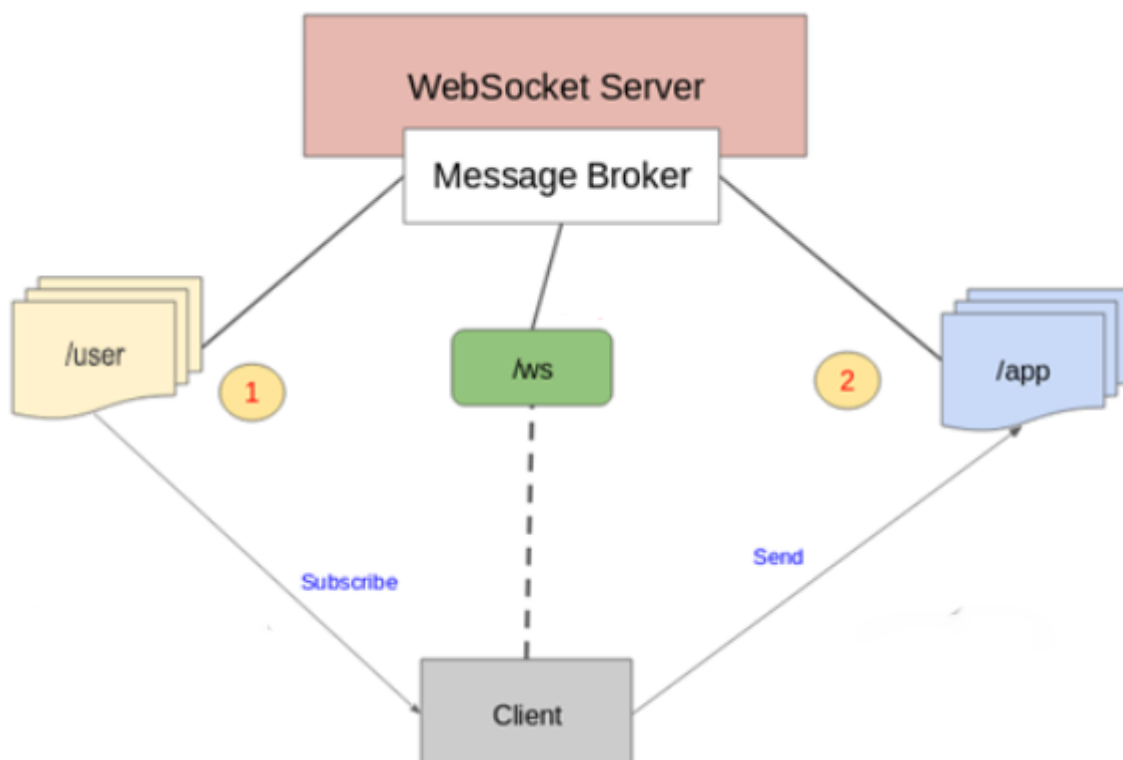


Рис 1.7 – Діаграма розгортання системи

2 РЕАЛІЗАЦІЯ КОМПОНЕНТІВ СИСТЕМИ

2.1. Структура веб-сокетів

У нашому communicator client-server проєкті, використання веб-сокетів грає ключову роль у забезпеченні надійної та ефективної двосторонньої комунікації між клієнтом та сервером. Веб-сокети дозволяють реалізувати безперервну взаємодію в реальному часі, що є критично важливим для чат-додатків.



Рис 2.1 – Структура web socket

Клієнтська частина (веб-браузер): Користувач вводить команди через графічний інтерфейс користувача. Ці команди відправляються на сервер через відкрите веб-сокет з'єднання.

Сервер: Після отримання команд від клієнта, сервер обробляє їх та відправляє відповідні повідомлення про стан назад до клієнта через те саме веб-сокет з'єднання.

Взаємодія в реальному часі: Веб-сокети дозволяють обмін повідомленнями між клієнтом та сервером в реальному часі, що забезпечує безперервну інтерактивність та актуальність даних.

Відсутність традиційної бази даних: Замість зберігання інформації про команди та їх стан у базі даних, дані передаються безпосередньо через веб-сокети. Це означає, що історія команд та їх результати обробляються і зберігаються в оперативному стані, що дозволяє швидко реагувати на запити користувачів.

2.2. Архітектура системи

2.2.1. Специфікація системи

У центрі цієї курсової роботи знаходиться communicator client система, розроблена як веб-застосунок, що використовує сучасні технології веб-сокетів для надання користувачам можливості спілкування в реальному часі. На відміну від десктоп-додатків, які вимагають інсталяції та працюють із ресурсами локального комп'ютера, наша система функціонує безпосередньо через веб-браузер, забезпечуючи миттєвий обмін повідомленнями між сервером та клієнтом.

Важливою особливістю communicator client системи є її здатність працювати в режимі онлайн, що дозволяє користувачам підключатися до чатів з будь-якої точки світу, маючи лише Інтернет-з'єднання. Система не зберігає історію повідомлень на стороні клієнта або сервера, що підкреслює ефемерний характер спілкування та спрощує вимоги до зберігання даних.

Розробка системи була здійснена за допомогою Java і Spring Framework, які є широко визнаними та випробуваними засобами для створення надійних веб-серверів та додатків. Для проектування користувацького інтерфейсу було використано стандартизовані технології HTML, CSS і JavaScript, що забезпечують кросплатформену сумісність та широкий спектр можливостей для розробки інтерфейсів користувача.

Веб-сокети були обрані як основний канал комунікації, оскільки вони дозволяють встановлювати двосторонній зв'язок між клієнтом та сервером, що є ключовою вимогою для систем миттєвого месенджеру. Це дозволяє користувачам отримувати та надсилати повідомлення без затримок, що є суттєвим для забезпечення відчуття безперервної розмови.

Завершена client система відображає високу ступінь інноваційності та технічної компетентності, демонструючи можливості сучасного веб-програмування та відповідаючи потребам користувачів у швидкому та надійному засобі для онлайн-спілкування.

2.2.2. Вибір та обґрунтування патернів реалізації

Шаблони проектування представляють собою перевірені рішення для розповсюджених викликів, які виникають під час створення програмної архітектури. Вони не є готовими до використання компонентами чи бібліотеками, які можна безпосередньо імплементувати в код; натомість, шаблони проектування надають загальні рекомендації для розв'язання програмних завдань, які вимагають індивідуального підходу в кожному окремому випадку.

Шаблони часто сприймаються аналогічно до алгоритмів, оскільки обидва концепти пропонують рішення стандартних проблем. Однак, ключова відмінність полягає у тому, що алгоритм є докладним планом дій, тоді як шаблон проектування пропонує гнучку структуру, яка адаптується до різних ситуацій і може бути реалізована по-різному в залежності від конкретних вимог проекту.

1. Паттерн "Strategy" (Стратегія) відноситься до паттернів проектування і визначає сімейство алгоритмів, розміщаючи кожен з них у відокремленому класі і зроблюючи їх взаємозамінними. Цей паттерн дозволяє об'єкту змінювати свій алгоритм виконання незалежно від клієнтів, які його використовують.

Реалізація в коді зображена на рисунку 2.2:

```
3 usages
public class OfficeCommunicator {

    3 usages
    private CommunicationStrategy communicationStrategy;

    3 usages
    public void setCommunicationStrategy(CommunicationStrategy communicationStrategy) {
        this.communicationStrategy = communicationStrategy;
    }

    3 usages
    public String communicate() {
        if (communicationStrategy != null) {
            return communicationStrategy.communicate();
        } else {
            return "Не вибрана стратегія комунікації";
        }
    }
}
```

Рисунок 2.2 – Реалізація патерну Strategy фрагмент коду

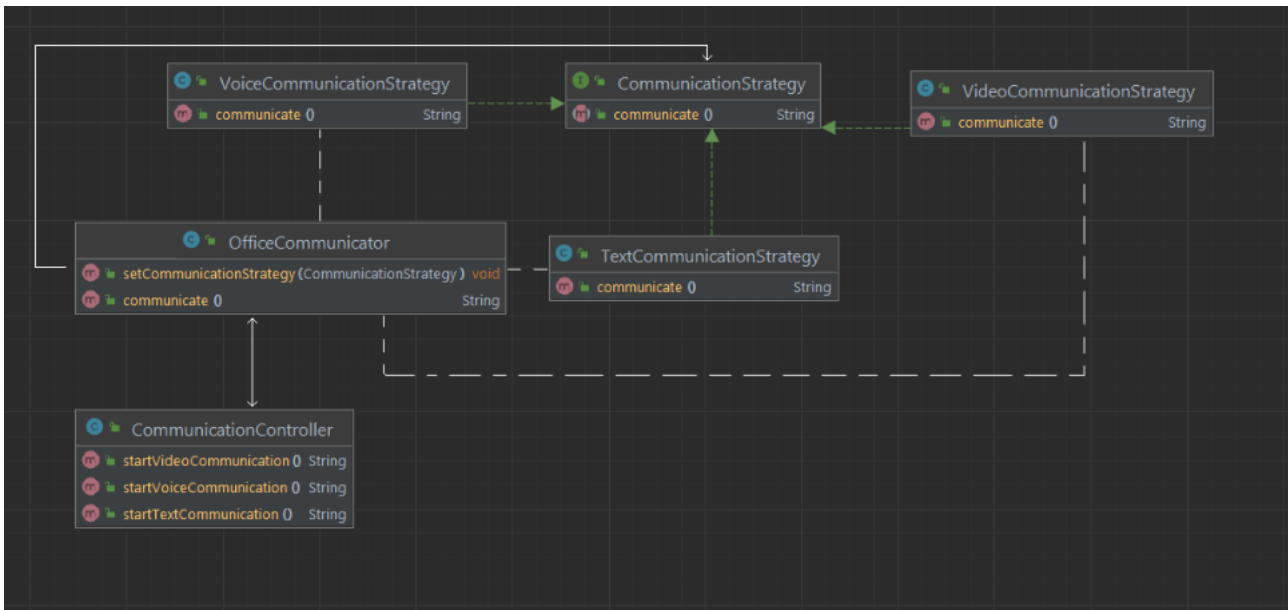


Рисунок 2.3 – Діаграма класу, в якому реалізовано Strategy

```

public interface CommunicationStrategy {
    1 usage 3 implementations
    String communicate();
}
  
```

Рисунок 2.4 - фрагмент коду для застосування реалізації патерну

2. Паттерн "Adapter" (Адаптер) — це структурний паттерн проектування, який дозволяє інтерфейс одного класу виглядати так, ніби це інший інтерфейс. В інших словах, він дозволяє об'єктам зі несумісними інтерфейсами працювати разом.

```

package com.example.communicator.web.strategy;

public interface VoiceCommunication {
    1 usage 1 implementation
    String startVoiceCall();
    1 usage 1 implementation
    String endVoiceCall();
}
  
```



```

public class VoiceCommunicationAdapter implements CommunicationStrategy {
    3 usages
    private final VoiceCommunication voiceCommunication;

    public VoiceCommunicationAdapter(VoiceCommunication voiceCommunication) {
        this.voiceCommunication = voiceCommunication;
    }

    1 usage
    @Override
    public String communicate() {
        String startResult = voiceCommunication.startVoiceCall();
        String endResult = voiceCommunication.endVoiceCall();

        return startResult + "\n" + endResult;
    }
}

```

```

public class VoiceCommunicationAdapter implements CommunicationStrategy {
    3 usages
    private final VoiceCommunication voiceCommunication;

    public VoiceCommunicationAdapter(VoiceCommunication voiceCommunication) {
        this.voiceCommunication = voiceCommunication;
    }

    1 usage
    @Override
    public String communicate() {
        String startResult = voiceCommunication.startVoiceCall();
        String endResult = voiceCommunication.endVoiceCall();

        return startResult + "\n" + endResult;
    }
}

```

Реалізація в коді зображена на рисунку 2.5:

3. Паттерн "Абстрактна фабрика" — це породжуючий паттерн проектування, який надає інтерфейс для створення сімейств взаємодіючих або взаємозалежних об'єктів без зазначення їхніх конкретних класів. Він вводить абстракцію для

створення об'єктів певної тематичної групи, таким чином роблячи систему незалежною від того, які саме класи створюються.

