

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 IA-12 Семестр 5

ЗАВДАННЯ

на курсову роботу студента

Молчанова Михайла Валерійовича

(прізвище, ім'я, по батькові)

1. Тема роботи: «Terminal Messenger»

2. Строк здачі студентом закінченої роботи:

3. Вихідні дані до роботи: тема «Terminal Messenger», опис програми та основних можливостей: Повинен бути сервер, який буде надавати послуги месенджера, та клієнт що вміє ними користуватись та надавати зручний користувацький інтерфейс в терміналі для користувача. Система повинна бути швидкою, надійною та безпечною.

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

Студент _____
(підпис)

Михайло МОЛЧАНОВ (Ім'я ПРИЗВИЩЕ)

Керівник _____
(підпис)

Олександр АМОНС (Ім'я ПРИЗВИЩЕ)

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

Тема **Terminal Messenger**

Курсова робота

З дисципліни «Технології розроблення програмного забезпечення»

Керівник

доц. Амонс О.А.

«Допущений до захисту»

Виконавець

ст. Молчанов М.В.

залікова книжка № IA –1220

гр. IA-12

(Особистий підпис керівника)

« » 2023р.

Захищений з оцінкою

(особистий підпис виконавця)

« » 2023р.

(оцінка) Члени

комісії:

(особистий підпис)

(розшифровка підпису)

(особистий підпис)

(розшифровка підпису)

Київ – 2023

ЗМІСТ

Вступ.....	6
1. Проектування системи	7
1.1. Огляд існуючих рішень.....	7
1.2. Загальний опис проекту.....	8
1.3. Вимоги до застосунків системи	9
1.3.1. Функціональні вимоги до системи	9
1.3.2. Нефункціональні вимоги до системи	10
1.4. Сценарій використання системи.....	12
1.5. Концептуальна модель системи	15
1.6. Вибір бази даних	16
1.7. Вибір мови програмування та середовища розробки.....	17
1.8. Діаграма розгортання системи.....	19
2. Реалізація компонентів системи	21
2.1. Структура бази даних.....	21
2.2. Архітектура системи	23
2.2.1. Специфікація системи.....	23
2.2.2. Діаграма послідовностей.....	24
2.2.3. Паттерн Redux.....	31
2.2.4. Паттерн Prototype	35
2.2.5. Паттерн Facade.....	39
2.2.6. Паттерн Адаптор	40
2.2.7. Паттерн Singleton.....	42
2.2.8. Паттерн Request-Reply	42
2.2.9. Паттерн Command Message	43
2.2.10. Паттерн Dead letter Queue	43
2.2.11. Мікро сервісна архітектура	44

2.2.12. Використання Device Authorization Flow	45
3. Інструкція користувача	48
3.1. Встановлення	48
3.2. Налаштування	49
3.3. Використання серверу.....	50
3.3.1. Обмін повідомленнями	50
3.3.2. Отримання попередніх повідомлень	55
3.3.3. Завантаження чатів користувача	55
3.3.4. Завантаження знайомих	56
3.3.5. Створення чату	56
3.3.6. Надсилання запрошення	57
3.3.7. Отримання запрошення в реальному часі	58
3.3.8. Отримання всіх запрошень користувача.....	58
3.3.9. Відхилення запрошення.....	58
3.3.10. Прийняття запрошення	59
3.3.11. Логін.....	62
4. Наступні кроки проекту	67
5. Висновок.....	68
6. Перелік використаних джерел.....	Error! Bookmark not defined.

Вступ

У наші дні, спілкування відіграє важливу роль у повсякденному житті людини. З давніх часів, коли єдиним способом комунікації на відстані були паперові листи, які подорожували місяцями, світ пройшов довгий шлях до миттєвого обміну повідомленнями. Ці листи, сповнені трепетних очікувань та хвилювань, часто губилися в шляху, залишаючи одержувачів у невіданні. Згодом, із прогресом технологій, людство вступило у нову еру, де комунікація перетворилася на швидкий, ефективний, та всеосяжний процес. Сучасний світ неможливо уявити без миттєвого обміну інформацією, який здійснюється завдяки численним месенджерам.

Незважаючи на величезну популярність месенджерів у нашему повсякденному житті, існує сфера, де інтеграція такого роду комунікаційних засобів ще не набула широкого розповсюдження - це сфера використання терміналів. Термінал, який часто асоціюється із складними технічними операціями та професійною роботою, рідко розглядається як платформа для обміну повідомленнями. Втім, уявіть, наскільки практичним було б мати можливість використовувати месенджер прямо у вашому терміналі - чи то під час роботи у Linux, підключені до хмарних сервісів через SSH, або навіть на Raspberry Pi.

Ця курсова робота присвячена розробці месенджера для терміналу, який може бути встановлений та безпечно використовуваний в різних умовах, відкриваючи нові горизонти у сфері миттєвого обміну повідомленнями. Хоча цей проект є лише доказом концепції (Proof of Concept), він має потенціал з'єднувати користувачів у різних середовищах, демонструючи нові можливості інтеграції месенджерів у технічно орієнтовані платформи.

1. ПРОЕКТУВАННЯ СИСТЕМИ

1.1. Огляд існуючих рішень

В сучасному світі технологій, де важливість надійності та зручності у використанні програмного забезпечення стрімко зростає, розгляд існуючих рішень у сфері месенджерів для терміналу набуває особливого значення. Один з варіантів, який вже існує на ринку, - це Messer [1], представлений на GitHub. Messer - це інструмент командного рядка для Facebook Messenger, який дозволяє користувачам надсилати та отримувати повідомлення через термінал. Хоча це рішення і відкриває нові можливості для використання месенджерів у нестандартних умовах, воно має декілька істотних обмежень.

Перш за все, інтерфейс Messer базується на використанні командного рядка, що може бути не зовсім зручно для звичайних користувачів, які не звичали до такого стилю взаємодії. Крім того, для виконання дій у Messer необхідно вводити специфічні команди, що вимагає певного рівня знань та навичок роботи з терміналом.

Ще одним значущим недоліком є питання безпеки. Для авторизації в Messer користувачам потрібно вводити дані для логіну безпосередньо в термінал, що може створювати потенційні ризики безпеки, особливо при використанні програми на загальнодоступних або незахищених комп'ютерах.

На відміну від Messer, наш проект прагне розробити месенджер для терміналу з інтерактивним інтерфейсом, який буде зручнішим і інтуїтивно зрозумілішим для користувачів. Основна увага приділяється створенню інтерфейсу, який буде мінімізувати необхідність введення складних команд, замість цього пропонуючи більш графічний та дружній користувацький інтерфейс.

Також, особлива увага приділяється питанням безпеки. Наша система планує використовувати більш безпечні методи аутентифікації, що знижує ризики пов'язані з введенням конфіденційної інформації, та забезпечує кращий захист даних користувачів. Це включає в себе використання сучасних протоколів шифрування та безпечних методів зберігання облікових даних, що є критично важливим для забезпечення конфіденційності користувачів у цифровому світі.

1.2. Загальний опис проекту

Основна мета цього проекту полягає у створенні месенджера, який надасть користувачам можливість зручно та ефективно обмінюватися повідомленнями через термінал. Цей проект прагне вирішити проблему відсутності інтуїтивно зрозумілих та легких у використанні месенджерів у середовищі терміналу, пропонуючи користувачам новий спосіб комунікації, який інтегрується з їхнім звичним робочим простором.

Програма, яка розробляється в рамках цього проекту, має бути швидкою та ефективною. Швидкість роботи є ключовим фактором, оскільки вона забезпечує користувачам можливість безперервної та незатриманої комунікації. Також, програма має відповідати сучасним принципам об'єктно-орієнтованого програмування, що сприятиме гнучкості, масштабованості та легкості підтримки проекту.

Розробка рішення поділяється на дві основні частини: клієнтську частину та серверну частину.

1. Сервер: Серверна частина проекту відповідатиме за обробку та управління повідомленнями. Сервер має надавати надійні та безперебійні послуги з обміном повідомленнями, гарантуючи високий рівень безпеки та конфіденційності інформації. Основні завдання сервера включають в себе автентифікацію користувачів, зберігання та пересилання повідомень, а також управління станом онлайн-сесій користувачів.

2. Клієнт: Клієнтська частина зосереджена на розробці зручного інтерфейсу для взаємодії користувача з сервером. Це включає створення інтуїтивно зрозумілого графічного інтерфейсу, який дозволяє користувачам легко надсилювати, отримувати та організовувати свої повідомлення. Важливим аспектом є також забезпечення високого рівня безпеки при обміні даними з сервером, а також оптимізація інтерфейсу для забезпечення максимальної ефективності і зручності користувача.

У підсумку, реалізація цього проекту забезпечить користувачам новий інструмент для спілкування, який інтегрується безпосередньо у їх звичне робоче середовище терміналу, пропонуючи при цьому якісний, bezpechnyj та ефективний досвід користування.

1.3. Вимоги до застосунків системи

1.3.1. Функціональні вимоги до системи

Цей розділ надає детальний опис функціональних вимог до розроблюваного месенджера для терміналу. Основні функції, які повинна підтримувати система, охоплюють широкий спектр можливостей комунікації та взаємодії з іншими користувачами.

1. Створення чату: Користувач має мати можливість створювати чати для обміну повідомленнями з одним або декількома співрозмовниками.
2. Запрошення до чату: Користувачі повинні мати змогу запрошувати інших користувачів до існуючих чатів.
3. Прийняття або відхилення запрошень до чату: Користувачі мають можливість приймати або відхилити запрошення до участі в чатах.
4. Надсилання повідомлень до чату: Система має забезпечувати можливість надсилання текстових повідомлень у чатах.
5. Пошук інших користувачів: Користувачам надається можливість шукати інших учасників системи за ім'ям, прізвищем або іншими ідентифікаційними даними.
6. Отримання нових повідомлень в реальному часі: Програма має підтримувати функціонал отримання повідомлень у режимі реального часу.
7. Завантаження старих повідомлень: Користувачі повинні мати можливість переглядати історію чату, включаючи доступ до старих повідомлень.
8. Отримання запрошень до чату в реальному часі: Важливою складовою системи є можливість отримувати запрошення до чату від інших користувачів у реальному часі. Ця функція забезпечить користувачам миттєве сповіщення про нові запрошення, дозволяючи своєчасно реагувати на них. Це важливо для підтримки безперервної та ефективної комунікації, а також додає зручності у використанні месенджера, оскільки користувачі не пропустять важливі запрошення та можливості для обміну інформацією.
9. Перегляд інформації про користувачів: Можливість переглядати профілі інших користувачів з основною інформацією та статусом.

10. Створення акаунту: Користувачі мають мати можливість створювати власні акаунти, вказуючи необхідні персональні дані.

11. Безпечна авторизація: Система має забезпечити безпечний метод логіну, який захищає конфіденційні дані користувача.

Ці функціональні вимоги створюють основу для розробки месенджера, який не тільки забезпечує ефективну та зручну комунікацію, але й відповідає високим стандартам безпеки та приватності. Реалізація цих функцій забезпечить користувачам гладке та інтуїтивно зрозуміле використання месенджера у термінальному середовищі.

1.3.2. Нефункціональні вимоги до системи

Нефункціональні вимоги - це вимоги, які визначають якість системи, але не обов'язково її конкретні функції. Вони зосереджені на аспектах, таких як надійність, ефективність, зручність, безпека, та інші параметри якості. Для мого проекту месенджера в терміналі, наступні нефункціональні вимоги можуть бути релевантними:

1. Продуктивність та швидкість: Система повинна забезпечувати високу продуктивність та швидкість обробки даних, включаючи миттєву відправку та отримання повідомлень.
2. Надійність: Месенджер має працювати стабільно та безперебійно, забезпечуючи коректну доставку повідомлень та запрошень в реальному часі.
3. Масштабованість: Система повинна бути спроектована таким чином, щоб легко підтримувати зростання кількості користувачів та збільшення обсягів даних.
4. Безпека: Забезпечення безпеки даних та конфіденційності користувачів є ключовим аспектом, включаючи захист від несанкціонованого доступу та шифрування даних.
5. Сумісність: Месенджер повинен бути сумісний з різними операційними системами та термінальними емуляторами.

6. Інтуїтивно-зрозумілий інтерфейс: Незважаючи на те, що основним середовищем є термінал, інтерфейс користувача повинен бути зрозумілим та простим у використанні.
7. Модульність та розширюваність: Архітектура месенджера має бути модульною, що дозволяє легко розширювати або модифікувати систему за потребою.
8. Оптимізація ресурсів: Програма повинна ефективно використовувати системні ресурси, зокрема пам'ять та процесорний час.
9. Легкість установки та оновлення: Установка та оновлення месенджера мають бути максимально спрощені, щоб користувачі могли легко отримувати останні оновлення та вдосконалення.

Виконання цих нефункціональних вимог забезпечить створення ефективного, надійного та зручного месенджера, який відповідає потребам сучасних користувачів і інтегрується з різноманітними робочими середовищами.

1.4. Сценарій використання системи

У контексті розробки програмного забезпечення, зокрема при створенні месенджера для терміналу, важливим інструментом для візуалізації та аналізу системних вимог є діаграма прецедентів, що є частиною Уніфікованої мови моделювання (UML). Ця діаграма слугує як міст між концептуальним уявленням системи та її фактичною реалізацією, відіграючи критичну роль у процесі розробки програмного забезпечення.

Діаграма прецедентів дозволяє детально моделювати функціональні можливості системи. Це дає можливість бізнес-аналітикам та розробникам чітко визначити, які завдання та функції повинна виконувати система, що є особливо важливим у випадку створення інноваційних рішень, таких як месенджер у терміналі.

Неоціненою є роль діаграми прецедентів у процесі збору та аналізу вимог. Вона допомагає всім зацікавленим сторонам розуміти функціональність системи на більш глибокому рівні, виявляючи потенційні випадки використання та взаємодії з кінцевими користувачами.

Діаграма прецедентів слугує як ефективний засіб спілкування між різними учасниками проекту – від бізнес-аналітиків до розробників та кінцевих користувачів. Вона сприяє кращому розумінню того, як кінцевий продукт буде взаємодіяти з його користувачами та виконувати свої завдання.

Основні Компоненти Діаграми Прецедентів:

- Актори (Actors): Вони представляють зовнішніх учасників, які взаємодіють із системою. У випадку месенджера для терміналу, акторами можуть бути користувачі системи, адміністратори або навіть зовнішні сервіси.
- Прецеденти (Use Cases): Ці елементи діаграми описують конкретні функції системи, як-от створення чату, відправлення повідомлення, управління акаунтом. Вони ілюструють важливі випадки використання, на які система повинна відповісти.

- Відносини (Relationships): Показують, як актори пов'язані з різними прецедентами, відображаючи динаміку взаємодії між користувачами та системою.

PlantUML [2] - це універсальний інструмент, що надає можливість швидкого та зрозумілого створення UML діаграм, включно з діаграмами прецедентів. Він ідеально підходить для оперативного моделювання та візуалізації складних процесів і взаємодій у системах, що є важливим для точного та ефективного проектування програмних рішень. PlantUML дозволяє користувачам розробляти різноманітні діаграми, використовуючи просту та інтуїтивно зрозумілу мову.

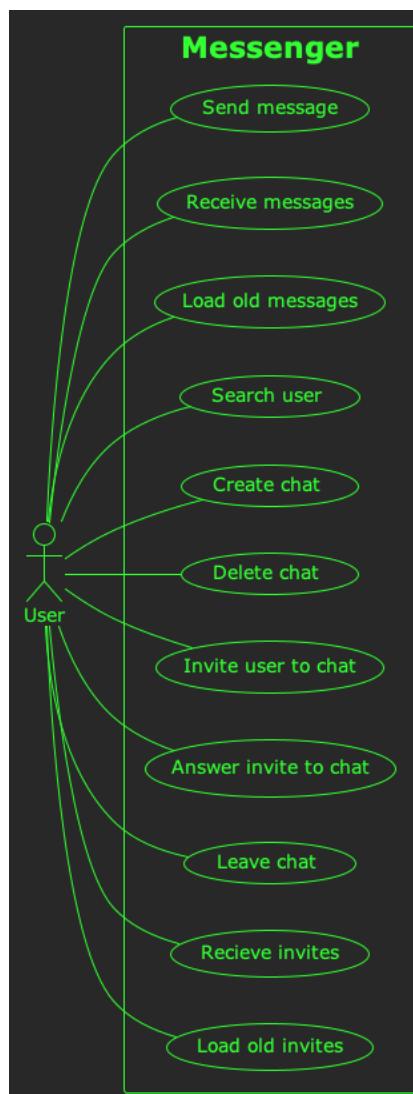


Рисунок 1.1 - Діаграма прецедентів

На діаграмі прецедентів для месенджера представлено актора "Користувач" та його взаємодії з системою через ряд варіантів використання:

1. Надіслати повідомлення (Send message): Користувач може надсилати повідомлення іншим учасникам чату.
2. Отримати повідомлення (Receive messages): Користувач може отримувати повідомлення від інших користувачів.
3. Завантажити старі повідомлення (Load old messages): Користувач може переглядати історію чату та завантажувати повідомлення, які були відправлені раніше.
4. Пошук користувачів (Search user): Користувач має можливість шукати інших користувачів для спілкування або додавання у чат.
5. Створити чат (Create chat): Користувач може створювати новий чат для обміну повідомленнями з іншими.
6. Видалити чат (Delete chat): Користувач має змогу видалити чат.
7. Запросити користувача до чату (Invite user to chat): Користувач може надсилати запрошення іншим користувачам, щоб вони приєдналися до чату.
8. Відповісти на запрошення до чату (Answer invite to chat): Користувач може приймати або відхиляти запрошення до чату від інших користувачів.
9. Покинути чат (Leave chat): Користувач має можливість вийти з чату.
10. Отримувати запрошення (Receive invites): Користувач отримує сповіщення про нові запрошення до чатів.
11. Завантажити старі запрошення (Load old invites): Користувач може переглядати та управляти раніше отриманими запрошеннями.