Реалізація в коді зображена на рисунку 2.6:



```

2 usages 2 inheritors
public abstract class Department {
    2 usages 2 implementations
    public abstract String getDepartmentName();
    2 usages 2 implementations
    public abstract List<UserDTO> getDepartmentEmployees( List<UserDTO> allUsers);
}

3 usages
public class FinanceReportDepartment extends Department {
    2 usages
    public String getDepartmentName() { return "Finance Department"; }
    2 usages
    @Override
    public List<UserDTO> getDepartmentEmployees( List<UserDTO> allUsers) {
        return allUsers.stream()
            .filter(user -> "FinanceReport".equalsIgnoreCase(user.getDepartment()))
            .collect(Collectors.toList());
    }
}

@RequiredArgsConstructor
public class HRReportDepartment extends Department {
    2 usages
    @Override
    public String getDepartmentName() { return "HR Department"; }
    2 usages
    @Override
    public List<UserDTO> getDepartmentEmployees( List<UserDTO> allUsers) {
        return allUsers.stream()
            .filter(user -> "HRReport".equalsIgnoreCase( user.getDepartment()))
            .collect(Collectors.toList());
    }
}

```

Переваги паттерну "Абстрактна фабрика":

1. Гнучкість системи: Дозволяє замінювати сімейства пов'язаних об'єктів, не змінюючи код клієнта.
2. Легкість впровадження: Визначає чіткі межі для створення об'єктів, що полегшує їх реалізацію.
3. Підтримка принципу відкритості/закритості: Легко можна додавати нові типи об'єктів, розширюючи абстрактні фабрики та додаючи нові конкретні фабрики та продукти.

Недоліки паттерну "Абстрактна фабрика":

1. Складність додавання нових продуктів: Додавання нових продуктів вимагає модифікації абстрактної фабрики та всіх її конкретних реалізацій.
2. Можливий зріст числа класів: Зі зростанням числа продуктів та їхніх варіацій збільшується кількість класів, що може призвести до перенасиченості системи.
3. Складність заміни продуктів: Заміна конкретних продуктів може бути складною, особливо якщо вони вже використовуються у багатьох місцях системи.

```

@RestController
@RequestMapping("@*/department")
@RequiredArgsConstructor
public class DepartmentController {

    @Inject
    private final UserMapper userMapper;

    @Inject
    private final UserService userService;

    @GetMapping("@*/finance/employees")
    public ResponseEntity<List<UserDTO>> getFinanceDepartmentEmployees() {
        Department financeDepartment = new FinanceReportDepartment();
        return ResponseEntity.ok(financeDepartment.getDepartmentEmployees(userMapper.toUserDTOList(userService.getAllUsers())));
    }

    @GetMapping("@*/hr/employees")
    public ResponseEntity<List<UserDTO>> getHRDepartmentEmployees() {
        Department hrDepartment = new HRReportDepartment();
        return ResponseEntity.ok(hrDepartment.getDepartmentEmployees(userMapper.toUserDTOList(userService.getAllUsers())));
    }
}

@GetMapping("@*/finance/name")
public ResponseEntity<String> getFinanceDepartment() {
    Department financeDepartment = new FinanceReportDepartment();
    return ResponseEntity.ok(financeDepartment.getDepartmentName());
}

@GetMapping("@*/hr/name")
public ResponseEntity<String> getHRDepartment() {
    Department hrDepartment = new HRReportDepartment();
    return ResponseEntity.ok(hrDepartment.getDepartmentName());
}

```

Рисунок 2.7 - фрагмент коду реалізації патерну Абстрактна фабрика

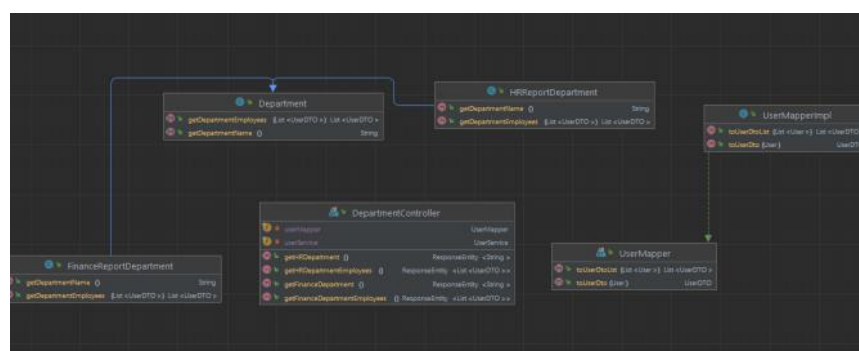


Рисунок 2.8 – взаємодія класів для патерну Абстрактна фабрика

4. Паттерн "Міст" відноситься до структурних паттернів проектування та використовує принцип композиції замість успадкування для розділення абстракції від її реалізації так, щоб обидві частини можна було змінювати незалежно одна від одної.

```

7 usages 2 inheritors
@Data
public abstract class Pizza {
    2 usages
    protected String sauce;
    2 usages
    protected String topping;
    2 usages
    protected String crust;
    1 usage 2 implementations
    public abstract void assemble();
    1 usage 2 implementations
    public abstract void qualityCheck();
}

```

```

public class PapperoniPizza extends Pizza{
    1 usage
    @Override
    public void assemble() {
        System.out.println("Adding Sauce: " + sauce);
        System.out.println("Adding Toppings: " + topping);
        System.out.println("adding peperoni");
    }

    1 usage
    @Override
    public void qualityCheck() {
        System.out.println("Crust is: " + crust);
    }
}

```

```

public class VeggiePizza extends Pizza{
    1 usage
    @Override
    public void assemble() {
        System.out.println("Adding Sauce: " +sauce);
        System.out.println("Adding Toppings: " + topping);
        System.out.println("adding cheese");
    }

    1 usage
    @Override
    public void qualityCheck() { System.out.println("Crust is: " + crust); }
}

```

```

public abstract class Restaurant {
    9 usages
    protected Pizza pizza;

    2 usages
    public Restaurant(Pizza pizza) {
        this.pizza = pizza;
    }

    1 usage 2 implementations
    abstract void addSauce();
    1 usage 2 implementations
    abstract void addTopping();
    1 usage 2 implementations
    abstract void makeCrust();
    1 usage
    public void deliver(){
        makeCrust();
        addSauce();
        addTopping();
        pizza.assemble();
        pizza.qualityCheck();
        System.out.println("Order in Progress!");
    }
}

```

```

public class ItalianRestaurant extends Restaurant{
    1 usage
    public ItalianRestaurant(Pizza pizza) {
        super(pizza);
    }

    1 usage
    @Override
    void addSauce() {
        pizza.setTopping(null);
    }

    1 usage
    @Override
    void addTopping() {
        pizza.setSauce("Oil");
    }

    1 usage
    @Override
    void makeCrust() {
        pizza.setCrust("Thin");
    }
}

```

Рисунок 2.9 - фрагмент коду

Переваги:

1. Розділення абстракції від реалізації: Дозволяє змінювати обидві частини незалежно, що полегшує розширення та модифікацію системи.
2. Гнучкість та розширюваність: Дозволяє додавати нові абстракції та реалізації без потреби модифікації іншої частини системи.

Недоліки:

1. Збільшення кількості класів: З використанням паттерну може збільшитися кількість класів, що може зробити систему складнішою для управління.
2. Додаткова складність: У випадку простих систем використання паттерну "Міст" може бути зайвим та вводити додаткову складність.

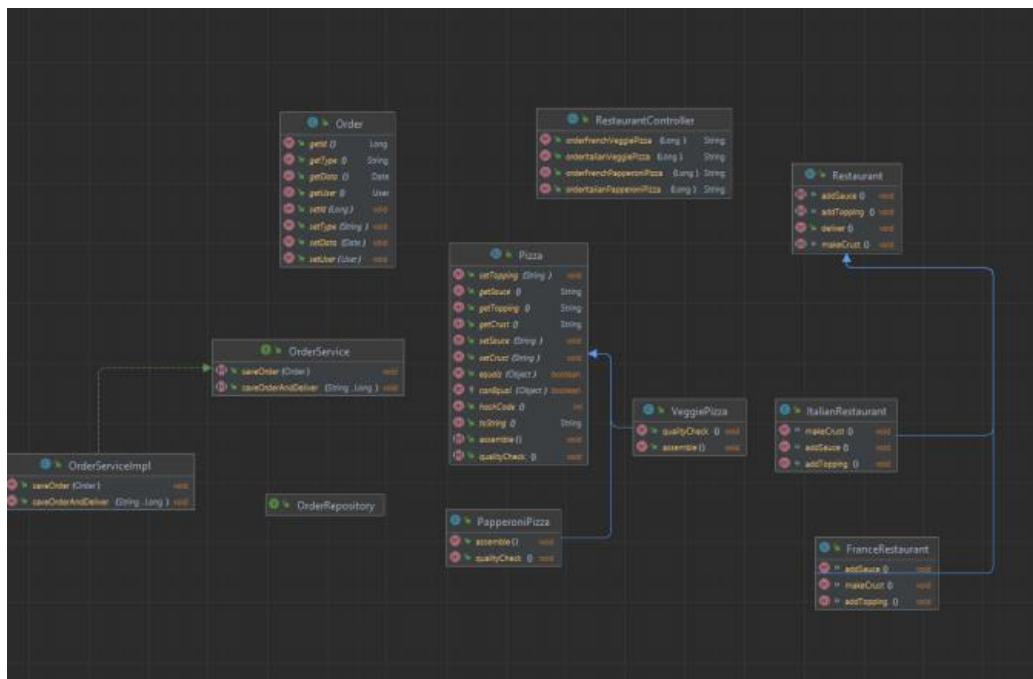


Рисунок 2.10 - діаграма класів

Паттерн "Компонувальник" відноситься до структурних паттернів проектування та дозволяє клієнтам обробляти окремі об'єкти та їхні складові однаковим чином. Цей паттерн створює ієрархію класів, де індивідуальні об'єкти та їхні комбінації (складові) обробляються єдиним інтерфейсом

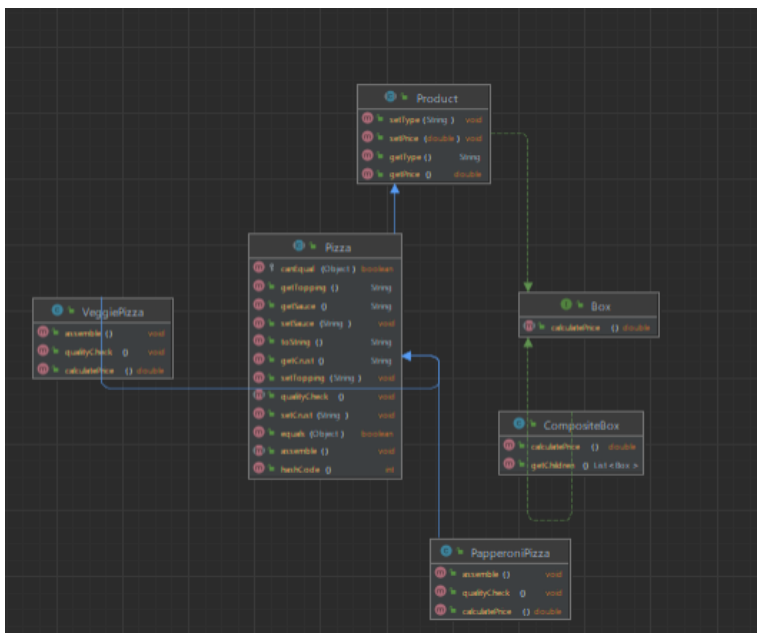


Рисунок 2.11 - діаграма класів



```

Pavlo Ilnitskyi
public interface Box {
    2 usages  3 implementations  Pavlo Ilnitskyi
    double calculatePrice();
}

public class CompositeBox implements Box {
    2 usages
    private final List<Box> children = new ArrayList<>();

    1 usage  Pavlo Ilnitskyi
    public CompositeBox(Box... boxes) { children.addAll(Arrays.asList(boxes)); }

    2 usages  Pavlo Ilnitskyi
    @Override
    public double calculatePrice() {
        return children.stream()
            .mapToDouble(Box::calculatePrice)
            .sum();
    }
}

@RequiredArgsConstructor
@Getter
@Setter
public abstract class Product implements Box {
    private String type;
    private double price;
}

```

Рисунок 2.12 – Приклад коду

Переваги паттерну "Компонувальник" (Composite):

1. Загальний інтерфейс: Всі об'єкти використовують однаковий інтерфейс, що полегшує їхнє використання клієнтом.
2. Гнучкість та розширюваність: Можливість динамічно створювати складні структури та додавати нові об'єкти без зміни коду клієнта.
3. Спрощення коду клієнта: Клієнт може взаємодіяти зі складними структурами так само, як і з індивідуальними об'єктами.
4. Зручність при роботі з деревом структур: Паттерн ефективний при представленні ієрархічних структур, таких як дерева.

Недоліки паттерну "Компонувальник" (Composite):

1. Обмеження уніфікації інтерфейсу: Спільний інтерфейс може обмежити те, що може бути реалізовано для різних типів компонентів.

2. Надмірне використання пам'яті: З великою кількістю листів та контейнерів може виникнути надмірне використання пам'яті.

3. Складність управління об'єктами: Деякі операції, які пов'язані із складними структурами, можуть бути складними для управління та обслуговування.

4. Неефективність при ітерації: Ітерація через всі елементи може бути неефективною в деяких випадках, особливо при великих деревах.

"Клієнт-сервер" є фундаментальним архітектурним патерном, в якому сервер надає послуги, а клієнти ці послуги використовують. В контексті веб-додатків та мережевого програмування, цей патерн описує взаємодію між двома частинами програми: клієнтом, який ініціює запит, та сервером, який обробляє запит і надає відповідь.

Переваги:

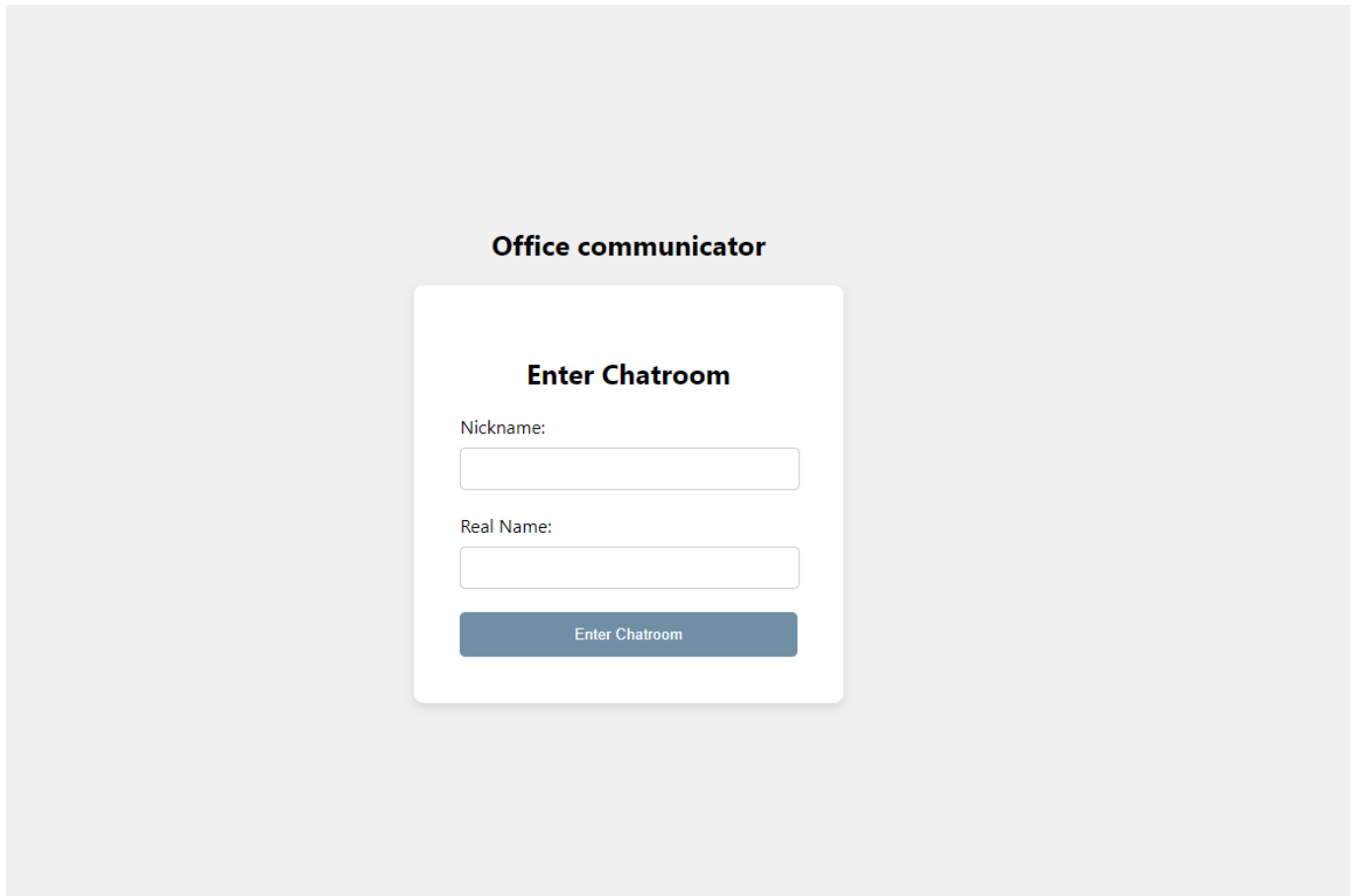
Модульність та розподіл відповідальності: Клієнтська та серверна частини розробляються та підтримуються незалежно, що сприяє кращій організації коду та легшому управлінню.

Масштабованість: Сервер може обслуговувати велику кількість клієнтів, що робить архітектуру ефективною для великих застосунків.

Гнучкість в розгортанні: Клієнтська частина може бути легко доступна через веб-браузери без необхідності інсталяції додаткового програмного забезпечення.

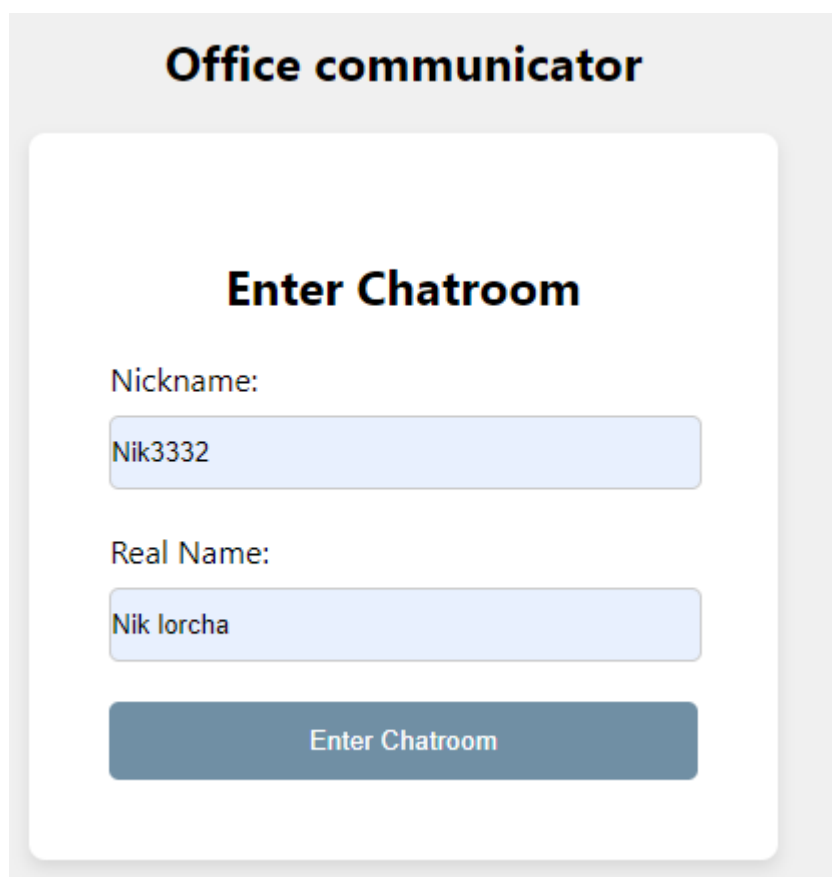
2.3. Інструкція користувача

На рисунку зображено графічний інтерфейс системи. На початку нас зустрічає сторінка із входом користувача.