1.5. Концептуальна модель системи

Концептуальна модель месенджер-системи, яку розглядаємо в рамках цієї курсової роботи, передбачає створення взаємодіючих компонентів, кожен з яких виконує певні функції:

1. Клієнт: Представляє кінцевий точковий пристрій користувача, зазвичай особистий комп'ютер, який встановлює з'єднання з сервером для забезпечення комунікації.
2. Сервер: Основний модуль, що обробляє запити клієнта, керує сеансами та взаємодіями між користувачами, але потребує допомоги в обміні повідомленнями.
3. Воркер: Допоміжний модуль, який в більшості допомагає серверу обробляти запити, де необхідно в реальному часі повідомити клієнта про щось.
4. RabbitMQ [3]: Система брокера повідомень, яка забезпечує асинхронне спілкування між серверами і воркерами, щоб доставляти повідомлення або нотифікації до правильного сервера та з'єднання клієнта до нього.
5. База даних: Використовується для зберігання всієї інформації, включаючи деталі користувачів, історію чатів та інші важливі дані.
6. Auth0 [4]: Система управління ідентифікацією, яка забезпечує безпеку автентифікаційних даних користувачів, їхні логіни та паролі. Також користувачі мають змогу створювати аккаунти за допомогою інших провайдерів персони.
7. Kubernetes [5]: Платформа для оркестрації контейнерів, яка забезпечує високу доступність та масштабованість компонентів системи, включаючи можливість горизонтального масштабування. Також для підвищення надійності системи, дозволяє розгорнути репліки важливих частин, щоб хтось завжди міг обробляти запити

Ця концептуальна модель є основою для детального проектування та реалізації месенджер-системи, орієнтованої на високу доступність, безпеку та ефективність обслуговування користувачів.

1.6. Вибір бази даних

Для потреб месенджер-системи вибрано PostgreSQL [6] — це реляційна база даних, яка ідеально підходить для роботи зі складними зв'язками даних, як-от відношення повідомлень до чату та автора. PostgreSQL відома своєю простотою у використанні, швидкістю та надійністю, що є важливим для систем, що вимагають високої продуктивності та стабільності. Крім того, вона є безкоштовною та відкритою, що забезпечує легкість встановлення та налаштування, а також підтримку великої спільноти розробників.

Переваги PostgreSQL включають:

- Масштабованість та гнучкість: Дозволяє легко масштабувати систему відповідно до зростаючих потреб.
- Транзакційна цілісність: Гарантує надійне управління даними, навіть при складних запитах.
- Розширені можливості запитів: Підтримує складні SQL-запити, що є незамінним для аналізу великих обсягів даних.
- Безпека: Надає розширені можливості для управління доступом та шифрування даних.
- Сумісність: Працює на багатьох операційних системах, що робить її універсальною.

Завдяки цим характеристикам, PostgreSQL є відмінним вибором для систем, де потрібна висока продуктивність в обробці зв'язаних даних, включаючи сценарії роботи з месенджерами.

1.7. Вибір мови програмування та середовища розробки

У виборі мови програмування для проекту месенджера припала перевага на Rust [7], сучасну мову, що заслужила визнання завдяки своїй універсальності та здатності до створення застосунків різноманітного спектру. Rust не має garbage collector, що веде до порівняльної швидкодії з такими мовами як C та C++, тим не менше, вона постачається з революційними засобами забезпечення безпеки даних через її унікальну систему власності.

Rust зарекомендувала себе як мова, здатна надати імпульс до "польоту" проекту з огляду на її швидкість виконання. Це надзвичайно важливо для систем, де час реакції та ефективність є критичними, і наш проект не є винятком. Із Rust наша система буде працювати максимально швидко та ефективно, забезпечуючи користувачам безперебійний доступ до функцій месенджера.

В контексті розробки термінального інтерфейсу, Rust виступає як ідеальний кандидат, оскільки її низькорівневі можливості роблять створення інтерфейсу менш складним завданням, забезпечуючи при цьому потужну продуктивність та надійність.

Хоча Rust може здатися складною для освоєння, вона пропонує багатий досвід, який відкриває широкі можливості для навчання та вдосконалення професійних навичок. Використання Rust є не тільки технічним вибором, але й шляхом до набуття глибоких знань у сфері сучасного програмування.

Зацікавленість великих компаній, таких як Amazon, Microsoft, Discord та Mozilla, у мові програмування Rust зростає, оскільки вони прагнуть переписати критичні частини своїх систем, де необхідні висока швидкість та надійність. Ринковий попит на розробників Rust зростає, а кількість фахівців, які володіють цією мовою, все ще залишається відносно малою, що призводить до вищої заробітної плати для кваліфікованих розробників Rust у порівнянні з іншими популярними мовами програмування.

Rust стала однією з обраних мов програмування, поряд з мовою C, яку дозволено використовувати в розробці ядра Linux. Це визнання підкреслює надійність та ефективність Rust, а також її здатність забезпечувати високий рівень безпеки і

стабільності, необхідних для такої критичної компоненти, як ядро операційної системи.

Як повідомляє видання The Register [8], Microsoft активно працює над переписуванням ключових бібліотек Windows на мові програмування Rust, і код, що є більш безпечним для пам'яті, вже досягає розробників. Девід "dwizzle" Вестон, директор з безпеки операційних систем Windows, оголосив про появу Rust у ядрі операційної системи на конференції BlueHat IL 2023 в Тель-Авіві, Ізраїль, минулого місяця. "Ви насправді побачите завантаження Windows з Rust у ядрі протягом наступних кількох тижнів або місяців, що є дійсно круто," сказав він. "Основна мета тут полягала у перетворенні деяких цих внутрішніх C++ типів даних на їх Rust еквіваленти."

Для підтримки розробки на Rust використовується новітнє середовище розробки RustRover [9] від JetBrains, яке забезпечує розширені інструменти для кодування, відладки та профілювання, дозволяючи розробникам максимально ефективно використовувати потенціал мови.

1.8. Діаграма розгортання системи

Використання Azure Cloud [10] для розгортання системи надає значні переваги. Azure — це гнучка, надійна та масштабована хмарна платформа, яка дозволяє легко управляти ресурсами та службами. Вона забезпечує високу доступність та стійкість до збоїв, а також пропонує широкий спектр інструментів та послуг для оптимізації процесів розгортання та управління.

Використання Kubernetes в Azure надає системі надзвичайну гнучкість і масштабованість. Kubernetes дозволяє автоматично масштабувати систему в залежності від потреб, забезпечуючи високу ефективність роботи без втрат продуктивності. Це особливо важливо для обробки великих навантажень та забезпечення безперебійної роботи системи.

Load Balancer в Kubernetes забезпечує рівномірний розподіл навантаження між нодами та контейнерами, що сприяє підвищенню доступності та ефективності системи. Він гарантує, що жоден одиничний вузол не стане "вузьким місцем", забезпечуючи стабільність і надійність системи.

Використання Helm Charts у Kubernetes спрощує управління розгортаннями та конфігураціями в кластерах. Ці "чарти" дозволяють швидко впроваджувати та оновлювати служби, забезпечуючи консистентність та відтворюваність установки на різних середовищах.

Наявність багатьох нод у Kubernetes забезпечує високий рівень масштабування та надійності. При виході з ладу однієї ноди, інші зможуть продовжити роботу без перебоїв, що є критично важливим для підтримки безперебійної роботи системи.

Використання Docker Images дозволяє стандартизувати та спростити процес розгортання застосунків. Docker забезпечує консистентність середовища на різних етапах розробки та розгортання, знижуючи ризик "воно працює у мене, але не тут".

Можливість налаштування Docker Images за допомогою змінних оточення робить систему надзвичайно гнучкою. Це дозволяє легко змінювати конфігурації та параметри без необхідності зміни коду, сприяючи швидким оновленням та адаптації системи до змінних умов.

Кожен з цих компонентів сприяє створенню потужної, масштабованої та надійної системи, здатної адаптуватися до різних сценаріїв використання та забезпечити високу продуктивність та стабільність роботи.

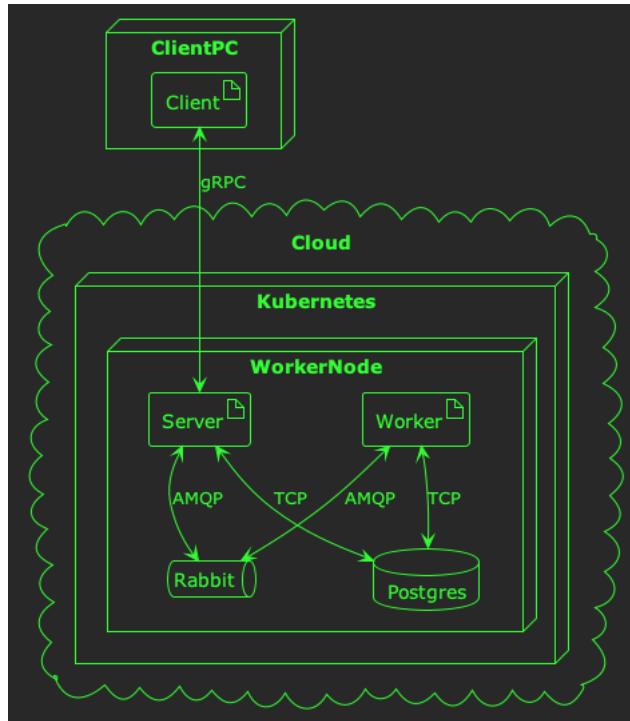


Рисунок 1.2 - Діаграма розгортання

Як видно на цьому рисунку, архітектура системи розгортання включає кілька ключових компонентів. Клієнтська частина (ClientPC) взаємодіє з сервером через gRPC, що є високопродуктивним механізмом віддаленого виклику процедур. В цілому розгортання відбувається у хмарному середовищі, використовуючи Kubernetes, яке забезпечує оркестрацію контейнерів.

В рамках Kubernetes, WorkerNode є віртуальною машиною або фізичним сервером, який виконує контейнери. У ньому розміщено два основних компоненти: Server та Worker, які взаємодіють між собою через RabbitMQ з використанням протоколу AMQP для асинхронного обміну повідомленнями, а також з базою даних PostgreSQL через TCP для збереження стійких даних. RabbitMQ тут виступає як посередник для повідомлень між Server та Worker, а PostgreSQL використовується для довгострокового зберігання даних.

2. РЕАЛІЗАЦІЯ КОМПОНЕНТІВ СИСТЕМИ

2.1. Структура бази даних

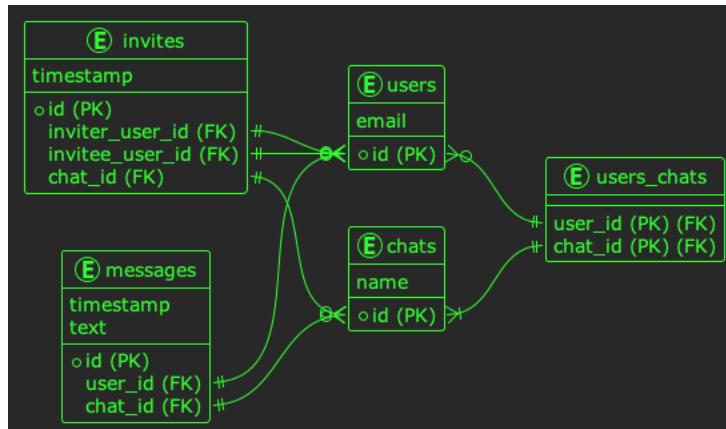


Рисунок 2.1 - Даталогічна модель БД

На наведеному рисунку представлено даталогічну модель бази даних. Даталогічна модель – це абстрактне представлення структури бази даних, яке описує як дані організовані та як вони між собою пов’язані. Вона включає визначення сутностей, їх атрибутів і відносин, а також обмежень на дані. Даталогічна модель дозволяє розробникам та аналітикам глибше розуміти предметну область і є фундаментом для фізичної реалізації бази даних. Давайте розглянемо цю модель детальніше.

Сутність "Користувач" репрезентує енд-юзера системи, обладнаного унікальним ідентифікатором (ID) та електронною адресою (email), причому ID кореспондує з ідентифікатором в системі Auth0. Аутентифікаційна інформація цілеспрямовано відсутня у базі даних, натомість зосереджена в Auth0, що підсилює безпеку та централізує управління обліковими записами.

Чати розглядаються як віртуальні кімнати або "чат-руми", де може спілкуватися багато користувачів. Кожен чат має найменування, що відображає його призначення, та унікальний ідентифікатор, щоб чат-руми могли існувати з різними id.

Кожне повідомлення включає часову мітку (timestamp) відправлення, текст повідомлення, посилання на автора (user_id) та чат (chat_id), куди повідомлення було надіслано. Це забезпечує чіткий контекст та слідування діалогу всередині системи.

Сутність "Запрошення" фіксує моменти надсилання запрошень, включаючи часову мітку, ідентифікатор запрошуочого, ідентифікатор запрошеноого та чат, до

якого відбувається запрошення. Це дозволяє організувати контроль над участю в чатах.

Для реалізації відношення "багато до багатьох" між користувачами та чатами використовується допоміжна таблиця "users_chats". Тут фіксуються належності користувачів до різних чатів та навпаки, це забезпечує гнучкість.

Усі інші відношення в базі даних мають тип "один до багатьох", забезпечуючи, що кожне повідомлення має одного автора, кожне запрошення належить до одного чату та виходить від одного користувача. Ця структура підтримує нормалізований підхід до зберігання даних, ефективно зменшуючи дублювання та сприяючи цілісності даних. Нормалізація також полегшує подальше розширення бази даних та її підтримку, водночас підтримуючи високий рівень продуктивності запитів.

2.2. Архітектура системи

2.2.1. Специфікація системи

Система месенджера проектується як високопродуктивна, розподілена платформа на основі мікросервісної архітектури. Основні компоненти системи будуть включати клієнтські додатки для кінцевих користувачів, сервери для обробки запитів та логіки додатку, а також зовнішні сервіси для аутентифікації користувачів та зберігання даних.

Система підтримуватиме gRPC для взаємодії між клієнтами та сервером, а також AMQP для ефективного обміну повідомленнями між серверними компонентами.

Функціональність месенджера охоплюватиме створення, редагування та управління чат-румами, а також можливість відправлення та отримання повідомлень в реальному часі. Система також підтримує створення запрошень.

Система була реалізована з допомогою Auth0, для зберігання логін інформації користувача та надавання ідентифікатора користувачеві. Це дозволить нашому застосунку аутентифікувати користувачів без їх паролів, що підвищує безпеку.

Завдяки використанню Kubernetes та контейнерізації через Docker, система зможе автоматично масштабуватися для підтримки зростаючого числа користувачів та навантаження. Це також забезпечує високий рівень доступності сервісу та мінімізацію простою.

Середовище розробки включатиме сучасні інструменти для забезпечення швидкого розгортання, тестування та впровадження змін. Використання CI/CD пайплайнів сприятиме неперервній інтеграції та доставці продукту, забезпечуючи постійну готовність системи до роботи.

Специфікація стане фундаментом для розробки, вказуючи на ключові аспекти, що включають технічні деталі, архітектурні рішення та бізнес-логіку, необхідну для забезпечення повноцінної та ефективної роботи месенджер-системи.

2.2.2. Діаграма послідовностей

Діаграма послідовностей — це тип діаграми в UML, яка показує, як об'єкти взаємодіють один з одним та в якій послідовності це відбувається протягом певного сценарію або бізнес-процесу. Вона візуально представляє послідовність повідомлень між об'єктами, що дає можливість зrozуміти потік даних та контроль за процесами в системі. Ця діаграма корисна для представлення динаміки взаємодії в системі, особливо коли потрібно відслідкувати зміни станів об'єктів або визначити, як система реагує на різні події.

2.2.2.1. Проста клієнт-серверна взаємодія

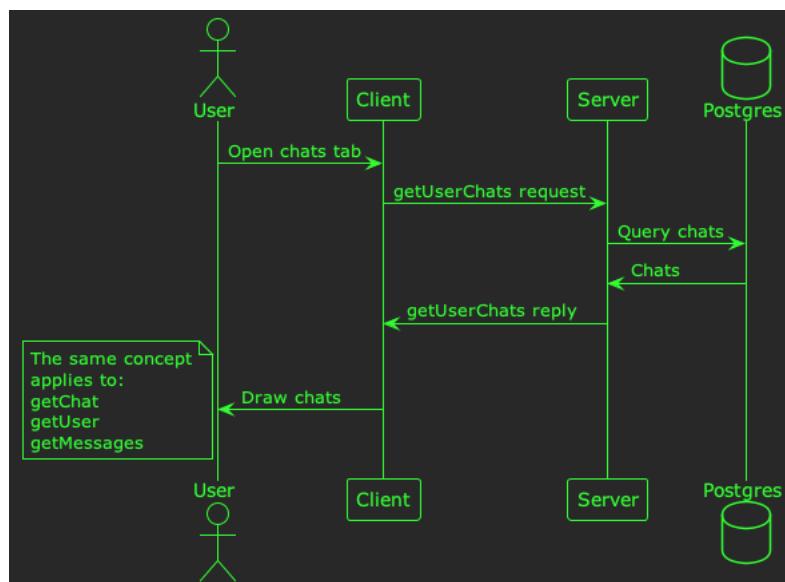


Рисунок 2.2 - Проста клієнт-серверна взаємодія

На цьому рисунку зображене приклад простої взаємодії між клієнтом та сервером. Розгляньмо його детальніше.