The image shows a login form for an 'Office communicator'. The form is centered on a light gray background. It has a title 'Office communicator' in bold black text. Below the title is a white box with rounded corners containing the text 'Enter Chatroom' in bold black text. Underneath, there are two input fields: 'Nickname:' followed by a text box, and 'Real Name:' followed by a text box. At the bottom of the white box is a blue button with the text 'Enter Chatroom' in white.

Рисунок 2.31 – графічний інтерфейс системи



The image shows a login form for an 'Office communicator'. The form is titled 'Enter Chatroom' and is set against a light gray background. It contains two text input fields: one for 'Nickname' with the value 'Nik3332' and one for 'Real Name' with the value 'Nik Iorcha'. Below these fields is a blue button labeled 'Enter Chatroom'.

Office communicator

Enter Chatroom

Nickname:

Nik3332

Real Name:

Nik Iorcha

Enter Chatroom

Рисунок 2.32 – графічне представлення початкової сторінки



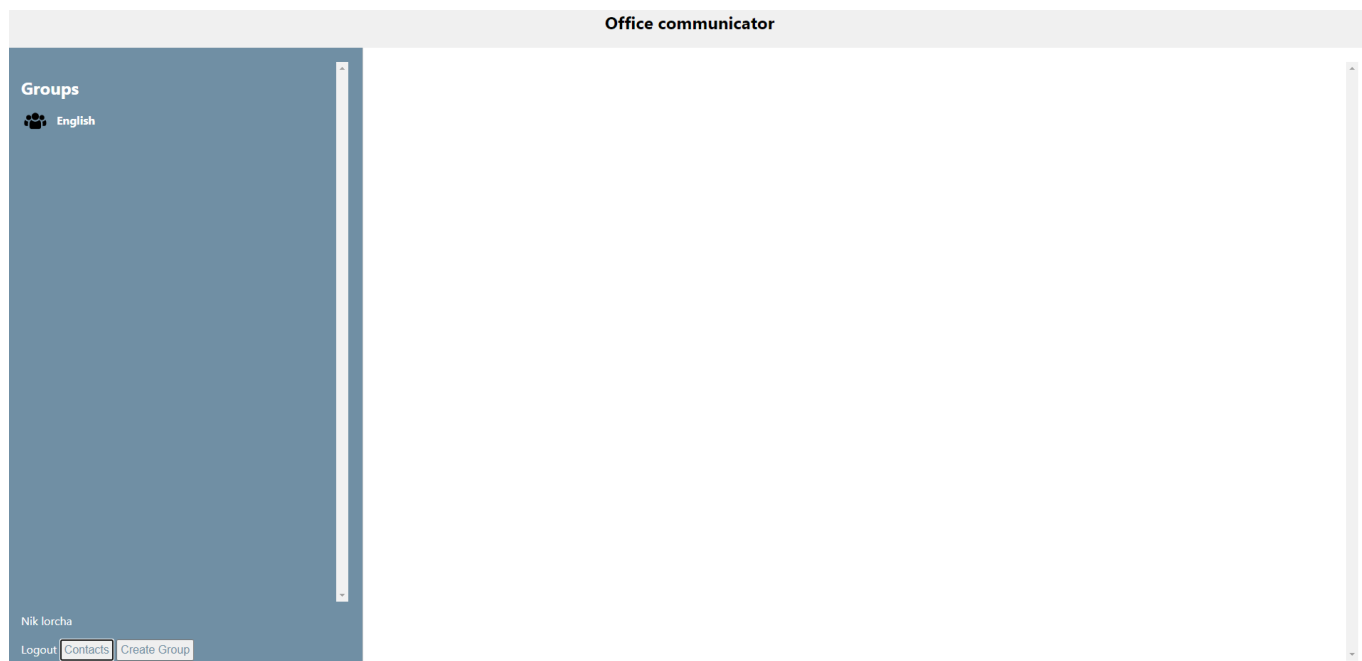


Рисунок 2.32 – графічне представлення головної сторінки з користувачами та групами