Спершу користувач ініціює процес, відкриваючи вкладку чатів у клієнтському застосунку. Це дія, яка, з технічної точки зору, активує здійснення запиту до сервера. Запит `getUserChats` формується клієнтською частиною за допомогою визначеного API і відправляється на сервер через зашифроване з'єднання, що забезпечує інтегритет та конфіденційність переданих даних.

По прибуттю на сервер, запит обробляється відповідним серверним програмним забезпеченням. Тут сервер виконує SQL-запит до бази даних Postgres, яка є дуже ефективною для роботи з транзакціями та запитами, пов'язаними з великими обсягами даних. Серверні оптимізації та індексація у базі даних дозволяють швидко знайти та повернути необхідний список чатів.

Отримавши список чатів від Postgres, сервер перетворює ці дані в формат, придатний для передачі через мережу, і відсилає відповідь `getUserChats reply` назад до клієнта. Цей крок демонструє важливість ефективного мережевого кодування даних для забезпечення швидкості та надійності передачі.

Завершальним етапом є відображення списку чатів у клієнтському застосунку. Клієнтська частина застосунку обробляє отримані від сервера дані та оновлює інтерфейс користувача, забезпечуючи актуальне та зрозуміле відображення інформації.

Особливості цього процесу включають безпосередній запит до бази даних, відсутність RabbitMQ і зосередженість на простоті та ефективності. Це свідчить про лаконічну та швидку систему, що не потребує додаткових шарів абстракції чи компонентів для обробки простих запитів.

Такий же принцип може бути застосований до інших типів запитів, таких як `getChat`, `getUser`, `getMessages`, що ілюструє масштабованість та універсальність підходу, придатного для широкого спектру задач взаємодії з базою даних. Основна ідея полягає в тому, що прості запити, які не вимагають складної логіки обробки або високої конкурентності, можуть бути виконані ефективніше без використання додаткових систем черг або посередників.

2.2.2.2. Відправлення запрошення

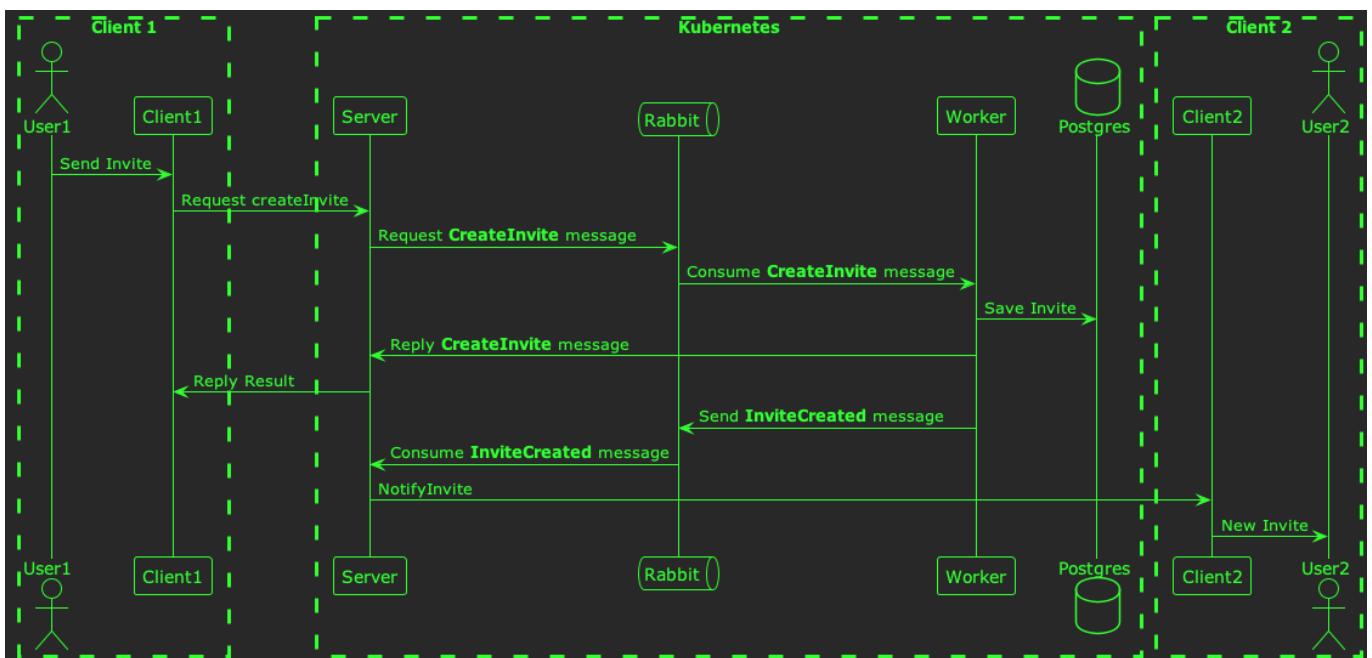


Рисунок 2.3 - Відправлення запрошення

На цьому рисунку зображено більш складний варіант використання. Оскільки після відправлення запрошення, потрібно повідомити клієнта, що його запросили. Тому потрібно щоб відповідне з'єднання клієнта з сервером отримало повідомлення.

Початковий крок процесу ініціюється користувачем 1, який через свій клієнтський застосунок відправляє запрошення. Клієнтський застосунок формує запит `createInvite` та надсилає його на сервер. Такий запит не просто передає інформацію, але й запускає каскад обробок на серверній стороні.

Сервер, отримавши запит на створення запрошення, відправляє повідомлення `CreateInvite` в RabbitMQ, що виступає як посередник для асинхронної обробки повідомень. RabbitMQ дозволяє відокремити процес створення запрошення від основного потоку обробки запитів, тим самим знижуючи затримку відповіді клієнту та забезпечуючи масштабованість системи.

Воркер, який працює в середовищі Kubernetes, споживає повідомлення `CreateInvite` з черги та виконує логіку збереження запрошення в базі даних Postgres.

Це гарантує, що важливі дані не втрачаються та зберігаються для подальшого використання.

Після успішного збереження запрошення, воркер відправляє підтвердження назад на сервер через RabbitMQ. Сервер, отримавши це підтвердження, інформує клієнтський застосунок користувача 1 про успішне створення запрошення, забезпечуючи таким чином зворотний зв'язок та відповідь на його дію.

Окрім цього, воркер також відправляє повідомлення `InviteCreated` в RabbitMQ, щоб повідомити інші компоненти системи про нове запрошення. Якщо користувач 2 в даний момент підключений до системи, його клієнтський застосунок, через сервер, отримує повідомлення про нове запрошення. Це дозволяє забезпечити миттєве сповіщення та взаємодію між користувачами в реальному часі. Також це реалізує патерн Request-Reply, про нього далі.

Клієнтський застосунок користувача 2 відображає запрошення, тим самим завершуючи цикл взаємодії. Цей процес підкреслює важливість асинхронної обробки для високопродуктивних систем, здатних швидко реагувати на дії користувачів та забезпечувати високу доступність та надійність зберігання даних.

Таким чином, весь процес відправлення запрошення є виразним прикладом складної мультикомпонентної взаємодії, яка оптимізована для забезпечення ефективності та надійності в розподілених системах, таких як месенджери сучасності.

2.2.2.3. Прийняття запрошення

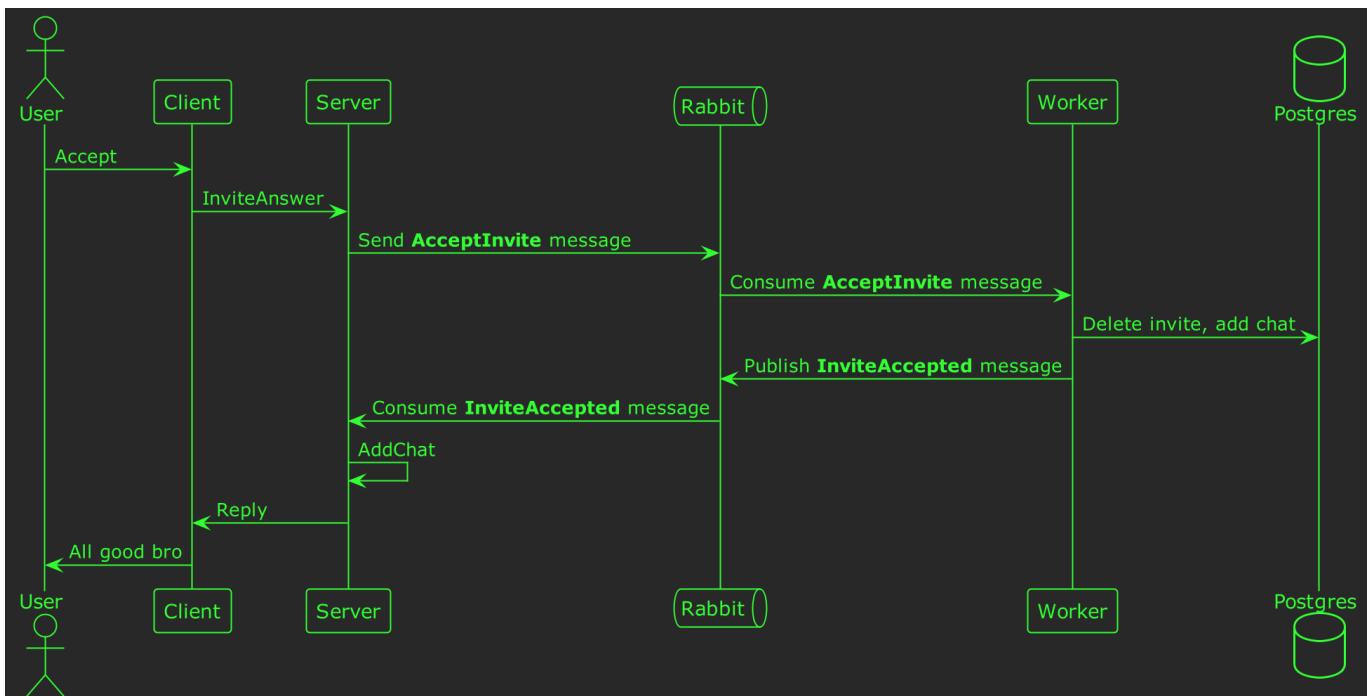


Рисунок 2.4 - Прийняття запрошення

Процес прийняття запрошення в месенджері є інтерактивним моментом, що включає взаємодію між користувачем, клієнтом, сервером, RabbitMQ, воркером і базою даних Postgres. Він відображає, як користувач, отримавши запрошення до чату, може прийняти його та відразу ж вступити у новостворений чатовий діалог.

Спочатку користувач вирішує прийняти запрошення, відправлене іншим користувачем, і через свій клієнтський застосунок ініціює дію прийняття.

Клієнтський застосунок генерує відповідь на запрошення, зазвичай це буде повідомлення з командою або маркером, що вказує на прийняття (наприклад, `InviteAnswer`), і відправляє це на сервер.

Сервер обробляє отриману відповідь та направляє повідомлення `AcceptInvite` в систему черг RabbitMQ, де це повідомлення стає задачею для асинхронної обробки.

Воркер, який моніторить чергу RabbitMQ, приймає це повідомлення, виконує запит на видалення запрошення з бази даних Postgres та створює запис про новий чат.

Після успішного виконання цих операцій, воркер публікує нове повідомлення `InviteAccepted` назад у чергу RabbitMQ, щоб повідомити інші компоненти системи про зміну статусу запрошення.

Сервер слухає чергу повідомень RabbitMQ і, отримавши `InviteAccepted`, динамічно додає нову чергу для чату в потік повідомлень клієнта.

Клієнтський застосунок отримує підтвердження від сервера про успішне приєднання до чату.

Клієнтський застосунок тепер оновлює свій інтерфейс, завантажує історію повідомлень з нового чату та відображає їх користувачу, одночасно видаляючи запрошення зі списку активних.

На останок, користувач отримує візуальне підтвердження в своєму клієнті про те, що він був успішно доданий до чату, що забезпечує плавне та зручне користувацьке досвід.

Цей процес висвітлює основні особливості таких систем: здатність динамічно змінювати потік повідомлень у клієнтському застосунку на основі подій в системі, асинхронну обробку для ефективності та швидкості, та надійність процесів, що забезпечують безперебійне приєднання до чату. Використання RabbitMQ в даному контексті дозволяє ефективно розподіляти завдання та оптимізувати реакцію системи на дії користувача.

2.2.2.4. Відправлення повідомлення

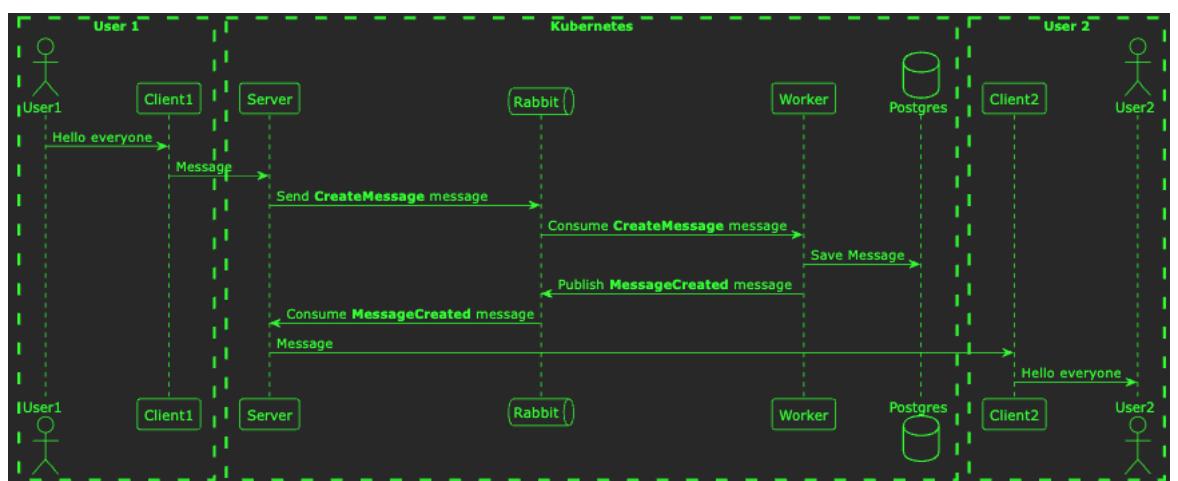


Рисунок 2.5 - Відправка повідомлення

Процес відправлення повідомлення в месенджері є основною частиною будь-якої системи спілкування, що забезпечує обмін інформацією між користувачами. На прикладі діаграми послідовностей можна прослідкувати кроки, які виконуються в системі при відправленні повідомлення від одного користувача до іншого.

Користувач 1 (User1) розпочинає процес, написавши повідомлення в своєму клієнтському застосунку (Client1) та натиснувши кнопку відправлення.

Повідомлення відправляється через мережу на сервер, де воно стає частиною потоку даних, що обробляється сервером.

Сервер, отримавши повідомлення, генерує команду CreateMessage, яка відправляється до системи черг повідомлень RabbitMQ. Це покрокове віddілення відправлення від обробки повідомлення дозволяє підвищити ефективність системи.

Воркер, який працює в середовищі Kubernetes та спостерігає за чергами RabbitMQ, приймає команду CreateMessage та зберігає повідомлення в базу даних Postgres. Це забезпечує довгострокове збереження повідомлення та можливість його відновлення чи перегляду в майбутньому.

Після збереження повідомлення, воркер публікує нове повідомлення MessageCreated в спеціалізований ексчендж у RabbitMQ, що є сигналом для системи про те, що повідомлення було успішно створено та готове до розповсюдження серед учасників чату.

Сервер моніторить повідомлення від RabbitMQ і, отримавши MessageCreated, негайно відправляє це повідомлення через стрім до всіх підключених клієнтів, таких як Client2, що забезпечує миттєве сповіщення.

Клієнт 2 (на стороні користувача 2) отримує повідомлення "Hello everyone" та відображає його, дозволяючи користувачу 2 побачити та відреагувати на повідомлення в реальному часі.

Ця послідовність дій відображає важливість асинхронної обробки в сучасних системах месенджерів, здатність зберігати стан та важливі дані, а також спроможність системи до миттєвого сповіщення учасників чату. Такий підхід забезпечує швидке, надійне та ефективне спілкування, яке є ключовим для користувацького досвіду в цифровому спілкуванні.

2.2.3. Паттерн Redux

Redux є популярним патерном управління станом для JavaScript-додатків, який часто використовується з бібліотеками та фреймворками, такими як React, Angular, і інші. Цей патерн забезпечує єдине джерело правди для стану додатку, що спрощує управління станом у складних додатках. Це, звичайно не дуже схоже на наш проект, але я вважаю цей патерн найкращим для розробки single-page-application.

Мій клієнт по суті дуже схожий на web spa, єдина різниця що замість браузера та тайпскрипта в мене термінал та Раст. Розгляньмо оригінальну версію патерна, а потім те як я адаптував її для моого проекту.