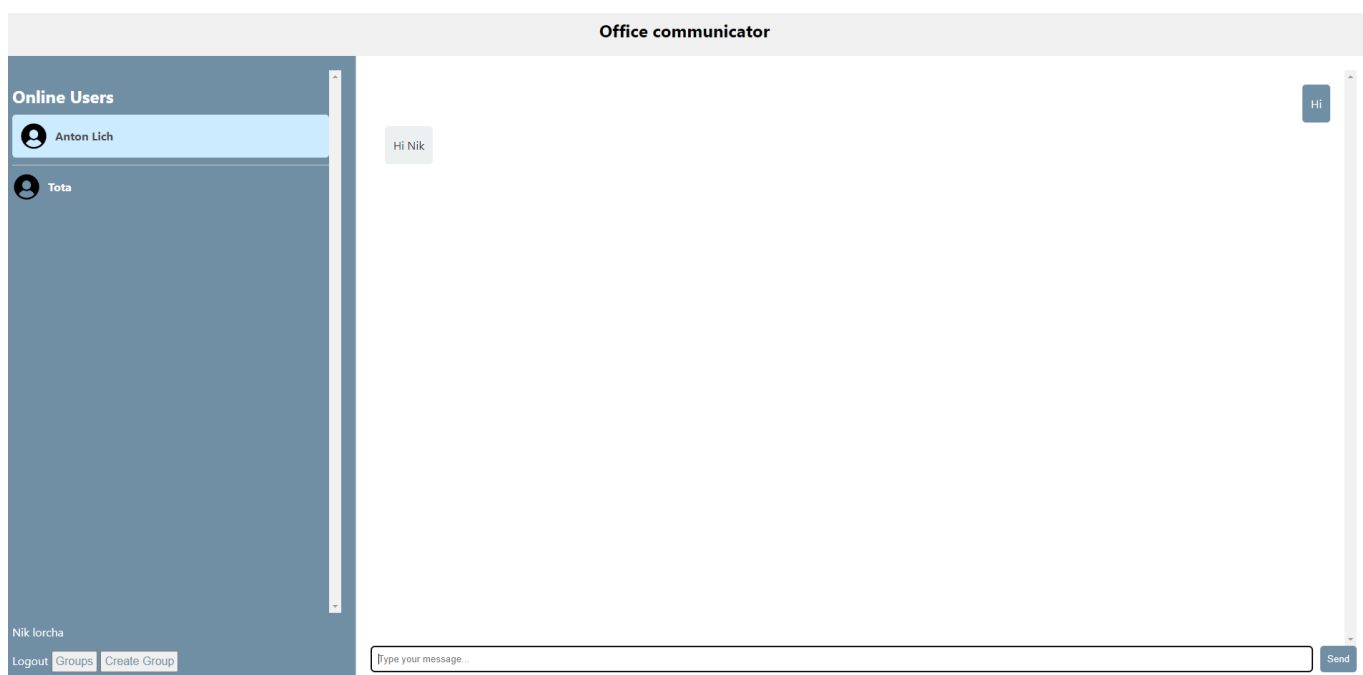


Рисунок 2.33 – відображення чату з іншим користувачем

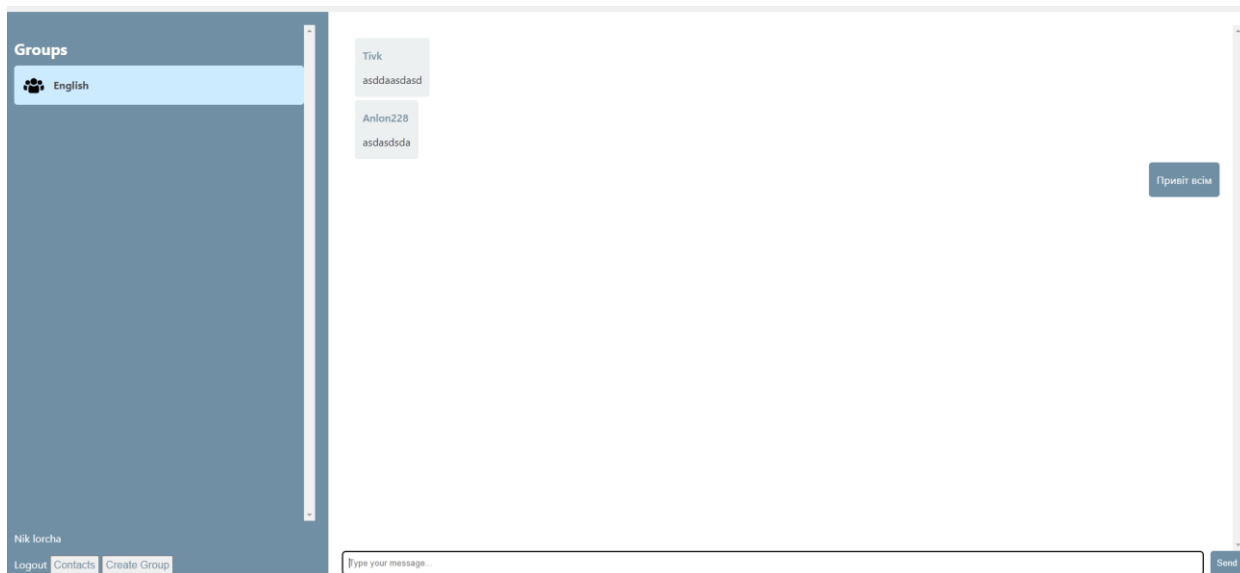


Рисунок 2.33 – відображення чату з групою

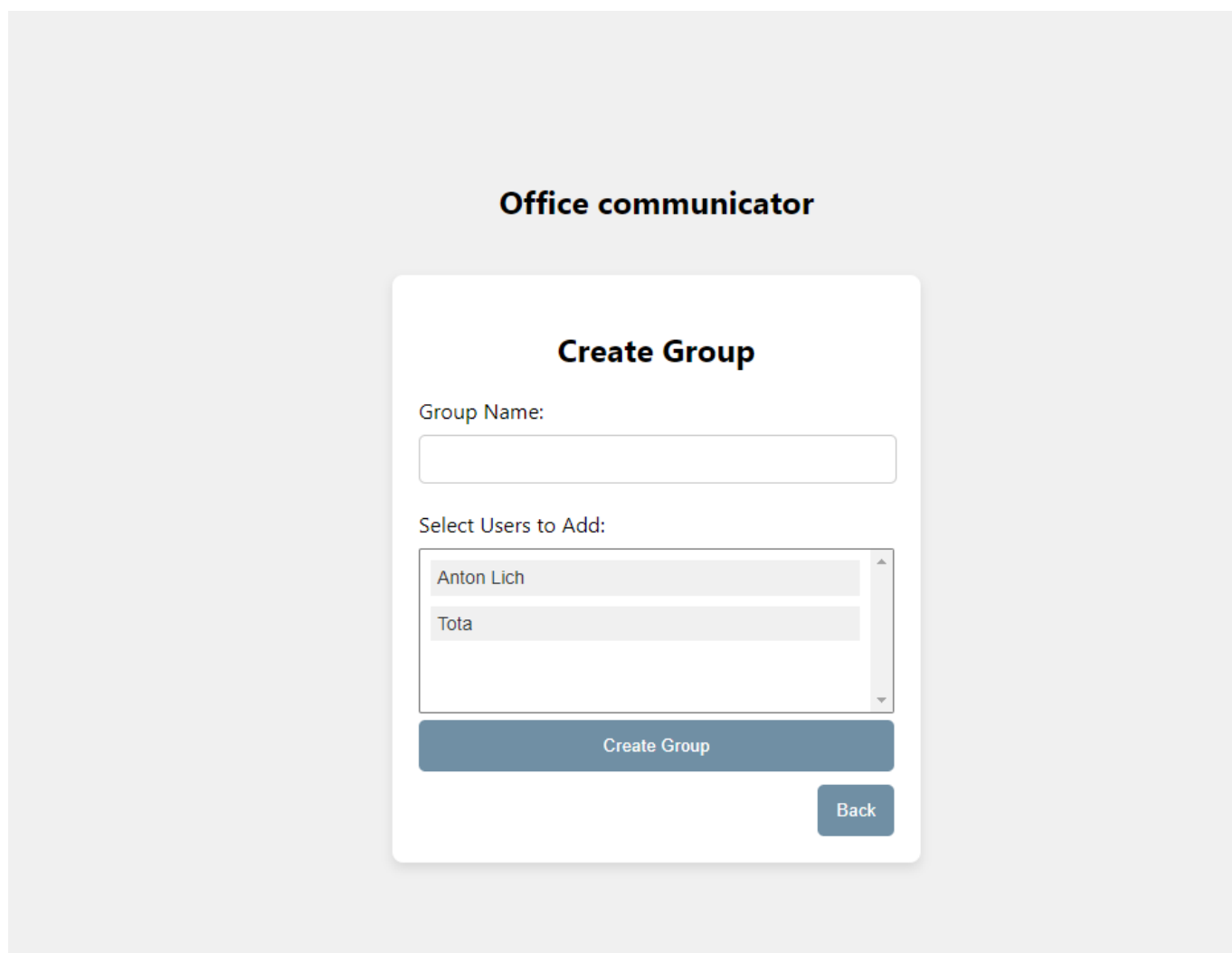


Рисунок 2.33 – відображення форми зі створенням групи та можливими учасниками

ВИСНОВКИ

Протягом виконання даної курсової роботи вдалося досягти значних результатів у розробці системи Office Communicator, яка відповідає поточним стандартам для миттєвого обміну повідомленнями в офісному оточенні. Робота охопила широкий спектр завдань, включаючи аналіз, проектування та реалізацію програмного забезпечення, з використанням голосового та відео-зв'язку, текстових повідомлень та організованого списку контактів.

Були визначені та ретельно описані функціональні та нефункціональні вимоги, що послужило основою для подальшого проектування системи. Особлива увага була приділена вибору мови програмування та середовища розробки, забезпечуючи надійність, масштабованість та переносимість проекту в офісному середовищі.

Під час реалізації проекту використовувалися різноманітні шаблони проектування, що сприяло організації коду згідно з принципами ООП та забезпечило його гнучкість та модифікованість. Розробка діаграм UML сприяла зрозумінню та чіткості структури системи, а також визначенню взаємозв'язків між її компонентами.

З використанням передових технологій та методологій система не лише відповідає всім поставленим вимогам, але й має широкі перспективи для подальшого розвитку та вдосконалення. Робота над проектом дозволила отримати цінний досвід у розробці мережевих додатків та в галузі офісного програмування.

Загалом, створена система Office Communicator представляє собою повнофункціональну та ефективну платформу для миттєвого обміну повідомленнями в офісному оточенні, що відповідає найвищим стандартам сучасності та забезпечує зручне та результативне спілкування між співробітниками..

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

Курс по Spring Framework [Електронний ресурс].

https://www.udemy.com/share/1070y03@vG1vJZS379gYiqYcJ6nak3qsG-nZ196F0tfGu9tcy89a_2wy87T3HmIliuMNA==/

Internet Relay Chat: Architecture. [Електронний ресурс]. Доступ:

<https://tools.ietf.org/html/rfc1459>.

WebSockets - A Conceptual Deep-Dive. [Електронний ресурс]. Доступ:

https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API.

UML 2.0 in a Nutshell: A Desktop Quick Reference. [Електронний ресурс].

Доступ: <https://www.oreilly.com/library/view/uml-20-in/0596007957/>.

Spring Framework Documentation. [Електронний ресурс]. Доступ:

<https://spring.io/docs>.

IntelliJ IDEA Documentation. [Електронний ресурс]. Доступ:

<https://www.jetbrains.com/idea/documentation/>.

WebSocket Protocol in Java. [Електронний ресурс]. Доступ:

<https://javaee.github.io/tutorial/websocket.html>.

Design Patterns: Elements of Reusable Object-Oriented Software. [Електронний

ресурс]. Доступ: [https://www.pearson.com/us/higher-education/program/Gamma-](https://www.pearson.com/us/higher-education/program/Gamma-Design-Patterns-Elements-of-Reusable-Object-OrientedSoftware/PGM310633.html)

[Design-Patterns-Elements-of-Reusable-Object-OrientedSoftware/PGM310633.html](https://www.pearson.com/us/higher-education/program/Gamma-Design-Patterns-Elements-of-Reusable-Object-OrientedSoftware/PGM310633.html).

WebSocket Testing with Postman. [Електронний ресурс]. Доступ:

<https://blog.postman.com/websocket-testing/>.

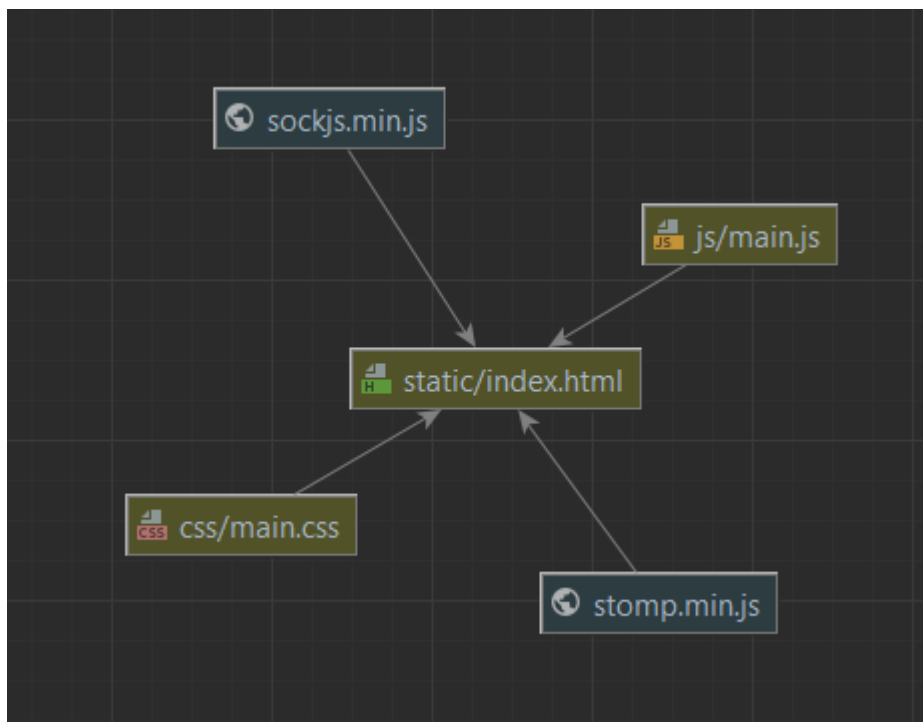
Java Programming Language. [Електронний ресурс]. Доступ:

<https://docs.oracle.com/javase/tutorial/>.

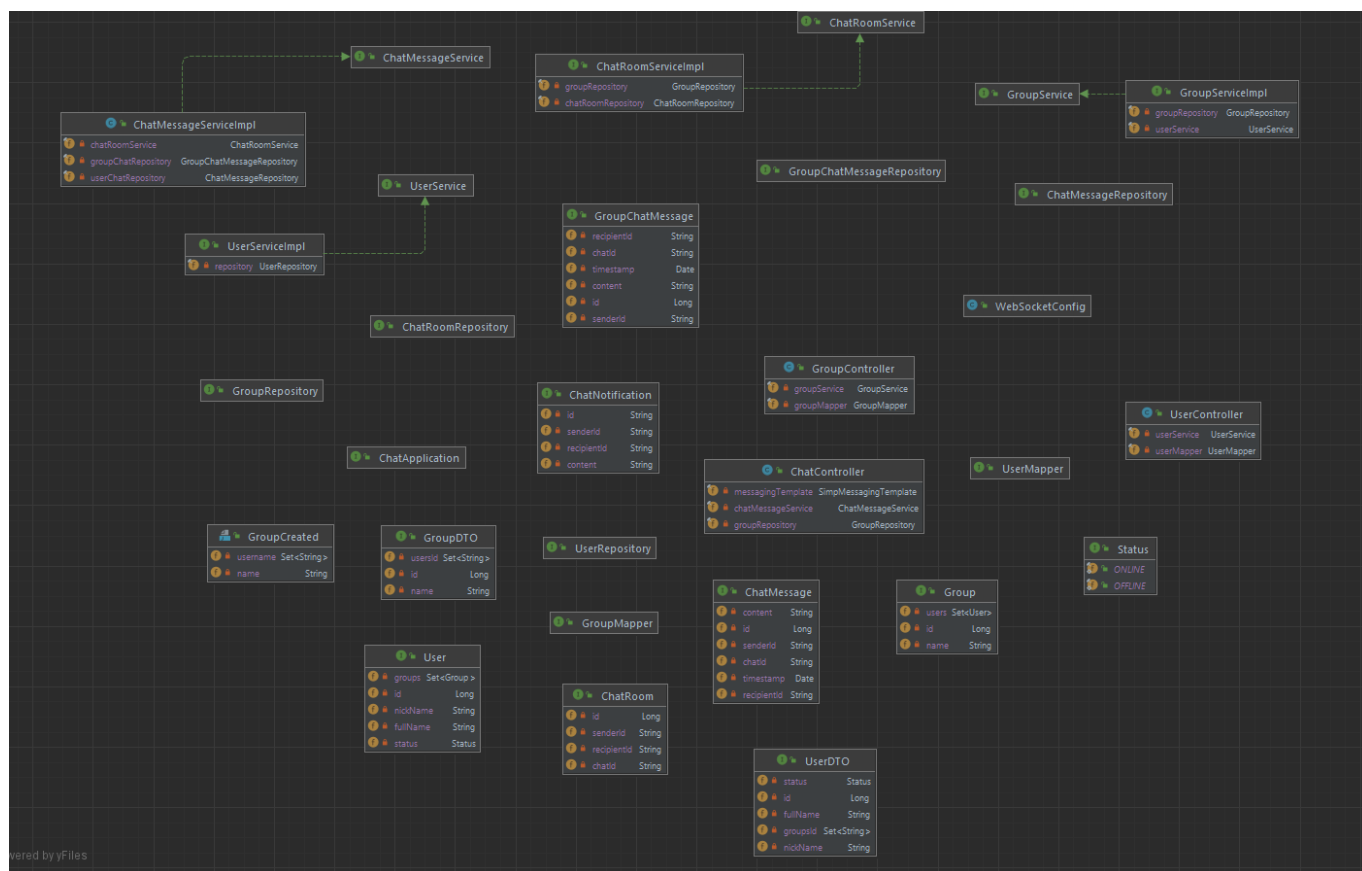
ДОДАТКИ

ДОДАТОК А

Javascript module dependencies



Діаграма класів системи



ДОДАТОК Б

Код проекту

Class AdaptedWebSocketConfigurer

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    hapancov *
    @Override
    public void configureMessageBroker(MessageBrokerRegistry registry) {
        registry.enableSimpleBroker( ...destinationPrefixes: "/user");
        registry.setApplicationDestinationPrefixes("/app");
        registry.setUserDestinationPrefix("/user");
    }

    hapancov
    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint( ...paths: "/ws")
            .withSockJS();
    }

    hapancov
    @Override
    public boolean configureMessageConverters(List<MessageConverter> messageConverters) {
        DefaultContentTypeResolver resolver = new DefaultContentTypeResolver();
        resolver.setDefaultMimeType(MimeTypeUtils.APPLICATION_JSON);
        MappingJackson2MessageConverter converter = new MappingJackson2MessageConverter();
        converter.setObjectMapper(new ObjectMapper());
        converter.setContentTypeResolver(resolver);
        messageConverters.add(converter);
        return false;
    }
}
```

ChatController

```

@Controller
@RequiredArgsConstructor
public class ChatController {

    2 usages
    private final SimpMessagingTemplate messagingTemplate;
    4 usages
    private final ChatMessageService chatMessageService;
    1 usage
    private final GroupRepository groupRepository;

    ▲ hapancov
    @PostMapping("/chat")
    public void processMessage(@Payload ChatMessage chatMessage) {
        ChatMessage savedMsg = chatMessageService.save(chatMessage);
        messagingTemplate.convertAndSendToUser(
            chatMessage.getRecipientId(), destination: "/queue/messages",
            new ChatNotification(
                savedMsg.getId(),
                savedMsg.getSenderId(),
                savedMsg.getRecipientId(),
                savedMsg.getContent()
            )
        );
    }

    new *
    @PostMapping("/group/chat")
    public void processMessageForGroup(@Payload GroupChatMessage chatMessage) {
        GroupChatMessage savedMsg = chatMessageService.saveForGroup(chatMessage);
        Group group = groupRepository.findByName(chatMessage.getRecipientId().replaceAll( regex: "group", replacement: ""));
        Set<User> users = group.getUsers();
        for (User user : users) {
            messagingTemplate.convertAndSendToUser(
                user.getNickName(), destination: "/queue/messages",
                new ChatNotification(
                    savedMsg.getId(),
                    savedMsg.getSenderId(),
                    savedMsg.getRecipientId(),
                    savedMsg.getContent()
                )
            );
        }
    }

    ▲ hapancov
    @GetMapping("/{senderId}/{recipientId}")
    public ResponseEntity<List<ChatMessage>> findChatMessages(@PathVariable String senderId,
                                                             @PathVariable String recipientId) {
        return ResponseEntity
            .ok(chatMessageService.findChatMessages(senderId, recipientId));
    }

    new *
    @GetMapping("/{senderId}/{selectedGroupId}")
    public ResponseEntity<List<GroupChatMessage>> findGroupChatMessages(@PathVariable String senderId, @PathVariable String selectedGroupId) {
        return ResponseEntity
            .ok(chatMessageService.findGroupChatMessages(senderId, selectedGroupId));
    }
}

```

GroupController

```

@Controller
@RequiredArgsConstructor

public class GroupController {
    2 usages
    private final GroupService groupService;
    2 usages
    private final GroupMapper groupMapper;

    hapancov *
    @PostMapping("/group.createGroup")
    @SendTo("/user/public")
    public GroupDTO createGroup(@Payload GroupCreated group) {
        return groupMapper.GroupDTO(groupService.saveGroupAsGroupCreated(group)) ;
    }

    hapancov *
    @GetMapping("/group")
    public ResponseEntity<List<GroupDTO>> findConnectedUsers() {
        System.out.println(21);
        return ResponseEntity.ok(groupMapper.GroupDTOList(groupService.findConnectedUsers()));
    }
}

```

UserController

```

@Controller
@RequiredArgsConstructor
public class UserController {

    3 usages
    private final UserService userService;
    2 usages
    private final UserMapper userMapper;

    hapancov *
    @PostMapping("/user.addUser")
    @SendTo("/user/public")
    public UserDTO addUser(
        @Payload User user
    ) {
        userService.saveUser(user);
        return userMapper.toUserDTO(user) ;
    }

    hapancov
    @PostMapping("/user.disconnectUser")
    @SendTo("/user/public")
    public User disconnectUser(
        @Payload User user
    ) {
        userService.disconnect(user);
        return user;
    }

    hapancov *
    @GetMapping("/users")
    public ResponseEntity<List<UserDTO>> findConnectedUsers() {
        return ResponseEntity.ok(userMapper.toUserDTOList(userService.findConnectedUsers()) );
    }
}

```

GroupMapper

```

@Mapper(componentModel = "spring")
public interface GroupMapper {

    1 usage
    @Mapping(target = "userId", source = "users")
    GroupDTO GroupDTO(Group group);

    default Set<String> mapUsersToUserIds(Set<User> users) {
        return users.stream() Stream<User>
            .map(User::getId) Stream<Long>
            .map(String::valueOf) Stream<String>
            .collect(Collectors.toSet());
    }

    1 usage
    List<GroupDTO> GroupDTOList(List<Group> groups);
}

```

UserMapper

```

@Mapper(componentModel = "spring")
public interface UserMapper {

    1 usage
    @Mapping(target = "groupId", source = "groups")
    UserDTO toUserDTO(User user);

    default Set<String> mapGroupsToIds(Set<Group> groups) {
        return groups.stream() Stream<Group>
            .map(Group::getId) Stream<String>
            .collect(Collectors.toSet());
    }

    1 usage
    List<UserDTO> toUserDTOList(List<User> connectedUsers);
}

```

GroupDTO

```

public class GroupDTO {
    2 usages
    private Long id;
    2 usages
    private String name;
    2 usages
    private Set<String> usersId = new HashSet<>();

    public Long getId() { return id; }

    public void setId(Long id) { this.id = id; }

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }

    public Set<String> getUsersId() { return usersId; }

    1 usage
    public void setUsersId(Set<String> usersId) { this.usersId = usersId; }
}

```

UserDTO

```

public class UserDTO {
    2 usages
    private Long id;
    2 usages
    private String nickName;
    2 usages
    private String fullName;
    2 usages
    private Status status;
    2 usages
    private Set<String> groupsId = new HashSet<>();
}

```

ChatMessage

```

@Entity
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class ChatMessage {

    2 usages
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    2 usages
    private String chatId;

    2 usages
    private String senderId;

    2 usages
    private String recipientId;

    2 usages
    private String content;

    2 usages
    private Date timestamp;
}

```

ChatNotification

```

@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class ChatNotification {
    private String id;
    private String senderId;
    private String recipientId;
    private String content;
}

```

ChatRoom


```

@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@Builder
@Entity
public class ChatRoom {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String chatId;
    private String senderId;
    private String recipientId;
}

```

Group

```

@Getter
@Setter
@Entity
@Table(name = "groups")
public class Group {
    2 usages
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    2 usages
    private String name;

    3 usages
    @ManyToMany(fetch = FetchType.EAGER)
    @JoinTable(
        name = "user_group",
        joinColumns = @JoinColumn(name = "group_id"),
        inverseJoinColumns = @JoinColumn(name = "user_id")
    )
    private Set<User> users = new HashSet<>();
}

```

GroupChatMessage

```

@AllArgsConstructor
@NoArgsConstructor
@Builder
@Entity
public class GroupChatMessage {

    3 usages
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    3 usages
    private String chatId;
    3 usages
    private String senderId;
    3 usages
    private String recipientId;
    3 usages
    private String content;
    3 usages
    private Date timestamp;

```

GroupCreated

```

@Data
public class GroupCreated {
    private String name;
    @JsonProperty("users")
    private Set<String> username = new HashSet<>();
}

```

Status

```

public enum Status {
    2 usages
    ONLINE, OFFLINE
}

```

User

```

@Getter
@Setter
@Entity
@Table(name = "users")
public class User {
    2 usages
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    2 usages
    private String nickName;
    2 usages
    private String fullName;
    2 usages
    private Status status;

    2 usages
    @ManyToMany(mappedBy = "users")
    private Set<Group> groups = new HashSet<>();
}

```

ChatMessageRepository

```

public interface ChatMessageRepository extends JpaRepository<ChatMessage, String> {
    1 usage 1 hapancov
    List<ChatMessage> findByChatId(String chatId);
}

```

ChatRoomRepository

```

public interface ChatRoomRepository extends JpaRepository<ChatRoom, String> {
    2 usages 1 hapancov
    Optional<ChatRoom> findBySenderIdAndRecipientId(String senderId, String recipientId);
}

```

GroupChatMessageRepository

```

public interface GroupChatMessageRepository extends JpaRepository<GroupChatMessage, String> {
    2 usages
    @Query(value = "SELECT * FROM group_chat_message WHERE chat_id LIKE %:suffix", nativeQuery = true)
    List<GroupChatMessage> findAllByChatIdEndingWith(@Param("suffix") String suffix);
}

```

GroupRepository

```

public interface GroupRepository extends JpaRepository<Group, String> {
    2 usages new *
    Group findByName(String recipientId);
}

```

UserRepository

```

public interface UserRepository extends JpaRepository<User, String> {
    1 usage  hapancov
    List<User> findAllByStatus(Status status);

    3 usages  hapancov
    Optional<User> findByNickName(String nickName);
}

```

ChatMessageServiceImpl

```

@Service
@RequiredArgsConstructor
public class ChatMessageServiceImpl implements ChatMessageService {
    2 usages
    private final ChatMessageRepository userChatRepository;
    4 usages
    private final ChatRoomService chatRoomService;
    3 usages
    private final GroupChatMessageRepository groupChatRepository;

    1 usage  hapancov *
    @Override
    public ChatMessage save(ChatMessage chatMessage) {
        var chatId :String = chatRoomService
            .getChatRoomId(chatMessage.getSenderId(), chatMessage.getRecipientId(), createNewRoomIfNotExists: true)
            .orElseThrow();
        chatMessage.setChatId(chatId);
        userChatRepository.save(chatMessage);
        return chatMessage;
    }