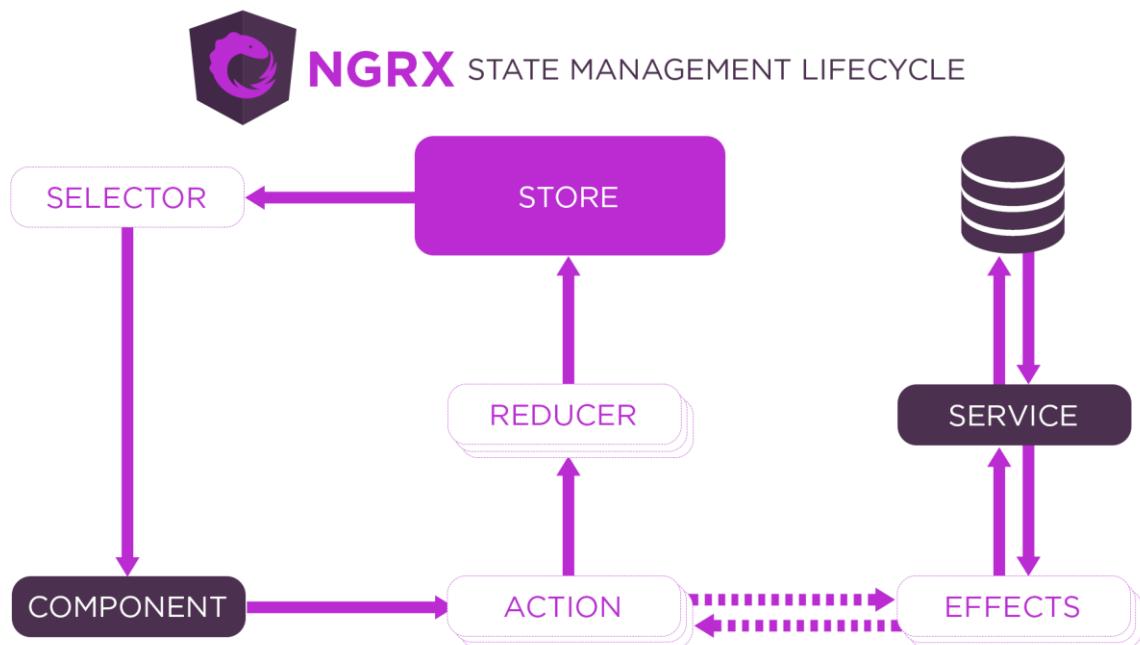


Рисунок 2.6 - Життєвий цикл NGRX

На цьому зображені – життєвий цикл управління станом звичайного single-page-application на Angular [11] за допомогою фреймворку NGRX [12]

Основні елементи життєвого циклу NgRx/Redux включають:

1. Store: Це централізоване сховище для всього стану додатку. Воно є не змінним (immutable), тобто стан не може бути змінений напряму, а замість цього має бути створено нову копію стану з будь-якими оновленнями.

2. Action: Коли в додатку відбувається подія, яка потребує зміни стану (наприклад, користувач клікає кнопку), компонент відправляє дію. Дія є об'єктом, який описує, що потрібно зробити, і може містити додаткові дані, необхідні для зміни стану.
3. Reducer: Це чиста функція, яка приймає попередній стан додатку та дію, що потрібно виконати, і повертає новий стан. Reducer визначає, як саме дія трансформує стан.
4. Effects: В NgRx, ефекти використовуються для обробки побічних ефектів, таких як асинхронні запити до API. Ефекти слухають дії, виконують асинхронні операції та повертають нові дії, які потім знову обробляються редьюсерами.
5. Selector: Це функції, що дозволяють обрати та поєднати дані зі стору. Вони допомагають ізолювати частини стану, які необхідні конкретним компонентам.
6. Component: В Angular, компоненти використовують селектори для отримання необхідних частин стану зі стору. Вони також відправляють дії до стору для ініціації змін стану.
7. Service: У контексті NgRx, сервіси часто використовуються для взаємодії з сервером через HTTP-запити. Вони можуть бути інтегровані з ефектами для виконання асинхронної логіки.

Разом, ці елементи створюють потужний цикл управління станом, який дозволяє розробникам легко відслідковувати, змінювати та відтворювати стан додатку, що особливо важливо в складних додатках з різноманітними взаємодіями та даними.

У своєму проекті я адаптував патерн Redux, враховуючи специфіку асинхронної роботи та вимоги до многопотоковості, які притаманні мові програмування Rust. Я вініс суттєві зміни у традиційну архітектуру Redux, які відображають унікальні властивості та обмеження моєї системи.

Перш за все, store у моєму додатку діє як центральний менеджер стану, але відрізняється тим, що стан є невеликим з можливістю швидкого клонування. Для оптимізації обробки великих обсягів даних я використовую Mutex, який дозволяє

безпечно управляти доступом до великих даних у багатопотоковому середовищі, зберігаючи їх за посиланням, а не копіюючи при кожній зміні.

Друга значна адаптація полягає у використанні Channel замість селекторів. Взявши на озброєння підхід з мови Go [13], я використовую channel для розсылки повідомлень між різними частинами системи. Коли стан змінюється, store автоматично розсилає нову версію стану через channel, що дозволяє іншим частинам додатку відгукнутися на ці зміни.

Найскладнішим виявилося реорганізувати reducer. В моєму рішенні, вони не є чистими функціями, оскільки мої дії (actions) можуть мати різне значення в залежності від контексту. Наприклад, дія «Натиснуто J» може означати введення літери J, прокручування екрану вниз чи іншу дію. Тому я створив ієархію reducer-ів, де один AppReducer керує викликом інших reducer-ів в залежності від пріоритету. Це дозволяє моїм reducer-ам вирішувати, як реагувати на дії в залежності від поточного стану додатку.

Reducer-и повертають enum, який вказує на результат обробки: новий стан, ініціація асинхронної задачі або ігнорування повідомлення. Це дозволяє послідовно передавати обробку від одного reducer-а до іншого, поки не буде знайдено відповідний обробник дії чи осягнуто конкретного зміненого стану.

Також для зображення стану застосунку користувачеві, я створив ієархію view структур, які вміють малювати поточний стан застосунку. Після кожного action-у або секунду, view буде перемальовувати поточний стан, щоб в клієнта оновлювався контент, це ніби 1 кадр в секунду, або як реакція на будь-який action.

Таким чином, я інтегрував патерн Redux у свій додаток, адаптувавши його до потреб і властивостей асинхронної обробки та багатопоточності Rust, забезпечуючи високу продуктивність та надійність управління станом.

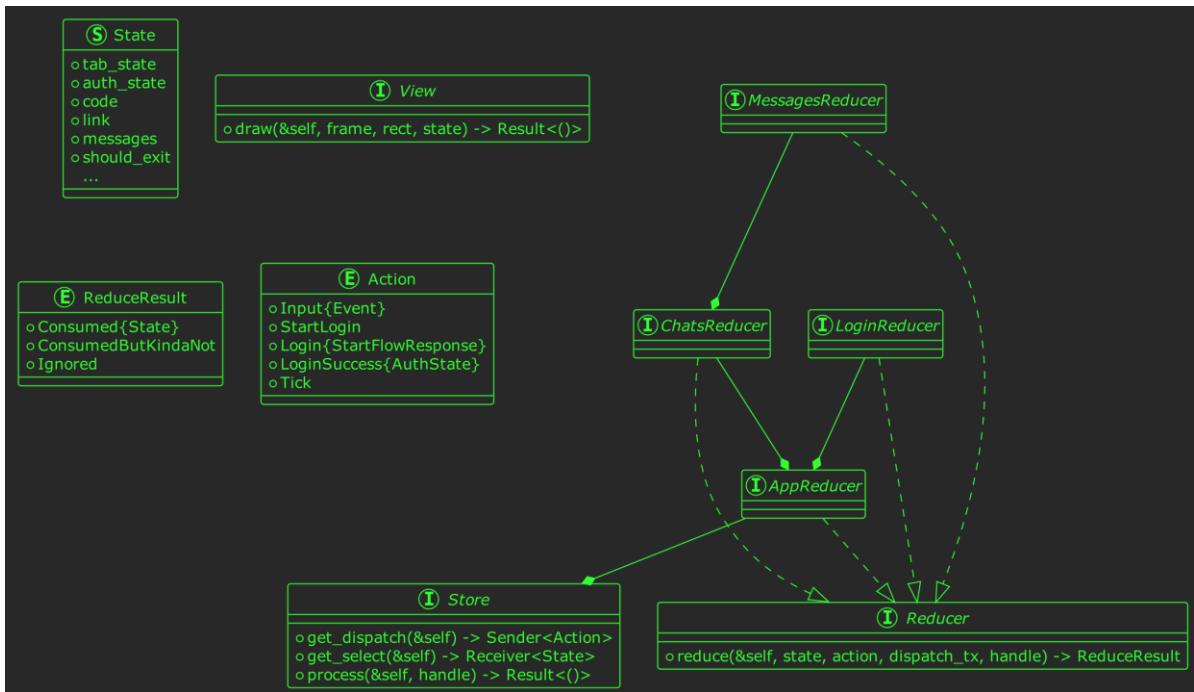


Рисунок 2.7 - Діаграма класів патерну Redux

На цьому рисунку зображена спрощена діаграма класів всього, що відноситься до патерну Redux. Замість трейтів та структур, що їх реалізують, я просто залишив трейти. Трейт це ніби інтерфейс в Расти. Розгляньмо її детальніше.

Є структура **State** – стан нашого застосунку, в якому зберігається вся інформація відома клієнту. Це може бути яка таба відкрита, які повідомлення є, які чати є, завантажена інформація про користувачів, та багато іншого. Все завдання клієнту – оновлювати цей стан відповідно до дій користувача.

Для зображення цього стану використовується ієархія **View**, але вони не сильно відносяться до самого Redux, тому я зобразив тільки їх базовий трейт. Щоб намалювати інтерфейс користувачу, необхідно викликати метод `draw`, та передати поточний стан. Далі він намалює всі необхідні компоненти

І тепер більш складна частина, **Store**, так званий менеджер поточного стану та серце всього патерну. В нього як залежність вставлений **AppReducer**, та вся сила **Store**, це метод `process`. Цей метод повинен бути асинхронно стартований в окремому потоці, та він буде слухати калан з `dispatch`, надавати отриманий `action` до **AppReducer**, та зберігати в тому методі цей новий стан, і відправляти його до `select`. Також в нього є 2 гетери, для того щоб відправляти нові `action` до `store` та щоб отримувати `state`. Це

дозволяє ніби викликати метод для обробки просто надіславши повідомлення, та інша частина програми отримає результат обробки.

Кожен Reducer імплементує трейт... Reducer, тому я не повторював методи. Важлива відмінність в тому, що для виклику методу reduce, треба передати канал для новий action, так Reducer-и, які ніби належать до Store, можуть використовувати його для обробки. Одне з таких використань – після натискання кнопки Enter, для відправки повідомлення Reducer може очистити поле для повідомлення, і надіслати новий action SendMessage, який буде оброблятись вже іншим Reducer-ом.

Також важливо зазначити, що до Reducer-ів також надається handle до tokio runtime. Бібліотека tokio [14], менеджер потоків для ефективної асинхронної обробки. Але оскільки process метод повинен працювати постійно та асинхронно, щоб не забирати в tokio робочих процесів, він запущений в окремому, тоді для запуску асинхронних тасок в tokio, потрібно мати до нього handle. Один з прикладів використання цього, це звернення до сервера. Якщо у відповідь на action, потрібно зробити запит до сервера, наприклад завантажити чати, reducer запускає асинхронну таску, яка зробить цей запит, почекає доки він завершиться, та запакує відповідь у новий Action та відправить його до dispatch каналу. Ця таска потім запускається в tokio та виконує потребу клієнта.

Отже, використання цього адаптованого патерну, дозволяє отримати ефективне використання ресурсів клієнтського комп’ютера, та максимально розділити код на маленьких дрібні частини, які відповідають тільки за своє завдання. Тому додавати новий функціонал буде надзвичайно легко, та ця структура буде підтримувати збільшення функціоналу. Великі дані не перекопіюються, всі правила безпеки даних Rust-а виконані, та це було дуже весело проектувати.

2.2.4. Паттерн Prototype

Паттерн Prototype - це креаційний паттерн проектування, який використовується для створення об'єктів шляхом копіювання існуючого об'єкта, замість створення нового об'єкта з нуля і ініціалізації його зі стандартними значеннями. В основі цього патерну лежить концепція прототипу - об'єкта, який може створювати копії самого себе.

Суть патерну полягає в тому, що ви маєте об'єкт, який конфігурований певним чином і якщо вам потрібен інший об'єкт, схожий на оригінал, ви просто клонуєте прототип замість того, щоб створювати новий об'єкт і конфігурувати його відповідно до тих же параметрів. Це дозволяє уникнути повторного створення коду і може бути більш ефективним з точки зору продуктивності, особливо якщо створення нового екземпляра є ресурсомісткою операцією.

У контексті мови програмування Rust, патерн Prototype реалізується за допомогою трейтів Copy, Clone та Drop.

- Трейт Copy: це спеціальний трейт, який може бути реалізований для типів об'єктів, які можуть бути безпечно копійовані біт-в-біт, тобто вони не управлюють ресурсами, які вимагають спеціального оброблення при копіюванні (наприклад, вказівники). Типи, що імплементують трейт Copy, можуть автоматично копіюватись у Rust, коли вони передаються або повертаються з функцій.
- Трейт Clone: використовується для створення глибоких копій об'єктів. Якщо тип управлює деякими ресурсами за межами самого об'єкта, наприклад, динамічною пам'яттю, файлами або мережевими з'єднаннями, для клонування таких типів потрібно визначити Clone для того, щоб вказати, як саме ресурси повинні бути скопійовані чи клоновані.
- Трейт Drop: використовується для очищення ресурсів, коли об'єкт виходить з області видимості. Це може бути використано у контексті патерну Prototype для знищення прототипу, якщо це необхідно, перед створенням його копії.

Таким чином, у Rust трейт Clone безпосередньо відповідає патерну Prototype, оскільки він використовується для створення нових об'єктів за допомогою клонування існуючих. Трейт Copy є спеціальним випадком, коли клонування можливе без додаткових витрат, тоді як трейт Drop гарантує належне управління ресурсами при

видаленні об'єктів. Усі ці трейти разом дозволяють Rust розробникам використовувати патерн Prototype ефективно, забезпечуючи контроль над життєвим циклом об'єктів і їх клонуванням.

У Rust, `derive` є макросом, що дозволяє автоматично імплементувати певні трейти для будь-якої структури чи перечислення (`enum`). Це значно спрощує роботу з типовими трейтами, такими як `Copy`, `Clone`, `Debug`, `Default`, `Eq`, `PartialEq`, `Hash`, і `Ord`, оскільки вам не потрібно вручну писати код для стандартної імплементації цих трейтів.

Коли ви додаєте атрибут `#[derive(TraitName)]` до вашої структури чи перечислення, компілятор Rust автоматично генерує код, який імплементує вказаний трейт для вашого типу. Наприклад, якщо ви хочете, щоб ваша структура могла бути клонована (має можливість створення глибоких копій), ви можете просто додати `#[derive(Clone)]` до оголошення структури, і метод `clone()` буде доступний для екземплярів цього типу.

В контексті патерну Prototype, `derive` спрощує процес клонування об'єктів, оскільки вам не потрібно турбуватися про написання специфічного для клонування коду - ви можете використовувати автоматично генерований метод `clone()`. Якщо ваш тип також є простим і не містить ресурсів, що потребують спеціального керування при копіюванні (наприклад, не містить `Box`, `Vec`, `String` тощо), ви можете використовувати `#[derive(Copy, Clone)]`, щоб зробити ваш тип імплементувати трейт `Copy`, який дозволяє ефективніші копії без виклику методу `clone()`.

Трейт `Drop` не можна імплементувати за допомогою `derive`, оскільки деструктори часто вимагають специфічної логіки для очищення ресурсів, і цю логіку потрібно писати вручну у методі `drop(&mut self)`.

Ось приклад, як можна використати `derive` для імплементації трейтів `Clone` і `Copy`:

```
#[derive(Clone, Copy)]  
struct Point {  
    x: i32,  
    y: i32,  
}
```

Рисунок 2.8 - Використання макросу для Clone та Copy

У цьому прикладі Point тепер можна безпечно копіювати біт-в-біт, і клонувати за допомогою методу `clone()`, що задовольняє вимоги патерну Prototype в контексті Rust.

Ось приклад використання тільки трейту `Clone`, оскільки в структурі існують поля, які небезпечно копіювати біт-в-біт, а саме `Arc`, або `Atomic Reference Counter`, який дозволяє організувати доступ до одних даних з багатьох потоків, при цьому автоматично видаливши ці данні коли на них не буде посилань.

```
#[derive(Default, Clone)]  
pub struct State {  
    pub tab_state: TabState,  
    pub auth_state: Option<AuthState>,  
    pub code: Option<String>,  
    pub link: Option<String>,  
    pub messages: Arc<RwLock<Vec<String>>>,  
    pub should_exit: bool,  
}
```

Рисунок 2.9 - Використання тільки трейту `Clone`

Оскільки в Rust цей паттерн буквально підтримується мовою, тому його реалізовувати та використовувати в застосунках дуже просто, та він значно покращує процес створення клонів певної структури.

2.2.5. Паттерн Facade

Паттерн Фасад (Facade Pattern) - це структурний паттерн проектування, який надає єдиний уніфікований інтерфейс до набору інтерфейсів у підсистемі. Фасад визначає вищий рівень інтерфейсу, який робить підсистему легшою у використанні.

Фасад зазвичай використовується для:

- забезпечення простого інтерфейсу до складної підсистеми;
- декомпозиції системи на декілька підсистем, зосереджуючи звернення до цих підсистем через один об'єкт, що веде до меншої складності та залежностей між системами;
- ізоляції клієнтів від компонентів підсистеми, що сприяє легшій модифікації та розвитку підсистем.

Розгляньмо як він реалізований на серверній стороні. Як вже було зазначено, сервер використовує gRPC [15] протокол для ефективної комунікації клієнта з сервером, який працює набагато швидше звичайного REST API, та підтримує стрімінг. Одна зі складових використання gRPC - .proto файл специфікації протоколу. Там зазначені всі сервіси, методи та повідомлення які передаються.