    1 usage  new *
    @Override
    public GroupChatMessage saveForGroup(GroupChatMessage chatMessage) {
        var chatId :String = chatRoomService
            .getChatRoomId(chatMessage.getSenderId(), recipientId: "group" + chatMessage.getRecipientId(), createNewRoomIfNotExists: true)
            .orElseThrow();
        chatMessage.setChatId(chatId);
        groupChatRepository.save(chatMessage);
        return chatMessage;
    }
}

```

```

1 usage  hapancov *
@Override
public List<ChatMessage> findChatMessages(String senderId, String recipientId) {
    var chatId : Optional<String> = chatRoomService.getChatRoomId(senderId, recipientId, createNewRoomIfNotExists: false);
    System.out.println(chatId);
    return chatId.map(userChatRepository::findByChatId).orElse(new ArrayList<>());
}

1 usage  hapancov *
@Override
public List<GroupChatMessage> findGroupChatMessages(String senderId, String selectedGroupId) {
    selectedGroupId= "group" + selectedGroupId;
    var chatId : Optional<String> = chatRoomService.getChatRoomId(senderId, selectedGroupId, createNewRoomIfNotExists: false);
    String a = getChatIdSuffix(chatId.orElse( other: "1_0"));
    List<GroupChatMessage> messages = groupChatRepository.findAllByChatIdEndingWith(a);
    return groupChatRepository.findAllByChatIdEndingWith(a);
}

1 usage  new *
public String getChatIdSuffix(String chatId) {
    String[] parts = chatId.split( regex: "_");
    if (parts.length == 2) {
        return parts[1];
    } else {
        throw new IllegalArgumentException("Invalid chatId format: " + chatId);
    }
}
}

```

ChatRoomServiceImpl

```

@Service
@RequiredArgsConstructor
public class ChatRoomServiceImpl implements ChatRoomService {

    6 usages
    private final ChatRoomRepository chatRoomRepository;

    1 usage
    private final GroupRepository groupRepository;

    4 usages  hapancov *
    @Override
    public Optional<String> getChatRoomId(
        String senderId,
        String recipientId,
        boolean createNewRoomIfNotExists
    ) {
        if (checkPattern(recipientId)) {
            return chatRoomRepository
                .findBySenderIdAndRecipientId(senderId, recipientId) Optional<ChatRoom>
                .map(ChatRoom::getChatId) Optional<String>
                .or(() -> {
                    if (createNewRoomIfNotExists) {
                        var chatId : String = createChatIdForGroup(senderId, recipientId);
                        return Optional.of(chatId);
                    }

                    return Optional.empty();
                });
        } else {

```

```

    } else {
        return chatRoomRepository
            .findBySenderIdAndRecipientId(senderId, recipientId) Optional<ChatRoom>
            .map(ChatRoom::getChatId) Optional<String>
            .or(() -> {
                if (createNewRoomIfNotExists) {
                    var chatId :String = createChatId(senderId, recipientId);
                    return Optional.of(chatId);
                }

                return Optional.empty();
            });
    }
}

1 usage hapancov
@Override
public String createChatId(String senderId, String recipientId) {
    var chatId :String = String.format("%s_%s", senderId, recipientId);

    ChatRoom senderRecipient = ChatRoom
        .builder()
        .chatId(chatId)
        .senderId(senderId)
        .recipientId(recipientId)
        .build();

    ChatRoom recipientSender = ChatRoom
        .builder()
        .chatId(chatId)
        .senderId(recipientId)
        .recipientId(senderId)
        .build();

    chatRoomRepository.save(senderRecipient);
    chatRoomRepository.save(recipientSender);

    chatRoomRepository.save(senderRecipient);
    chatRoomRepository.save(recipientSender);

    return chatId;
}

1 usage new *
private String createChatIdForGroup(String senderId, String recipientId) {
    var chatId :String = String.format("%s_%s", senderId, recipientId);

    ChatRoom senderRecipient = ChatRoom
        .builder()
        .chatId(chatId)
        .senderId(senderId)
        .recipientId(recipientId)
        .build();

    chatRoomRepository.save(senderRecipient);

    Group group = groupRepository.findByName(recipientId.replaceAll( regex: "group", replacement: ""));
    Set<User> users = group.getUsers();
    for (User u : users) {
        if (!senderId.equals(u.getNickName())) {
            chatId = String.format("%s_%s", u.getNickName(), recipientId);
            ChatRoom senderRecipient2 = ChatRoom
                .builder()
                .chatId(chatId)
                .senderId(u.getNickName())
                .recipientId(recipientId)

```

```

        .builder()
        .chatId(chatId)
        .senderId(u.getNickName())
        .recipientId(recipientId)
        .build();
        chatRoomRepository.save(senderRecipient2);
    }
}
return chatId;
}
}

```

```

1 usage new *
private static boolean checkPattern(String recipientId) {
    String regex = "^group.*$";
    Pattern pattern = Pattern.compile(regex);
    Matcher matcher = pattern.matcher(recipientId);
    return matcher.matches();
}

```

GroupServiceImpl

```

@Service
@RequiredArgsConstructor
public class GroupServiceImpl implements GroupService {
    3 usages
    private final GroupRepository groupRepository;
    1 usage
    private final UserService userService;

    hapancov
    @Override
    public void saveGroup(Group group) { groupRepository.save(group); }

    1 usage new *
    @Override
    public Group saveGroupAsGroupCreated(GroupCreated group) {
        Group group1 = new Group();
        group1.setName(group.getName());
        System.out.println(group.getUsername());
        group1.setUsers(userService.findAllByUsername(group.getUsername()));
        groupRepository.save(group1);
        return group1;
    }

    1 usage new *
    @Override
    public List<Group> findConnectedUsers() { return groupRepository.findAll(); }
}

```

UserServiceImpl

```

public class UserServiceImpl implements UserService {

    7 usages
    private final UserRepository repository;

    1 usage  hapancov *
    @Override
    public void saveUser(User user) {
        user.setStatus(Status.ONLINE);

        Optional<User> existingUser = repository.findByNickName(user.getNickName());
        if (existingUser.isPresent()) {
            User userToUpdate = existingUser.get();
            userToUpdate.setStatus(user.getStatus());
            repository.save(userToUpdate);
        } else {
            repository.save(user);
        }
    }

    1 usage  hapancov *
    @Override
    public void disconnect(User user) {
        var storedUser :User = repository.findByNickName(user.getNickName()).orElse( other: null);
        if (storedUser != null) {
            storedUser.setStatus(Status.OFFLINE);
            repository.save(storedUser);
        }
    }

    1 usage  hapancov *
    @Override
    public void disconnect(User user) {
        var storedUser :User = repository.findByNickName(user.getNickName()).orElse( other: null);
        if (storedUser != null) {
            storedUser.setStatus(Status.OFFLINE);
            repository.save(storedUser);
        }
    }

    1 usage  hapancov *
    @Override
    public List<User> findConnectedUsers() { return repository.findAllByStatus(Status.ONLINE); }

    1 usage  new *
    @Override
    public Set<User> findAllByUsername(Set<String> usernames) {
        Set<User> users = new HashSet<>();
        for (String nick : usernames) {
            User user = repository.findByNickName(nick)
                .orElseThrow(() -> new NoSuchElementException("User not found with nickname: " + nick));
            users.add(user);
        }

        return users;
    }
}

```

Main.js


```

'use strict';
const groupPage = document.querySelector('#createGroupForm');
const usernamePage = document.querySelector('#username-page');
const chatPage = document.querySelector('#chat-page');
const usernameForm = document.querySelector('#usernameForm');
const messageForm = document.querySelector('#messageForm');
const messageInput = document.querySelector('#message');
const connectingElement = document.querySelector('.connecting');
const chatArea = document.querySelector('#chat-messages');
const logout = document.querySelector('#logout');
const usersSelect = document.querySelector('#users_group_list');
let stompClient = null;
let nickname = null;
let fullname = null;
let userID = null;
let selectedUserId = null;
let selectedGroupId = null;
let isContactsListVisible = true;
let clickedGroup = null;
document.getElementById('createGroupBtn').addEventListener('click',
showCreateGroupForm);
document.getElementById('backToChatContainer').addEventListener('click', backToChat);

function backToChat(event) {
    event.preventDefault();
    document.getElementById('chat-page').classList.remove('hidden');
    document.getElementById('create-group-page').classList.add('hidden');
}

//створення групи

function showCreateGroupForm() {
    document.getElementById('chat-page').classList.add('hidden');
    document.getElementById('create-group-page').classList.remove('hidden');
    findAndDisplayConnectedUsersForGroup();
}

async function findAndDisplayConnectedUsersForGroup() {
    const connectedUsersResponse = await fetch('/users');
    let connectedUsers = await connectedUsersResponse.json();
    connectedUsers = connectedUsers.filter(user => user.nickname !== nickname);
    const connectedUsersList = document.getElementById('users_group_list');
    connectedUsersList.innerHTML = '';

    connectedUsers.forEach(user => {
        appendUserElementForGroup(user, connectedUsersList);
    });
}

function appendUserElementForGroup(user, connectedUsersList) {
    const listItem = document.createElement('option');
    listItem.classList.add('user-item');
    listItem.id = user.nickname;
    const usernameSpan = document.createElement('span');
    usernameSpan.textContent = user.fullName;

    listItem.appendChild(usernameSpan);
    connectedUsersList.appendChild(listItem);
}

function selectGroupInfo(event) {
    name = document.querySelector('#groupName').value.trim();

```

```

let users = Array.from(usersSelect.selectedOptions).map(option => option.id)
users.push(nickname);
if (name && users.length > 0) {

    stompClient.send("/app/group.createGroup",
        {},
        JSON.stringify({name: name, users: users})
    );
    document.getElementById('chat-page').classList.remove('hidden');
    document.getElementById('create-group-page').classList.add('hidden');
}
setTimeout(findAndDisplayConnectedGroup, 1000)
event.preventDefault();
}

//группы
async function findAndDisplayConnectedGroup() {

    const connectedGroupResponse = await fetch('/group');
    let connectedGroup = await connectedGroupResponse.json();
    connectedGroup = connectedGroup.filter(group => {
        return group.userId.includes(userID.toString());
    });

    const connectedGroupList = document.getElementById('connectedGroup');
    connectedGroupList.innerHTML = '';

    connectedGroup.forEach(group => {
        appendGroupElement(group, connectedGroupList);
        if (connectedGroup.indexOf(group) < connectedGroup.length - 1) {
            const separator = document.createElement('li');
            separator.classList.add('separator');
            connectedGroupList.appendChild(separator);
        }
    });
}

function appendGroupElement(group, connectedUsersList) {

    const listItem = document.createElement('li');
    listItem.classList.add('user-item');
    listItem.id = group.name;

    const userImage = document.createElement('img');
    userImage.src = '../img/group_icon.png';
    userImage.alt = group.fullName;

    const usernameSpan = document.createElement('span');
    usernameSpan.textContent = group.name;

    listItem.appendChild(userImage);
    listItem.appendChild(usernameSpan);

    listItem.addEventListener('click', groupItemClick);

    connectedUsersList.appendChild(listItem);
}

// переписка в группах

function groupItemClick(event) {
    document.querySelectorAll('.user-item').forEach(item => {

```