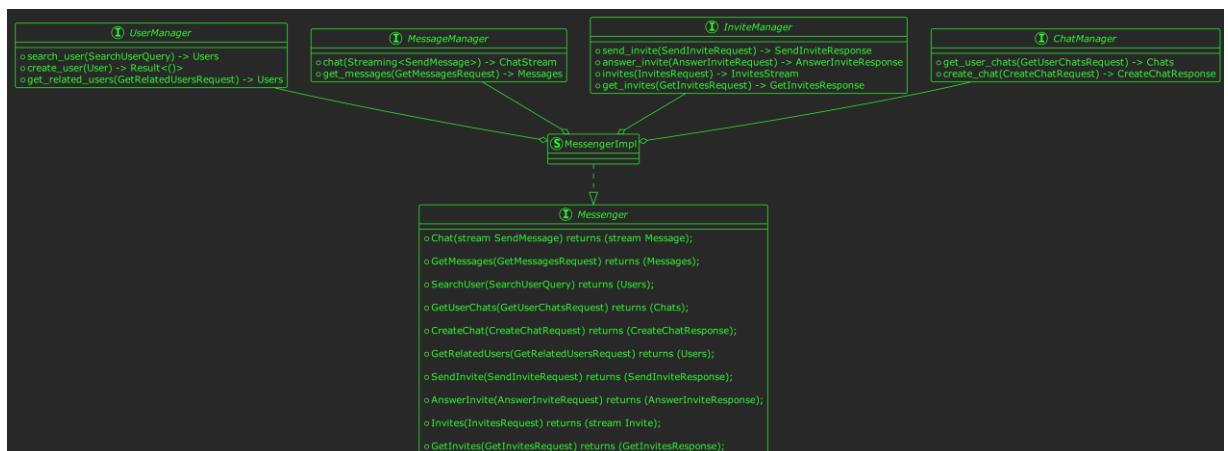


Рисунок 2.10 - Діаграма класів для реалізації Фасад

Як видно на цьому рисунку, трейт Messenger – автоматично згенерована адаптація для прото файлу в Расти. Я не змінював стиль дикларації Віддаленого Виклику Процедур, щоб було простіше його розуміти. Але реалізовувати всі ці методи

в одному місці – величезне порушення першого принципу SOLID [16], Single Responsibility Principle. Для розбиття цієї логіки на менші, більш прості частини використано паттерн Фасад.

Структура яка реалізовує цей трейт, має багато залежностей, в яких і знаходиться логіка обробки кожного запиту. Структура MessengerImpl просто викликає відповідні методи і все.

Реалізація Фасаду в цьому випадку сприяє не тільки легшому розумінню та використанню системи, але й забезпечує гнучкість для майбутньої модифікації та розширення функціональності сервера. Це дозволяє розробникам зосередитись на конкретних завданнях в рамках кожної залежності, підтримуючи високу читабельність коду та спрощуючи процес тестування та налагодження.

Таким чином, використання паттерну Фасад у розробці серверної частини на Rust з використанням gRPC є ефективним рішенням, що відповідає сучасним вимогам до архітектури розподілених систем.

2.2.6. Паттерн Адаптор

Під час написання в мене з'явилася проблема, я хотів використати звичайний паттерн ООП, з принципом солід, передати нащадка замість конкретного типу. У Rust, коли ви маєте трейт B, який є супертрейтом для іншого трейта A, це означає, що для реалізації трейта A необхідно також імплементувати функціонал трейта B. Можливо, виникне питання, чому об'єкт трейта A не можна використовувати там, де очікується трейт B. Це пов'язано зі строгостю системи типів та трейтів у Rust, яка вимагає, щоб кожен тип чи об'єкт трейта явно реалізовував очікуваний у функції трейт.

Система типів Rust не дозволяє автоматично "піднімати" об'єкт похідного трейта до об'єкта трейта-супертрейта. Це через те, що об'єкти трейтів представлені у вигляді "товстих" вказівників (покажчик на об'єкт плюс покажчик на таблицю віртуальних методів, або vtable), і структура vtable супертрейта може відрізнятися від vtable похідного трейта.

Проте в Rust існує експериментальна функція, звана "trait upcasting", яка дозволила б передавати об'єкт трейта похідного трейта там, де очікується об'єкт трейта-супертрейта. Ця функція все ще знаходиться в розробці і не є стабільною. Вона

передбачає дозвіл на "підняття" об'єктів трейтів до об'єктів їхніх супертрейтів, що може бути корисним у ситуаціях, де потрібен поліморфізм.

Я вирішив цю проблему використавши щось типу патерну Adapter. Де я створив нову структуру, яка має всередині дочірній трейт, а назовні імплементує батьківський. Всередині, він просто викликає методи дочірнього трейту та це дозволяє расту розуміти що від нього хочуть навіть не маючи успадкування.

Патерн адаптер (Adapter Pattern) використовується для перетворення інтерфейсу одного класу у форму, яку може використовувати інший клас. Це дозволяє класам з несумісними інтерфейсами працювати разом, що є корисним у ситуаціях, де зміна інтерфейсу класу не є можливою або практичною.

```
4 usages  ↲ Michael Molchanov
127 ④  pub struct MessengerAdapter {
128      messenger: Arc<
129          dyn CrabMessenger<ChatStream = ChatResponseStream, InvitesStream = InviteResponseStream>,
130          >,
131      }
132
133      ↲ Michael Molchanov
134      impl MessengerAdapter {
135          ↲ Michael Molchanov
136          pub fn new(
137              messenger: Arc<
138                  dyn CrabMessenger<
139                      ChatStream = ChatResponseStream,
140                      InvitesStream = InviteResponseStream,
141                      >,
142              >) -> Self { Self { messenger } }
143
144      } ⓘ
145
146      ↲ Michael Molchanov
147      #[async_trait]
148      ④  impl Messenger for MessengerAdapter {
149          ↲ Michael Molchanov
150          type ChatStream = ChatResponseStream;
151          ↲ Michael Molchanov
152          async fn chat(
153              &self,
154              request: Request<Streaming<SendMessage>>,
155          ) -> Result<Response<Self::ChatStream>, Status> { self.messenger.chat(request).await }
```

Рисунок 2.11 - Патерн Адаптер

На цьому рисунку видно імплементацію патерну Адаптер. Мій CrabMessenger – це трейт, який імплементує трейт від gRPC, але я його інджектую як тільки трейт CrabMessenger, і я не можу його використовувати в місцях для Messenger, тому я створив MessengerAdapter, який просто повторює виклики покладеного до нього CrabMessenger-а, і я його створюю руками а не інджектую, тому отримавши CrabMessenger я роблю з ним Adapter, який можу вже використовувати в місці для Messenger.

2.2.7. Паттерн Singleton

Дуже часто зустрічається патерн singleton, бо якщо писати чистий ООП код на чисто інтерфейсах, будуть тільки класи з даними та тільки класи з кодом. Ці класи з кодом не мають стану та не зберігають даних всередині себе, тільки залежності, тому можна створити один такий екземпляр та використовувати його одного в усіх місцях де треба. Він тут буде відігравати роль ніби просто функції яку можна викликати, але з залежностями які можна замінити.

Його часто називають анти-паттерном, оскільки потрібно робити статичну змінну та на неї всім посилятись, що дуже сильно зв'язує код, але якщо додати dependency injection, можна отримати розв'язаний код і одинаків, тобто найкраще з обох варіантів.

В моєму проекті використовується compile-time dependency injection, бо я не хотів платити швикістю програми під час виконання. За допомогою крейту shaku [17], який надає прості макроси для dependency injection, я створив розв'язаний код який побудований на основі dependency injection, але оскільки Rust пишеться трохи в іншому стилі, вийшло не максимально красиво.

2.2.8. Паттерн Request-Reply

Патерн "Запит-Відповідь" у контексті RabbitMQ передбачає взаємодію двох учасників: запитувача, який відправляє запитове повідомлення і чекає на відповідь, та відповідача, який отримує запитове повідомлення і відповідає на нього. Ця взаємодія здійснюється через два канали: канал запиту та канал відповіді. Канал запиту може бути або каналом точка-до-точки, або каналом публікації-підписки, залежно від того,

чи повинен запит бути переданий одному споживачу або всім зацікавленим сторонам. Канал відповіді, зазвичай, є каналом точка-до-точки, оскільки передача відповідей широкому колу отримувачів зазвичай не має сенсу. Детальніше можна переглянути тут [18]

В нашому застосунку це реалізовано у наприклад прийнятті запрошення. Цей процес є на Рисунку 2.4. Де сервер надсилає запит на під'єднання до чату та потім отримує від воркера відповідь і тільки тоді повідомляє про це клієнту.

2.2.9. Паттерн Command Message

Паттерн "Командне Повідомлення" [19] визначає одну з компонент системи, скажімо компонент А, як ініціатор команди, а компонент В — як ціль команди. У цьому патерні завжди існує лише один цільовий об'єкт для команди. Ініціатор знає, кому надіслати команду і не очікує негайної відповіді. Якщо ініціатор очікує відповідь, то вона може прийти в майбутньому іншим способом. З точки зору ініціатора, це операція типу "надіслати та забути".

В моєму проекті так працює створення повідомлення. Сервер надсилає команду воркеру створити повідомлення. І воркер її виконує.

2.2.10. Паттерн Dead letter Queue

Паттерн Dead Letter Queue (DLQ) [20] використовується в системах обміну повідомленнями, таких як RabbitMQ, для обробки повідомлень, які не можуть бути доставлені або оброблені з якоїсь причини. Він дозволяє зберігати ці повідомлення в спеціальній черзі, замість того, щоб їх відкидати, надаючи можливість подальшого аналізу та обробки.

З виходом RabbitMQ версії 3.10, було введено новий підхід до dead lettering, який називається "at-least-once" dead lettering. Цей підхід гарантує, що повідомлення, які потрапляють в DLQ, обов'язково будуть доставлені в цільові черги, навіть якщо з'являються помилки або збої у мережі, що значно підвищує надійність системи. Ця функція вимагає включення певних налаштувань, таких як встановлення політики dead-letter-strategy у значення at-least-once та конфігурації dead-letter-exchange.

Цей патерн є особливо важливим для систем, де втрата повідомлень може привести до серйозних проблем, наприклад, у фінансових застосунках, системах замовлень або в інших критичних додатках. Використання DLQ дозволяє забезпечити стійкість та надійність системи обміну повідомленнями.

В моїй системі це побудовано так, що якщо під час обробки повідомлення сталась будь-яка помилка, це повідомлення буде надіслано до DLQ, де воно буде зберігатись доки я вручну його не отримаю. Так я зможу швидко знайти присутнісь помилок в системі та виправити їх, і перенадіслати ці повідомлення, чи поправити помилку в самому повідомлення та знову надіслати його після виправлення помилки.

2.2.11. Мікро сервісна архітектура

Мікросервісна архітектура - це метод розробки програмного забезпечення, в якому додаток розбивається на набір невеликих, незалежних сервісів, які виконують певні бізнес-функції і спілкуються між собою за допомогою легких протоколів, зазвичай HTTP. Кожен мікросервіс є самодостатнім і може бути розгорнутий незалежно. Це дозволяє різним командам працювати паралельно, спрощує розширення та масштабування додатку, а також полегшує виявлення і виправлення помилок.

У моєму проекті, мікросервіс для обміну повідомленнями є моїм єдиним і найвижливішим сервісом. Він складається з сервера, що приймає запити від користувачів, воркера, що обробляє бізнес-логіку, бази даних Postgres для збереження інформації та RabbitMQ для управління повідомленнями між компонентами.

Мікросервісна архітектура дозволяє моєму проекту бути гнучким і масштабованим, з кожним сервісом, що фокусується на конкретній функції. Так, в мене могли б бути сервіс зберігання медіа, він би займався обробкою, стисканням та збереженням медіафайлів, таких як фотографії та відео, що користувачі обмінюються в чатах. Можливо ще сервіс сповіщень, який відповідав би за відправлення електронних листів, інформуючи користувачів про важливі події в системі.

2.2.12. Використання Device Authorization Flow

Device Authorization Flow є частиною OAuth 2.0 і представляє собою протокол для авторизації пристройв, які не мають можливості безпосереднього вводу облікових даних користувача, таких як смарт-ТВ, ігрові консолі або пристрої з обмеженим інтерфейсом користувача. Цей протокол забезпечує безпечний спосіб для користувачів надати свої облікові дані через веб-браузер на іншому пристрої, такому як смартфон або комп'ютер.

Коли користувач хоче увійти у додаток на пристрой, додаток запитує у сервера авторизації спеціальний користувацький код. Сервер авторизації повертає цей код разом із URL-адресою, куди користувач має перейти для входу в систему. Користувач вводить отриманий код на цій веб-сторінці у своєму браузері на іншому пристрої, що має повноцінні можливості вводу. Після вводу коду та успішної авторизації користувача, сервер авторизації надсилає access token на пристрой, який тепер може використовувати цей токен для доступу до захищених ресурсів.

Така схема входу є безпечною, оскільки користувацькі облікові дані не передаються через менш захищений пристрой, і не вимагає від користувачів запам'ятовувати та вводити складні паролі на пристроях з обмеженим інтерфейсом. Замість цього, вони використовують знайомий і безпечний інтерфейс веб-браузера на своєму основному пристрої. Це також знижує ризик фішингу, оскільки користувачі можуть бути більш впевнені у легітимності веб-сторінки авторизації, яку вони вже знають і довіряють.

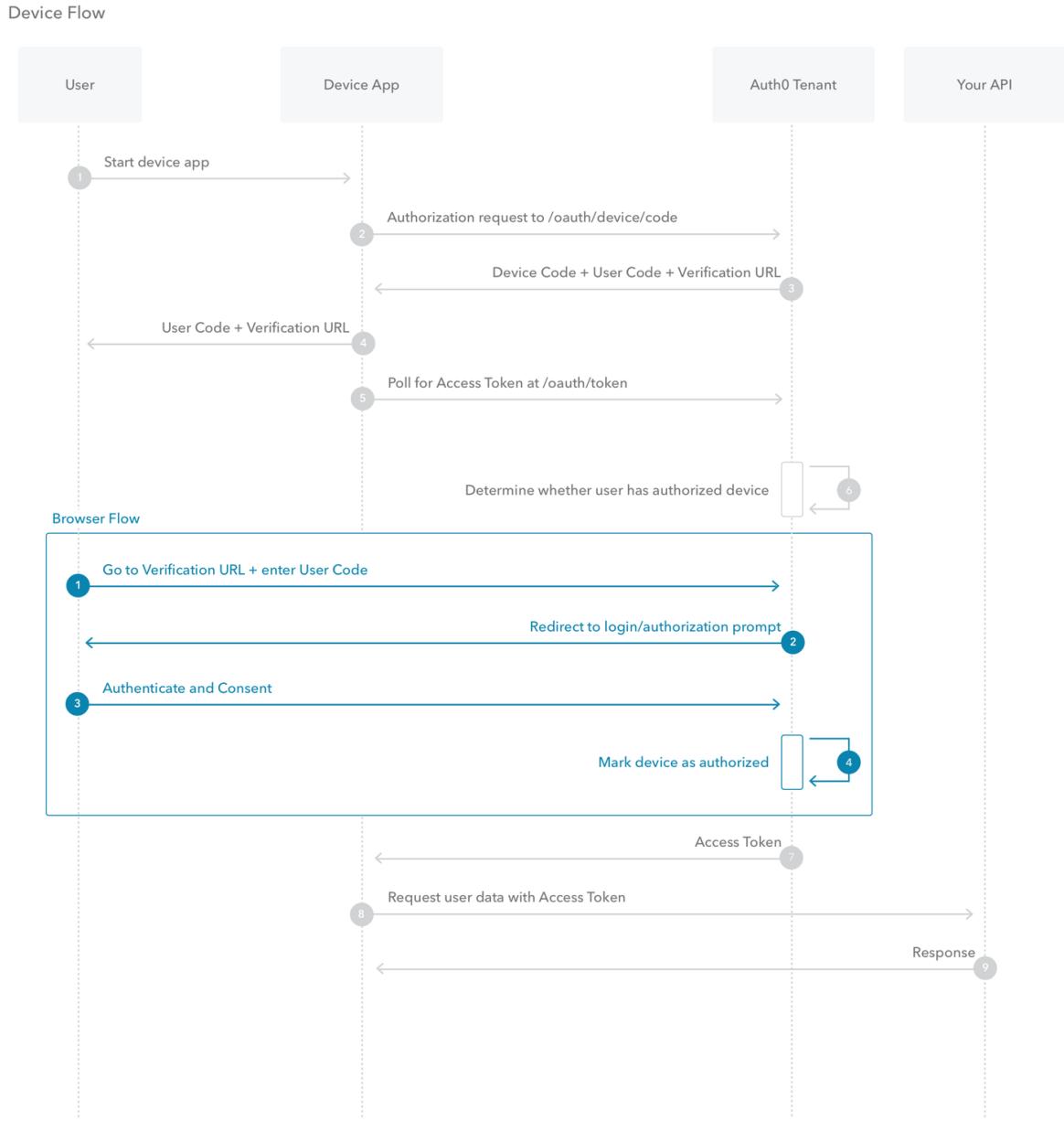


Рисунок 2.12 - Діаграма Device Authorization Flow

Тепер, звернемось до діаграми, що ілюструє процес Device Authorization Flow [21]:

1. Запуск додатку на пристрой: Користувач ініціює додаток, який потребує авторизації.
2. Запит на авторизацію: Додаток на пристрой відправляє запит до сервера авторизації Auth0 через спеціальний кінцевий пункт.
3. Отримання кодів: Auth0 відповідає, надаючи Device Code для додатка та User Code з Verification URL для користувача.

4. Введення коду користувачем: Користувач переходить до вказаного URL у браузері і вводить отриманий User Code.
5. Аутентифікація та згода: Після перенаправлення на сторінку логіну, користувач входить в систему і дає згоду на авторизацію пристрою.
6. Підтвердження авторизації: Auth0 позначає пристрій як авторизований.
7. Запит на Access Token: Додаток регулярно запитує токен доступу до тих пір, поки користувач не завершить авторизацію на веб-сторінці.
8. Отримання Access Token: Коли авторизація завершена, Auth0 надає Access Token додатку.
9. Використання Access Token: З допомогою цього токену додаток може запитувати дані користувача з API.

Цей метод авторизації є важливим для забезпечення безпеки в моєму проекті, де клієнт у терміналі користувача, і де користувач ніколи не вводить свій пароль і я не зберігаю його, оскільки використовую Auth0 для керування авторизацією.

3. ІНСТРУКЦІЯ КОРИСТУВАЧА

3.1. Встановлення

В контексті сучасної розробки програмного забезпечення, особливу увагу приділяється питанням ефективності управління залежностями та пакетами. У мові програмування C++, це питання часто призводить до складнощів, зумовлених відсутністю єдиного стандартизованого інструменту управління пакетами та залежностями, що може ускладнити процес компіляції та розгортання програм. На противагу цьому, мова програмування Rust використовує інструмент Cargo, який є втіленням філософії "convention over configuration", тобто надання переваги домовленостям перед налаштуванням.

Cargo автоматизує багато аспектів роботи з Rust-проектами, включаючи компіляцію, завантаження залежностей, збирання пакетів та їх публікацію. crates.io виступає як централізоване сховище, що містить тисячі пакетів ("crates"), що значно спрощує обмін кодом та повторне використання програмних компонентів у спільноті Rust.

З допомогою інструментарію Cargo, процес інсталяції застосунку-клієнта мого месенджера зводиться до елементарної виконання однієї команди. Ця процедура є прикладом високорівневої автоматизації в маніпуляціях із програмним забезпеченням, яка відбувається у введенні команди cargo install crab-messenger у командному рядку. Cargo, як маніфестація модерного підходу до управління залежностями та пакетами, обіцяє значне спрощення розробки та розгортання застосунків, що підкріплено широким спектром функціональних можливостей, забезпечуючи при цьому високий рівень інтеграції та узгодженості залежностей.

Інший варіант встановлення – це просте завантаження скомпільованого артефакту в релізі з репозиторію. Цей метод може бути більш трудомістким для користувачів, оскільки потребує від них додаткових кроків: визначення сумісної версії залежно від операційної системи та архітектури процесора, ручного завантаження та встановлення. Крім того, цей метод не забезпечує автоматичного управління залежностями, яке виконує Cargo, та може вимагати від користувачів вручну встановлювати необхідні бібліотеки та налаштовувати системні змінні. У ситуації,

коли безпосереднє використання Cargo не є можливим, цей підхід може стати в нагоді, але він потребує більш глибоких технічних знань та розуміння процесу конфігурації програмного забезпечення.

3.2. Налаштування

Якщо користувач хоче встановити мій сервер та використовувати його замість моого основного, який повинен буди в Azure клауді, це можна зробити налагодивши змінні оточення. Сервер знає його деталі такі як audience, origin, де йому хоститься, параметри під'єднання до бази даних, робіту зі змінних оточення, та можливо встановити в себе rabbitmq та postgres, налагодити auth0 tenant, на налаштувати змінні оточення для використання цього всього для отримання свого серверу.

В такому випадку також потрібно буде налаштувати клієнт, щоб той використовував вашу адресу серверу та auth0 під час роботи. Оскільки це клієнтський застосунок, замість змінних оточення, необхідні параметри завантажуються в клієнта під час компіляції, щоб той міг одразу працювати за замовчуванням при першому запуску. Але змінні оточення мають пріоритет, тому можна перегалагодити його для роботи з вашим сервером

Цей підхід дозволить будь яким бажаючим використати клієнт з моїм сервером за замовчуванням, або створити свою підмережу зробивши відповідні налаштування. Цей проект має ліцензію MIT, тобто він дозволяє вільне використання, модифікацію та розповсюдження як у вихідному, так і в зміненому вигляді, у комерційних та некомерційних цілях, за умови збереження тексту ліцензії та авторських прав разом з програмним забезпеченням.

3.3. Використання серверу

В цьому розділі будуть надані тестові запити та відповіді від сервера на запити. Це буде зроблено за допомогою маленьких тестових бінарних застосунків, які імітують клієнта але ним не являються. Це зроблено число для тестування того як працює сервер.

3.3.1. Обмін повідомленнями

Для тесту, я зайшов з двох простих клієнтів одночасно та почав надсилати повідомлення.

```
Hello, I'm simple client 1

Received message: Message { id: 183, user_id: "auth0|657821f5a1e9bf99450fff22", chat_id: 1, text: "Hello, I
'm simple client 1", created_at: Some(Timestamp { seconds: 1703886562, nanos: 595200000 }) }

Received message: Message { id: 184, user_id: "google-oauth2|108706181521622783833", chat_id: 1, text: "Hi
there, I'm simple client 2", created_at: Some(Timestamp { seconds: 1703886569, nanos: 204296000 }) }

Pleased to meet ya

Received message: Message { id: 185, user_id: "auth0|657821f5a1e9bf99450fff22", chat_id: 1, text: "Pleased
to meet ya", created_at: Some(Timestamp { seconds: 1703886580, nanos: 888474000 }) }

Received message: Message { id: 186, user_id: "google-oauth2|108706181521622783833", chat_id: 1, text: "Sam
e, bro", created_at: Some(Timestamp { seconds: 1703886584, nanos: 243731000 }) }
```

Рисунок 3.1 - Обмін повідомленнями 1

```
Received message: Message { id: 183, user_id: "auth0|657821f5a1e9bf99450fff22", chat_id: 1, text: "Hello, I
'm simple client 1", created_at: Some(Timestamp { seconds: 1703886562, nanos: 595200000 }) }

Hi there, I'm simple client 2

Received message: Message { id: 184, user_id: "google-oauth2|108706181521622783833", chat_id: 1, text: "Hi
there, I'm simple client 2", created_at: Some(Timestamp { seconds: 1703886569, nanos: 204296000 }) }

Received message: Message { id: 185, user_id: "auth0|657821f5a1e9bf99450fff22", chat_id: 1, text: "Pleased
to meet ya", created_at: Some(Timestamp { seconds: 1703886580, nanos: 888474000 }) }

Same, bro

Received message: Message { id: 186, user_id: "google-oauth2|108706181521622783833", chat_id: 1, text: "Sam
e, bro", created_at: Some(Timestamp { seconds: 1703886584, nanos: 243731000 }) }
```

Рисунок 3.2 - Обмін повідомленнями 2

Як видно з цих рисунків, 2 простих клієнти запущені паралельно в двох потоках, та при відправці повідомлення одним, обидва його отримують, бо вони знаходяться в одному чат румі. Але оскільки повідомлення надходять від усіх чат румів, вони могли

бути і у різних, та отримувати повідомлення якщо в один з їх чат румів надійшло повідомлення.

181	183 Hello, I'm simple client 1	2023-12-29 21:49:22.595200 +00:00	auth0 657821f5a1e9bf99450fff22	1
182	184 Hi there, I'm simple client 2	2023-12-29 21:49:29.204296 +00:00	google-oauth2 108706181521622783833	1
183	185 Pleased to meet ya	2023-12-29 21:49:40.888474 +00:00	auth0 657821f5a1e9bf99450fff22	1
184	186 Same, bro	2023-12-29 21:49:44.243731 +00:00	google-oauth2 108706181521622783833	1

Рисунок 3.3 - Повідомлення в базі даних

Як видно з цього малюнку, надіслані повідомлення з'явились у базі даних у відповідному чаті від відповідних користувачів. Це свідчить про правильну роботу системи збереження та маршрутизації повідомлень.

/	ErrorExchange	fanout
/	S_ChatConnectExchange-auth0 657821f5a1e9bf99450fff22	fanout
/	S_ChatConnectExchange-google-oauth2 108706181521622783833	fanout
/	S_MessagesExchange-1	fanout
/	S_MessagesExchange-2	fanout
/	S_MessagesExchange-3	fanout
/	S_MessagesExchange-4	fanout
/	S_MessagesExchange-5	fanout
/	W_AcceptInviteExchange	direct
/	W_NewMessageExchange	direct
/	W_SendInviteExchange	direct

Рисунок 3.4 - RabbitMQ Exchanges

На цьому малюнку видно як саме працює RabbitMQ. Було створено ексчендж для помилок як частина паттерну DLQ, для кожного користувача було створено ChatConnectExchange, в які будуть надходити команди для під'єднання клієнта до нового чату, та оскільки в нас 2 клієнти активних, в нас 2 ексченджи. Далі для кожного чату в яких є клієнти було створено ексчендж, та клієнти під'єднані тільки якщо вони є в тому чаті. Це були серверні ексченджи, тобто серверу вони потрібні для читання.

Наступні – ексченджи воркера. На відміну від серверних, ці мають тип direct, тобто тільки один воркер може обробити повідомлення, а в fanout, повідомлення копіюється всім під'єднаним. Є 3 різних ексченджи, для надсилання нових

повідомлень, щоб вони збереглись в базі даних та далі потрапили до чату, для відправки запрошення, бо треба буде його клієнту надсилати, та для прийняття запрошення, бо треба правильному серверу відпрацювати підключення клієнта до нового чату.

Overview				
Virtual host	Name	Type	Features	State
/	2C8FVrxzjUbKaIUW	classic	AD	idle
/	Cibw4DYcxsU7xO4S	classic	AD	idle
/	ErrorQueue	classic	D	idle
/	accept_invite_queue	classic	D	idle
/	connect-2C8FVrxzjUbKaIUW	classic	AD	idle
/	connect-Cibw4DYcxsU7xO4S	classic	AD	idle
/	new_message_queue	classic	D	idle
/	send_invite_queue	classic	D	idle

Рисунок 3.5 - RabbitMQ Queues

На цьому малюнку можна побачити які саме черги були створені для клієнтів. Перші 2 – черги для відправки повідомлень до клієнта, вони мають унікальний ідентифікатор на кожне з’єднання клієнта, вони під’єднані до ексченджей в чатах яких є ці клієнти. Також для них є черга під’єднання, туди надійде команда на під’єднання клієнта по певного чату. Також є черги для воркера, з ними все просто.

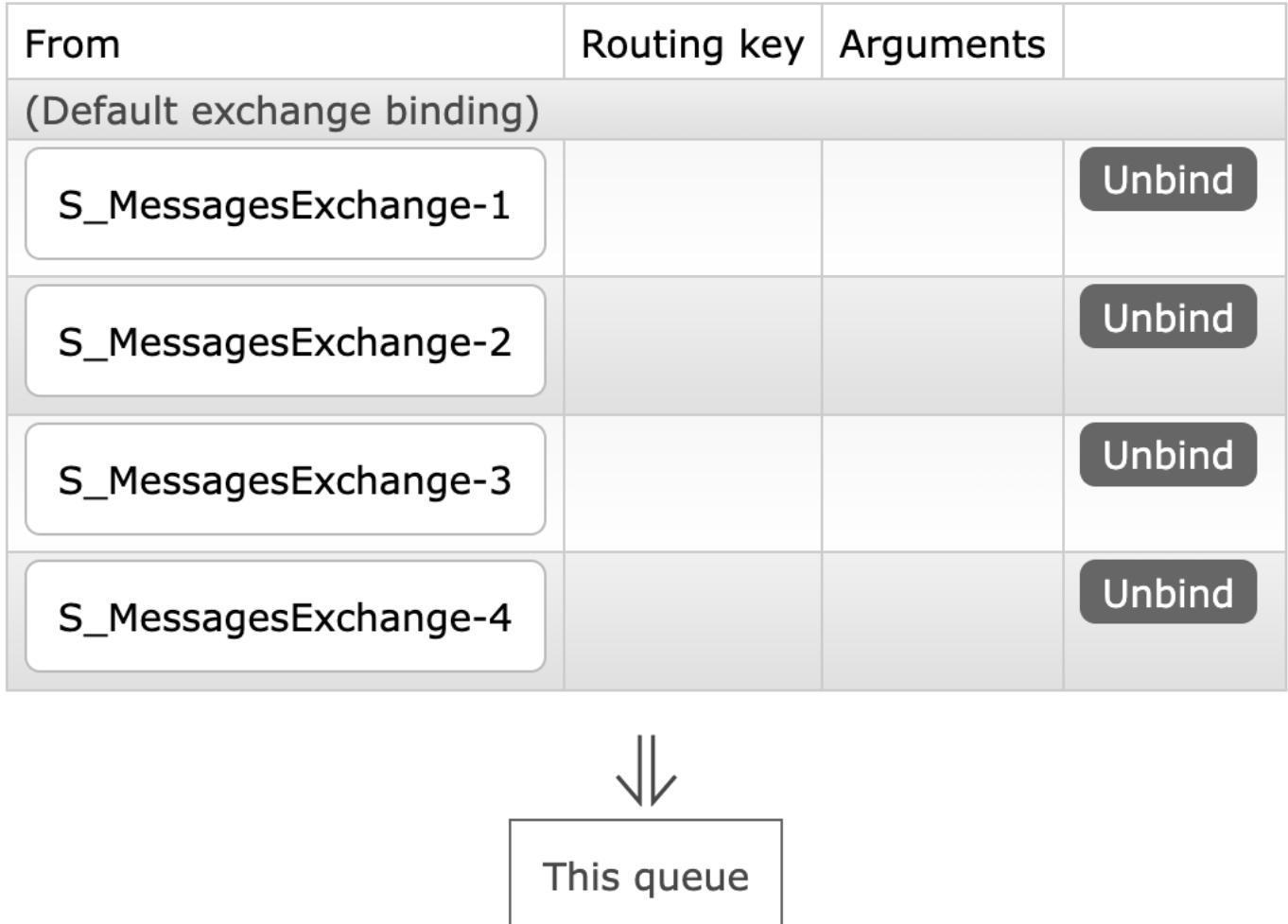


Рисунок 3.6 - Під'єднання черги 1

Тут можемо побачити що перша черга під'єднана тільки до 4 чатів.

From	Routing key	Arguments	
(Default exchange binding)			
S_MessagesExchange-1			Unbind
S_MessagesExchange-2			Unbind
S_MessagesExchange-3			Unbind
S_MessagesExchange-4			Unbind
S_MessagesExchange-5			Unbind

Рисунок 3.7 - Під'єднання черги 2

А цей клієнт має під'єднання до п'яти чатів, тобто він буде отримувати повідомлення від них усіх.

	User_id	chat_id
1	auth0 657821f5a1e9bf99450ffff22	1
2	auth0 657821f5a1e9bf99450ffff22	2
3	auth0 657821f5a1e9bf99450ffff22	3
4	auth0 657821f5a1e9bf99450ffff22	4
5	google-oauth2 104747573923008450838	1
6	google-oauth2 108706181521622783833	1
7	google-oauth2 108706181521622783833	2
8	google-oauth2 108706181521622783833	3
9	google-oauth2 108706181521622783833	4
10	google-oauth2 108706181521622783833	5

Рисунок 3.8 - Належність користувачів до чатів

На цьому малюнку бачимо, що один з під'єднаних клієнтів є тільки в 4 чатах, тому його черга під'єднана тільки до чотирьох екскенджерів, в той час як другий клієнт, там де 108, під'єднаний до всіх п'яти чатів.

3.3.2. Отримання попередніх повідомлень

Цей ґру створений щоб завантажити в чат попередні повідомлення.

```
: 1, text: "Hello, I'm simple client 1", created_at: Some(Timestamp { seconds: 1703886562, nanos: 595200000 }) }, Message { id: 184, user_id: "google-oauth2|108706181521622783833", chat_id: 1, text: "Hi there, I'm simple client 2", created_at: Some(Timestamp { seconds: 1703886569, nanos: 204296000 }) }, Message { id: 185, user_id: "auth0|657821f5a1e9bf99450ffff22", chat_id: 1, text: "Pleased to meet ya", created_at: Some(Timestamp { seconds: 1703886580, nanos: 888474000 }) }, Message { id: 186, user_id: "google-oauth2|108706181521622783833", chat_id: 1, text: "Same, bro", created_at: Some(Timestamp { seconds: 1703886584, nanos: 243731000 }) } ] }, extensions: Extensions }
```

Рисунок 3.9 - Завантаження попередніх повідомлень

На цьому рисунку видно попередні повідомлення з чату 1, як раз ті, що ми нещодавно надіслали.

3.3.3. Завантаження чатів користувача

```
RESPONSE=Response { metadata: MetadataMap { headers: {"content-type": "application/grpc", "date": "Fri, 29 Dec 2023 22:26:05 GMT", "grpc-status": "0" } }, message: Chats { chats: [Chat { id: 1, name: "Dad and I" }, Chat { id: 2, name: "super secret chat" }, Chat { id: 3, name: "ShattaMatte" }, Chat { id: 4, name: "Test c hat" }, Chat { id: 5, name: "Test chat 2" } ] }, extensions: Extensions }
```

Рисунок 3.10 - Завантаження чатів користувача

На цьому рисунку видно виконання запиту на чати в яких є поточний користувач. В запиті нема жодних даних, але в хедері аутентифікації є `access_token` користувача, за допомогою якого сервер і дізнається хто це.

3.3.4. Завантаження знайомих

```
RESPONSE=Response { metadata: MetadataMap { headers: { "content-type": "application/grpc", "date": "Fri, 29 Dec 2023 22:29:27 GMT", "grpc-status": "0" } }, message: Users { users: [User { id: "auth0|657821f5a1e9bf9450fff22", email: "████████████████████████████████████████", User { id: "google-oauth2|104747573923008450838", email: "va████████████████████████████████████████", User { id: "google-oauth2|108706181521622783833", email: "va████████████████████████████████████████", User { id: "████████████████████████████████████████", extensions: Extensions } ] }, extensions: Extensions }
```

Рисунок 3.11 - Завантаження знайомих

Цей віддалений виклик процедури завантажує інформацію про користувачів які є в одному чаті з поточним, щоб можна було відобразити їх email в якості імені в чаті.