```

        item.classList.remove('active');
    });
    messageForm.classList.remove('hidden');

    clickedGroup = event.currentTarget;

    clickedGroup.classList.add('active');

    selectedGroupId = clickedGroup.getAttribute('id');

    fetchAndDisplayGroupChat().then();

    const nbrMsg = clickedGroup.querySelector('.nbr-msg');
    nbrMsg.classList.add('hidden');
    nbrMsg.textContent = '0';
}

async function fetchAndDisplayGroupChat() {
    const userChatResponse = await
    fetch(`/group/messages/${nickname}/${selectedGroupId}`);
    const userChat = await userChatResponse.json();

    chatArea.innerHTML = '';
    userChat.forEach(chat => {
        displayMessageGroup(chat.senderId, chat.content);
    });
    chatArea.scrollTop = chatArea.scrollHeight;
}

function displayMessageGroup(senderId, content) {
    const messageContainer = document.createElement('div');
    messageContainer.classList.add('message');
    if (senderId === nickname) {
        messageContainer.classList.add('sender');
    } else {
        messageContainer.classList.add('receiver');
        const user = document.createElement('p');
        user.classList.add('username_bold');
        user.textContent = senderId;
        messageContainer.appendChild(user);
    }
    const message = document.createElement('p');
    message.textContent = content;
    messageContainer.appendChild(message);
    chatArea.appendChild(messageContainer);
}

//далі переписка користувачів

function connect(event) {
    nickname = document.querySelector('#nickname').value.trim();
    fullname = document.querySelector('#fullname').value.trim();

    if (nickname && fullname) {
        usernamePage.classList.add('hidden');
        chatPage.classList.remove('hidden');

        const socket = new SockJS('/ws');
        stompClient = Stomp.over(socket);

        stompClient.connect({}, onConnected, onError);
    }
}

```

```

    }
    event.preventDefault();
}

function onConnected() {
    stompClient.subscribe(`/user/${nickname}/queue/messages`, onMessageReceived);
    stompClient.subscribe(`/user/public`, onMessageReceived);
    stompClient.send("/app/user.addUser",
        {},
        JSON.stringify({nickname: nickname, fullName: fullname, status: 'ONLINE'}))
    );
    document.querySelector('#connected-user-fullname').textContent = fullname;
    findAndDisplayConnectedUsers().then();
    findAndDisplayConnectedGroup().then();
}

async function findAndDisplayConnectedUsers() {

    const connectedUsersResponse = await fetch('/users');
    let connectedUsers = await connectedUsersResponse.json();
    const user = connectedUsers.find(user => user.nickname === nickname);

    if (user) {
        userID = user.id;
    } else {
        console.log("Користувача з таким нікнеймом не знайдено.");
    }
    connectedUsers = connectedUsers.filter(user => user.nickname !== nickname);
    const connectedUsersList = document.getElementById('connectedUsers');
    connectedUsersList.innerHTML = '';

    connectedUsers.forEach(user => {
        appendUserElement(user, connectedUsersList);
        if (connectedUsers.indexOf(user) < connectedUsers.length - 1) {
            const separator = document.createElement('li');
            separator.classList.add('separator');
            connectedUsersList.appendChild(separator);
        }
    });
}

function appendUserElement(user, connectedUsersList) {
    const listItem = document.createElement('li');
    listItem.classList.add('user-item');
    listItem.id = user.nickname;

    const userImage = document.createElement('img');
    userImage.src = '../img/user_icon.png';
    userImage.alt = user.fullName;

    const usernameSpan = document.createElement('span');
    usernameSpan.textContent = user.fullName;

    const receivedMsgs = document.createElement('span');
    receivedMsgs.textContent = '0';
    receivedMsgs.classList.add('nbr-msg', 'hidden');

    listItem.appendChild(userImage);
    listItem.appendChild(usernameSpan);
    listItem.appendChild(receivedMsgs);

    listItem.addEventListener('click', userItemClick);
}

```

```

    connectedUsersList.appendChild(listItem);
}

function userItemClick(event) {
    document.querySelectorAll('.user-item').forEach(item => {
        item.classList.remove('active');
    });
    messageForm.classList.remove('hidden');

    const clickedUser = event.currentTarget;
    clickedUser.classList.add('active');

    selectedUserId = clickedUser.getAttribute('id');
    fetchAndDisplayUserChat().then();

    const nbrMsg = clickedUser.querySelector('.nbr-msg');
    nbrMsg.classList.add('hidden');
    nbrMsg.textContent = '0';
}

function displayMessage(senderId, content) {
    const messageContainer = document.createElement('div');
    messageContainer.classList.add('message');
    if (senderId === nickname) {
        messageContainer.classList.add('sender');
    } else {
        messageContainer.classList.add('receiver');
    }
    const message = document.createElement('p');
    message.textContent = content;
    messageContainer.appendChild(message);
    chatArea.appendChild(messageContainer);
}

async function fetchAndDisplayUserChat() {
    const userChatResponse = await fetch(`/messages/${nickname}/${selectedUserId}`);
    const userChat = await userChatResponse.json();
    chatArea.innerHTML = '';
    userChat.forEach(chat => {
        displayMessage(chat.senderId, chat.content);
    });
    chatArea.scrollTop = chatArea.scrollHeight;
}

function onError() {
    connectingElement.textContent = 'Could not connect to WebSocket server. Please refresh this page to try again!';
    connectingElement.style.color = 'red';
}

function sendMessage(event) {
    const messageContent = messageInput.value.trim();
    if (messageContent && stompClient) {
        if (isContactsListVisible) {
            const chatMessage = {
                senderId: nickname,
                recipientId: selectedUserId,
                content: messageInput.value.trim(),
                timestamp: new Date()
            };
            stompClient.send("/app/chat", {}, JSON.stringify(chatMessage));

```

```

        displayMessage(nickname, messageInput.value.trim());
        messageInput.value = '';
    } else {
        const chatMessage = {
            senderId: nickname,
            recipientId: selectedGroupId,
            content: messageInput.value.trim(),
            timestamp: new Date()
        };

        messageInput.value = '';
        stompClient.send("/app/group/chat", {}, JSON.stringify(chatMessage));
    }

}

chatArea.scrollTop = chatArea.scrollHeight;
event.preventDefault();
}

async function onMessageReceived(payload) {
    await findAndDisplayConnectedUsers();
    await findAndDisplayConnectedGroup();

    console.log('Message received', payload);
    const message = JSON.parse(payload.body);
    if (message.recipientId === selectedGroupId) {
        displayMessageGroup(message.senderId, message.content);
        chatArea.scrollTop = chatArea.scrollHeight;
        document.querySelector(`[id="${selectedGroupId}"]`).classList.add('active');
    } else {
        if (selectedUserId && selectedUserId === message.senderId) {
            displayMessage(message.senderId, message.content);
            chatArea.scrollTop = chatArea.scrollHeight;
        }

        if (selectedUserId) {
            document.querySelector(`#${selectedUserId}`).classList.add('active');
        } else {
            messageForm.classList.add('hidden');
        }

        const notifiedUser = document.querySelector(`#${message.senderId}`);
        if (notifiedUser && !notifiedUser.classList.contains('active')) {
            const nbrMsg = notifiedUser.querySelector('.nbr-msg');
            nbrMsg.classList.remove('hidden');
            nbrMsg.textContent = '';
        }
    }
}

function onLogout() {
    stompClient.send("/app/user.disconnectUser",
        {},
        JSON.stringify({nickName: nickname, fullName: fullname, status: 'OFFLINE'}));
    window.location.reload();
}

usernameForm.addEventListener('submit', connect, true);

```

```

groupPage.addEventListener('submit', selectGroupInfo, true);
messageForm.addEventListener('submit', sendMessage, true);
logout.addEventListener('click', onLogout, true);
window.onbeforeunload = () => onLogout();

document.getElementById('toggleListBtn').addEventListener('click', function () {
    const usersListContainer = document.getElementById('usersListContainer');
    const groupListContainer = document.getElementById('groupListContainer');
    const toggleListBtn = document.getElementById('toggleListBtn');

    if (isContactsListVisible) {
        usersListContainer.classList.add('hidden');
        groupListContainer.classList.remove('hidden');
        toggleListBtn.textContent = 'Contacts';
    } else {
        usersListContainer.classList.remove('hidden');
        groupListContainer.classList.add('hidden');
        toggleListBtn.textContent = 'Groups';
    }

    isContactsListVisible = !isContactsListVisible;
});

```

Index.html

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="stylesheet" href="css/main.css">
    <title>Chat Application</title>
</head>
<body>

<h2>Office communicator</h2>

<div class="user-form " id="username-page">
    <h2>Enter Chatroom</h2>
    <form id="usernameForm">
        <label for="nickname">Nickname:</label>
        <input type="text" id="nickname" name="nickname" required>

        <label for="fullname">Real Name:</label>
        <input type="text" id="fullname" name="realname" required>

        <button type="submit">Enter Chatroom</button>
    </form>
</div>

<div class="chat-container hidden" id="chat-page" >
    <div class="users-list">
        <div class="users-list-container" id="usersListContainer">
            <h2>Online Users</h2>
            <ul id="connectedUsers">
            </ul>
        </div>
        <div class="users-list-container hidden" id="groupListContainer">
            <h2>Groups</h2>
            <ul id="connectedGroup">
            </ul>
        </div>
    </div>

```

```

        <div>
            <p id="connected-user-fullname"></p>
            <a class="logout" href="javascript:void(0)" id="logout">Logout</a>
            <button id="toggleListBtn">Groups</button>
            <button id="createGroupBtn">Create Group</button>
        </div>
    </div>

    <div class="chat-area">
        <div class="chat-area" id="chat-messages">
        </div>

        <form id="messageForm" name="messageForm" class="hidden">
            <div class="message-input">
                <input autocomplete="off" type="text" id="message" placeholder="Type
your message...">
                <button>Send</button>
            </div>
        </form>
    </div>
</div>

<div id="create-group-page" class="hidden">

    <h2>Create Group</h2>
    <form id="createGroupForm">
        <label for="groupName">Group Name:</label>
        <input type="text" id="groupName" name="groupName" required>
        <br>
        <label for="users_group_list">Select Users to Add:</label>
        <select id="users_group_list" name="users" multiple>
        </select>
        <br>
        <button id="createGroupBtnForm" >Create Group</button>
        <button id="backToChatContainer" class="back-to-chat" >Back</button>
    </form>
</div>

<script src="https://cdnjs.cloudflare.com/ajax/libs/sockjs-
client/1.1.4/sockjs.min.js"></script>
<script
src="https://cdnjs.cloudflare.com/ajax/libs/stomp.js/2.3.3/stomp.min.js"></script>
<script src="/js/main.js"></script>

</body>
</html>

```