3.3.5. Створення чату

```
RESPONSE=Response { metadata: MetadataMap { headers: {"content-type": "application/grpc", "date": "Fri, 29 Dec 2023 22:39:24 GMT", "grpc-status": "0"}, message: CreateChatResponse { chat: Some(Chat { id: 6, name: "Chat for course" }) }, extensions: Extensions }  
base ~/proj/Rust/crab-messenger}❯ main 7❯ 1+ 2❯ %
```

Рисунок 3.12 - Створення чату

	id	name
1		Dad and I
2		super secret chat
3		ShattaMatte
4		Test chat
5		Test chat 2
6		Chat for course

Рисунок 3.13 - Створений чат у базі даних

На цих рисунках видно як було виконано запит на створення нового чату, у відповідь надійшла інформація про цей чат, щоб його одразу можна було використовувати.

Також він створився у базі даних, та як видно на наступному малюнку автор автоматично додався до нього

8	google-oauth2 108706181521622783833	3
9	google-oauth2 108706181521622783833	4
10	google-oauth2 108706181521622783833	5
11	google-oauth2 108706181521622783833	6

Рисунок 3.14 - Автор у чаті

3.3.6. Надсилання запрошення

```
RESPONSE=Response { metadata: MetadataMap { headers: {"content-type": "application/grpc", "date": "Fri, 29 Dec 2023 23:15:08 GMT", "grpc-status": "0" } }, message: SendInviteResponse { success: true }, extensions: Extensions }
```

Рисунок 3.15 - Надсилання запрошення

	id	inviter_user_id	invitee_user_id	chat_id	created_at
1	30	google-oauth2 108706181521622783833	auth0 657821f5a1e9bf99450fff22	5	2023-12-29 23:19:59.878505 +00:00
2	31	google-oauth2 108706181521622783833	auth0 657821f5a1e9bf99450fff22	6	2023-12-29 23:20:12.022987 +00:00

Рисунок 3.16 - Створені запрошення

Як видно на рисунках, запрошення були створені успішно. Далі переглянемо чи були вони отримані в реальному часі.

3.3.7. Отримання запрошення в реальному часі

```
Received message: Invite { id: 35, inviter_user_id: "google-oauth2|108706181521622783833", invitee_user_id: "auth0|657821f5a1e9bf99450fff22", chat_id: 6, created_at: Some(Timestamp { seconds: 1703892350, nanos: 248062000 }) }
```

Рисунок 3.17 - Отримання запрошення в реальному часі

Це працювало під час тестування надсилання, та ми отримали це повідомлення в реальному часі

3.3.8. Отримання всіх запрошень користувача

```
RESPONSE=Response { metadata: MetadataMap { headers: {"content-type": "application/grpc", "date": "Fri, 29 Dec 2023 23:32:47 GMT", "grpc-status": "0"} }, message: GetInvitesResponse { invites: [Invite { id: 35, inviter_user_id: "google-oauth2|108706181521622783833", invitee_user_id: "auth0|657821f5a1e9bf99450fff22", chat_id: 6, created_at: Some(Timestamp { seconds: 1703892350, nanos: 248062000 }) }, Invite { id: 36, inviter_user_id: "google-oauth2|108706181521622783833", invitee_user_id: "auth0|657821f5a1e9bf99450fff22", chat_id: 5, created_at: Some(Timestamp { seconds: 1703892367, nanos: 506534000 }) }] }, extensions: Extensions }
```

Рисунок 3.18 - Отримання всіх запрошень

Як видно, ті самі запрошення, що ми надіслали та отримали в реальному часі ми можемо переглянути потім, навіть після від'єднання. А тепер давайте на одне відповімо, а на інше ні.

3.3.9. Відхилення запрошення

```
RESPONSE=Response { metadata: MetadataMap { headers: {"content-type": "application/grpc", "date": "Fri, 29 Dec 2023 23:37:06 GMT", "grpc-status": "0"} }, message: AnswerInviteResponse { success: true }, extensions: Extensions }
```

Рисунок 3.19 - Відхилення запрошення

	id	inviter_user_id	invitee_user_id	chat_id	created_at
1	35	google-oauth2 108706181521622783833	auth0 657821f5a1e9bf99450fff22	6	2023-12-29 23:25:50.248062 +00:00

Рисунок 3.20 - Видалення відхиленого повідомлення

	user_id	chat_id
1	auth0 657821f5a1e9bf99450fff22	1
2	auth0 657821f5a1e9bf99450fff22	2
3	auth0 657821f5a1e9bf99450fff22	3
4	auth0 657821f5a1e9bf99450fff22	4
5	google-oauth2 104747573923008450838	1
6	google-oauth2 108706181521622783833	1
7	google-oauth2 108706181521622783833	2
8	google-oauth2 108706181521622783833	3
9	google-oauth2 108706181521622783833	4
10	google-oauth2 108706181521622783833	5
11	google-oauth2 108706181521622783833	6

Рисунок 3.21 - Відсутність належності до чату

Як видно, після запиту на видалення повідомлення, воно зникло з бази даних та нового з'єднання не з'явилось.

3.3.10. Прийняття запрошення

From	Routing key	Arguments	
(Default exchange binding)			
S_MessagesExchange-1			Bind
S_MessagesExchange-2			Bind
S_MessagesExchange-3			Bind
S_MessagesExchange-4			Bind

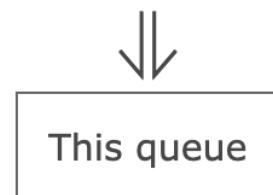


Рисунок 3.22 - Стан з'єднань до прийняття

```

HTTP/1.1 200 OK
Content-Type: application/json
Date: Fri, 29 Dec 2023 23:43:08 GMT
Server: Apache/2.4.41 (Ubuntu)
Content-Length: 113
Connection: keep-alive
Etag: "544-5440000000000"
Last-Modified: Fri, 29 Dec 2023 23:43:08 GMT
Accept-Ranges: bytes, messages, invitees

RESPONSE=Response { metadata: MetadataMap { headers: { "content-type": "application/grpc", "date": "Fri, 29 Dec 2023 23:43:08 GMT", "grpc-status": "0" } }, message: AnswerInviteResponse { success: true }, extensions: Extensions }

base ~/proj/Rust/crab-messenger % main 12 1+ 2 %
  
```

Рисунок 3.23 - Запит на прийняття

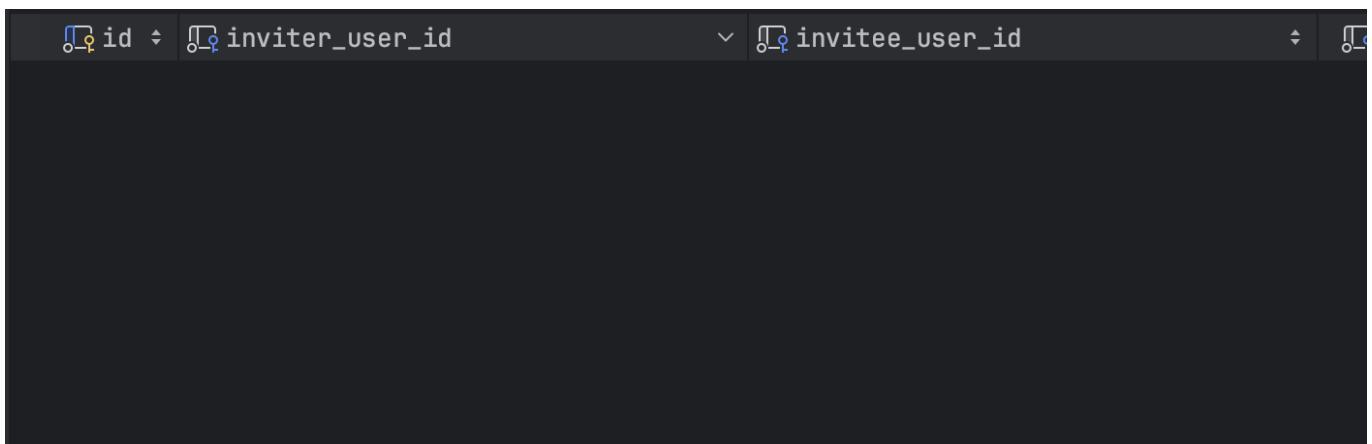


Рисунок 3.24 - Відсутність запрошення

	👤 user_id	^K	👤 chat_id
1	auth0 657821f5a1e9bf99450fff22		1
2	auth0 657821f5a1e9bf99450fff22		2
3	auth0 657821f5a1e9bf99450fff22		3
4	auth0 657821f5a1e9bf99450fff22		4
5	auth0 657821f5a1e9bf99450fff22		6
6	google-oauth2 104747573923008450838		1
7	google-oauth2 108706181521622783833		1
8	google-oauth2 108706181521622783833		2
9	google-oauth2 108706181521622783833		3
10	google-oauth2 108706181521622783833		4
11	google-oauth2 108706181521622783833		5
12	google-oauth2 108706181521622783833		6

Рисунок 3.25 - Належність користувача до нового чату

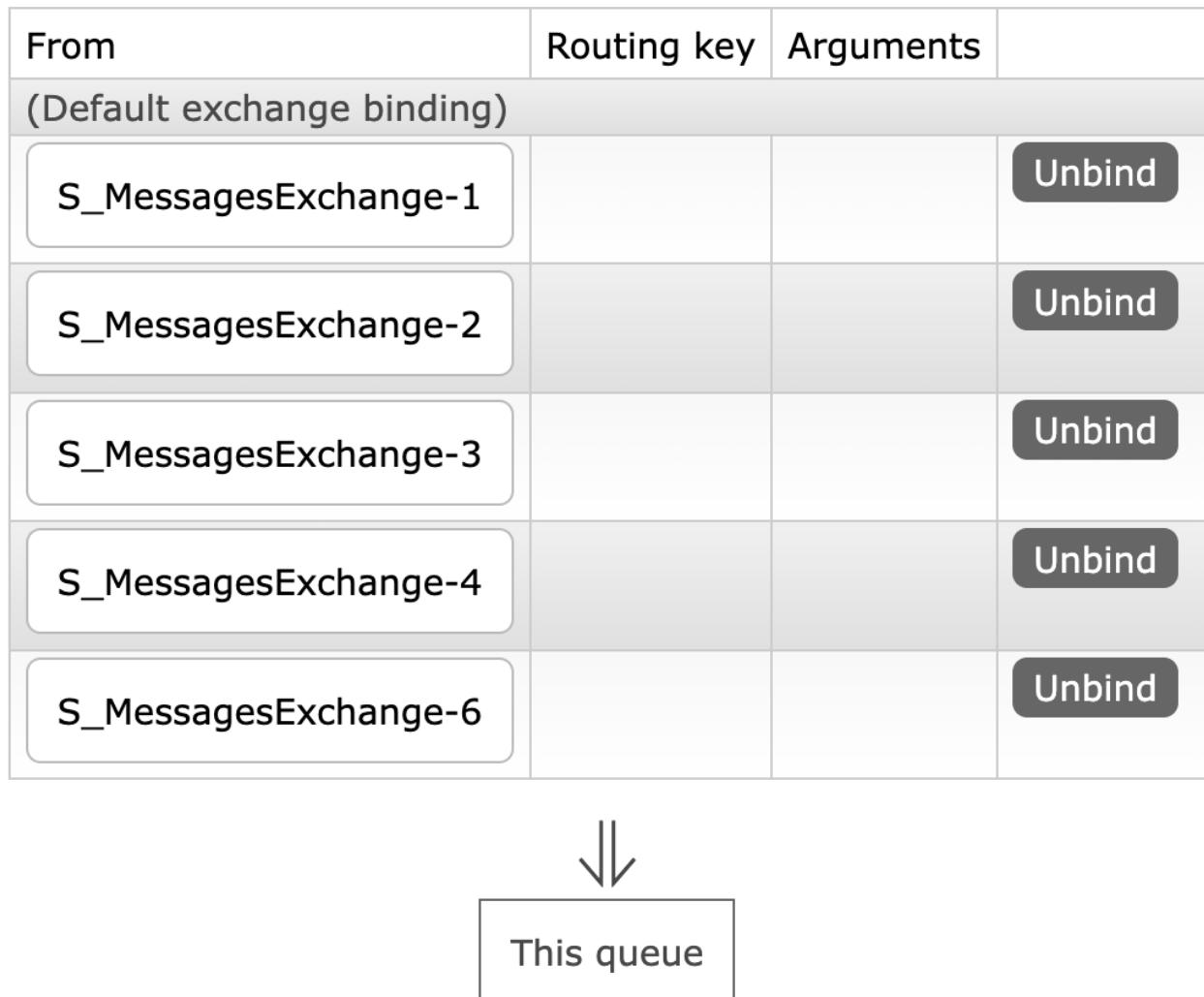


Рисунок 3.26 - Стан з'єднань після прийняття

Як видно, після прийняття повідомлення сталося багато чого: Надійшла успішна відповідь клієнту, зникло повідомлення з бази даних, користувач тепер є в новому чаті, та найголовніше його черга під'єднана до екченджу нового чату, що значить що він буде отримувати повідомлення з нього.

3.3.11. Логін

Це єдина частина справжнього клієнту яку було зроблено на цей момент. Він працює за вже описаним флоу, перегляньмо як це виглядає для користувача.

The screenshot shows a terminal window with four tabs at the top:

- /proj/Rust/crab-messenger — server
- /proj/Rust/crab-messenger — worker
- /proj/Rust/crab-messenger — client
- /proj/Rust/crab-messenger — -zsh

The "client" tab is active, displaying the following text:

```
Login
Welcome to the Crab messenger, a terminal messenger written fully in Rust! Please login
Login link will soon open in your browser. Please log in there and then proceed in this app.
If it didn't work, or you know you don't have ui browser, please visit this link:
https://crab-messenger.eu.auth0.com/activate , and enter this code: HXXF-DZFP on your phone
This is your access_token: No token
This is your id_token: No id
```

Рисунок 3.27 - Логін сторінка клієнту

Коли користувач вперше відкриває клієнт, йому потрібно буде залогінитись, для цього треба перейти на відповідне посилання та ввести наданий йому код. Клієнт автоматично відкриє дефольтний браузер користувача з вже вставленим кодом.

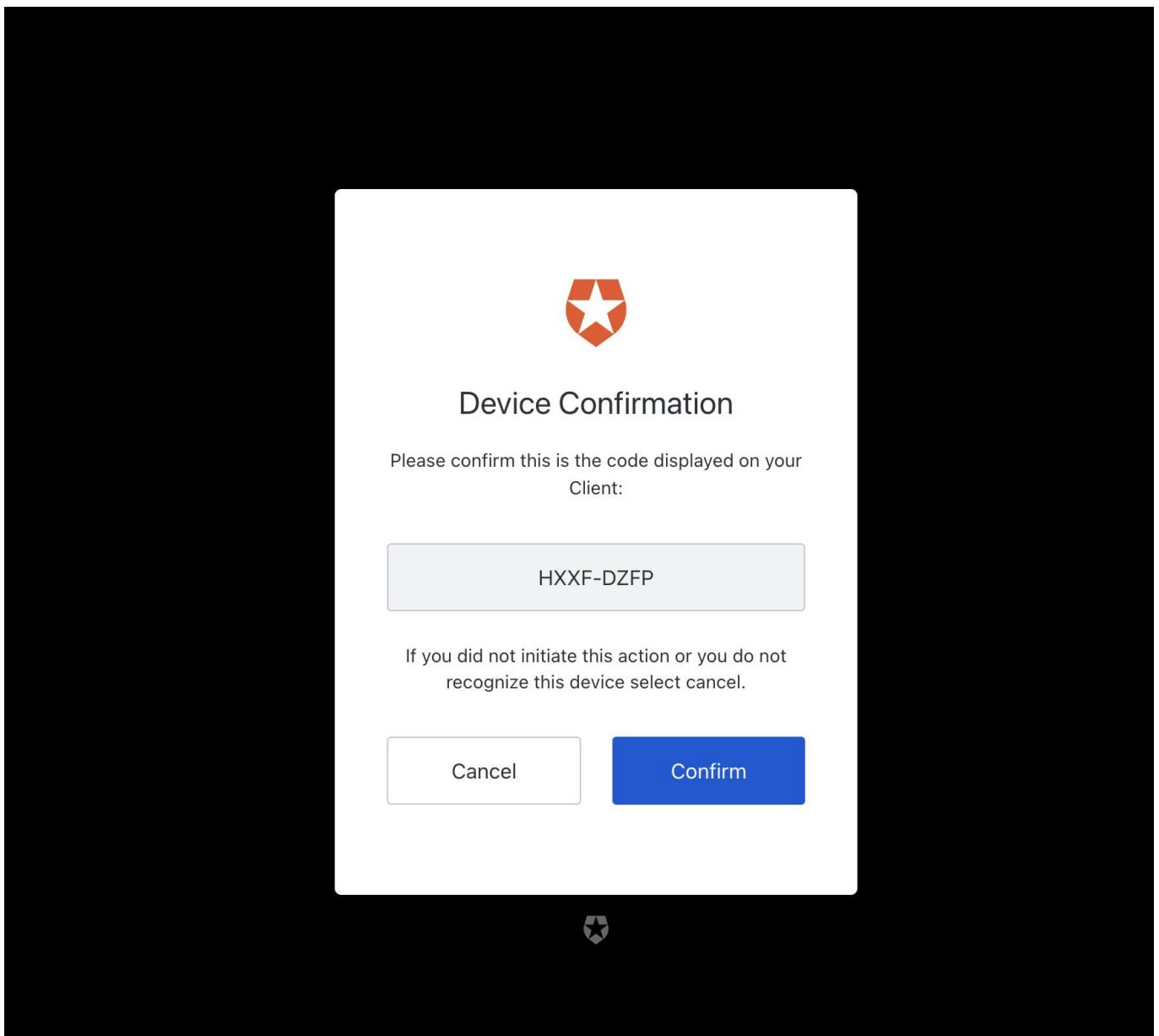


Рисунок 3.28 - Логін сторінка в браузері

Після натискання кнопки Confirm, потрібно буде зайти в свій аккаунт або створити новий. Можна використати google identity provider, щоб вашого пароля не було навіть в базі даних Auth0, а ваш Google аккаунт – це найзахищений профіль що в вас тільки є, тому під ним зайди ніхто не зможе окрім вас.

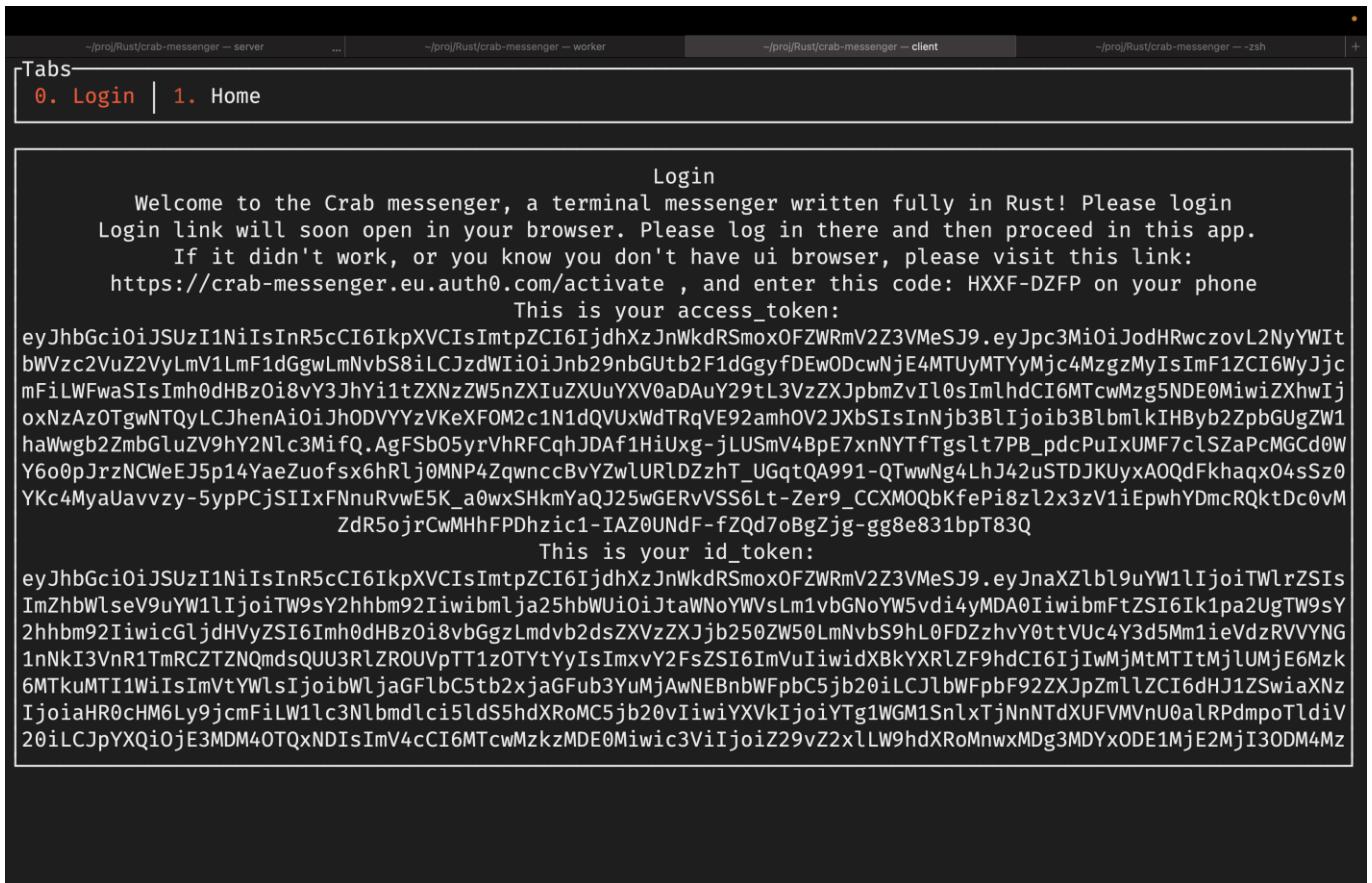


Рисунок 3.29 - Логін сторінка після логіну

Як видно на цьому рисунку, після підтвердження коду, клієнт отримав токен, та просто його вивів. Це просто proof of concept моєго патерну Redux, та я його використовував для тестування сервера.

Щодо іншого функціоналу клієнта, можна натиснути клавішу 1, щоб перейти до Home сторінки.

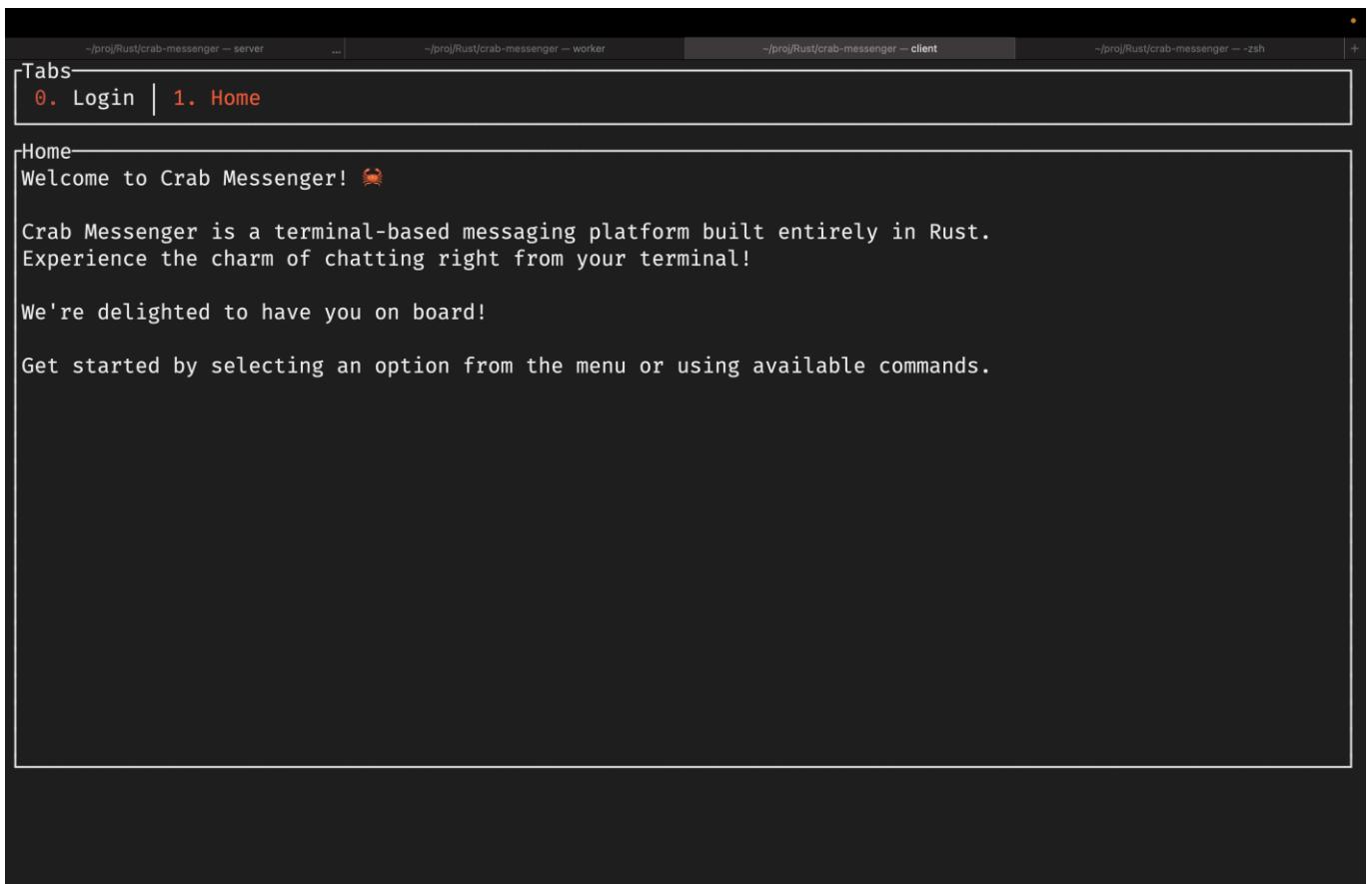


Рисунок 3.30 - Home сторінка

4. НАСТУПНІ КРОКИ ПРОЕКТУ

Протягом цього проекту було написано багато цікавинок, але він не ідеален, тому тут я спробую пояснити можливі кроки для покращення проекту.

Напевно найперше що я хочу сказати, дякую що дочитали та зацікавились моїм проектом. Мені цікаво було це все розробляти на сподіваюсь вам сподобалось.

Далі, в мене нема способу вийти з чату. Для цього потрібно буде задати нового конс'юмера для видалення з'єднання клієнта до того екскенджу щоб він більше не отримував звідти повідомлень, ну і з бази даних його видалити.

Переписати всі відповіді сервера у випадках помилок на менш точні, щоб складніше було взломати сервер. Також якщо користувач не знаходиться в чаті кидати йому не 403 а 404, щоб не можна було знайти id чатів які вже є.

Думаю ще треба було оформити нормальну сертифікат, щоб було шифрування у з'єднанні нормальне.

Переписати всі повідомлення з json серіалізації на protoc, бо... бінарна швидша та більш компактна за текстову серіалізацію.

Просто можна багато порефакторити з більшим ооп, бо часу не було зробити максимально красиво.

Та ще переписати перші спроби інджектування, я там поки руку набив некрасоти наробив.

Ну і ще це, клієнта нормального написати.

Зробити ci/cd пайплайні для автоматичного оновлення серверу та версії клієнту в cargo.

А ще це, зробити downtime-less оновлення сервера, де після запуску нової версії запустити її паралельно з попередньою, але всі запити перенаправляти на нову, та чекати коли всі поточні запити на стару версію закінчаться та вирубити її. Так не буде жодного клієнту якого було від'єднано від сервера.

І ще помилки нормально оброблять, і конекти переподклювать. Не було просто фреймворку щоб транзитивні помилки автоматично вирішувати, а екпоненційний бекофф я не хотів самому писати.

А все інше вже не таке важливе напевно, чи я просто забув.

5. Висновок

В рамках цієї курсової роботи я обрав надзвичайно амбіційний та захоплюючий проект, який вимагав розробки на мові програмування, яка традиційно не використовується для таких завдань, що стало значним викликом. Цей процес дав мені можливість поглиблено вивчити різноманітні потужні патерни проектування, які стосуються не лише кодування, а й комплексної організації та управління системами.

У процесі створення месенджера для терміналу були розроблені серверні, клієнтські та допоміжні застосунки, що в сукупності забезпечують швидкий, безпечний та надійний обмін повідомленнями. Особливо важливим елементом стало адаптування паттерну Redux для клієнтського застосунку, що істотно спростило проектування складного інтерфейсу та функціоналу. Робота на низькому рівні програмування для створення ефективного клієнту була не тільки інтелектуально стимулюючою, але й приносила задоволення.

Серверна частина використовує RabbitMQ, потужний брокер повідомень, який є ключовим у розробці мікросервісної архітектури багатьма великими компаніями. Ця технологія дозволила максимально ефективно використовувати потенціал розподілених систем та мікросервісів для обміну повідомленнями.

Крім того, я набув цінного досвіду у написанні пайплайнів для розгортання додатків у хмарних сервісах, що є ключовою компетенцією сучасного розробника DevOps. Цей досвід безсумнівно стане в нагоді у моїй подальшій кар'єрі.

У підсумку, проект не тільки виконав поставлені завдання, але й перетворився на тріо високоякісних застосунків, які разом створюють унікальну та ефективну систему для обміну повідомленнями.

6. ВИКОРИСТАНА ЛІТЕРАТУРА

- [1] "Messer," [Online]. Available: <https://github.com/mjkaufner/Messer>.
- [2] "Plantuml," [Online]. Available: <https://plantuml.com/>.
- [3] "RabbitMQ," [Online]. Available: <https://rabbitmq.com/>.
- [4] "Auth0," [Online]. Available: <https://auth0.com/>.
- [5] "Kubernetes," [Online]. Available: <https://kubernetes.io/>.
- [6] "PostreSQL," [Online]. Available: <https://www.postgresql.org/>.
- [7] "Rust," [Online]. Available: <https://www.rust-lang.org/>.
- [8] "The register," [Online]. Available: https://www.theregister.com/2023/04/27/microsoft_windows_rust/.
- [9] "Rust Rover," [Online]. Available: <https://www.jetbrains.com/rust/>.
- [10] "Azure," [Online]. Available: <https://azure.microsoft.com/en-us>.
- [11] "Angular," [Online]. Available: <https://angular.io/>.
- [12] "NGRX," [Online]. Available: <https://ngrx.io/>.
- [13] "Go," [Online]. Available: <https://go.dev/>.
- [14] "tokio," [Online]. Available: <https://tokio.rs/>.
- [15] "gRPC," [Online]. Available: <https://grpc.io/>.
- [16] "SOLID," [Online]. Available: [https://simple.wikipedia.org/wiki/SOLID_\(object-oriented_design\)](https://simple.wikipedia.org/wiki/SOLID_(object-oriented_design)).
- [17] "Shaku," [Online]. Available: <https://crates.io/crates/shaku>.
- [18] "Enterprise Integration Patterns," [Online]. Available: <https://www.enterpriseintegrationpatterns.com/patterns/messaging/RequestReply.html>.
- [19] "Enterprise Integration Patterns," [Online]. Available: <https://www.enterpriseintegrationpatterns.com/patterns/messaging/CommandMessage.html>.
- [20] "Dead Letter Queue," [Online]. Available: <https://www.enterpriseintegrationpatterns.com/patterns/messaging/DeadLetterChannel.html>.

[21] "Device Authorization Flow," [Online]. Available: <https://auth0.com/docs/starteds/authentication-and-authorization-flow/device-authorization-flow>.

7. ДОДАТКИ

7.1. Дотаток А

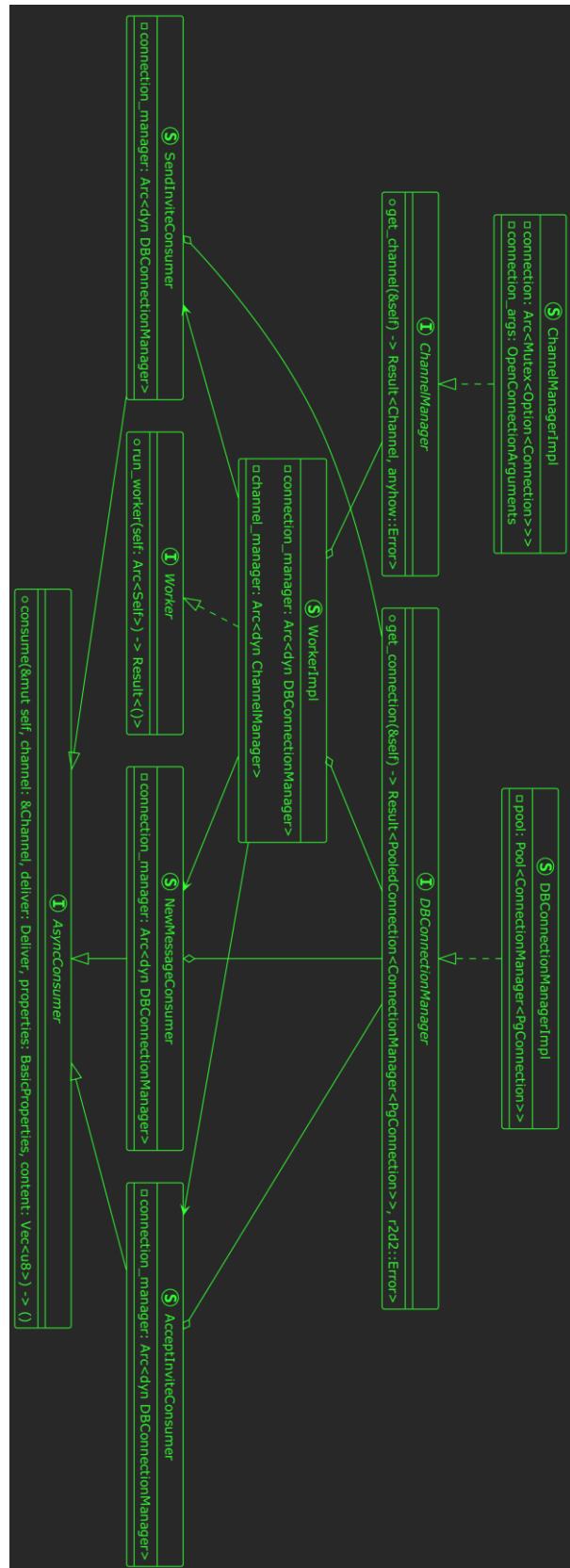


Рисунок 7.1 - Діаграма класів Воркера



Рисунок 7.2 - Частвова діаграма класів Сервера

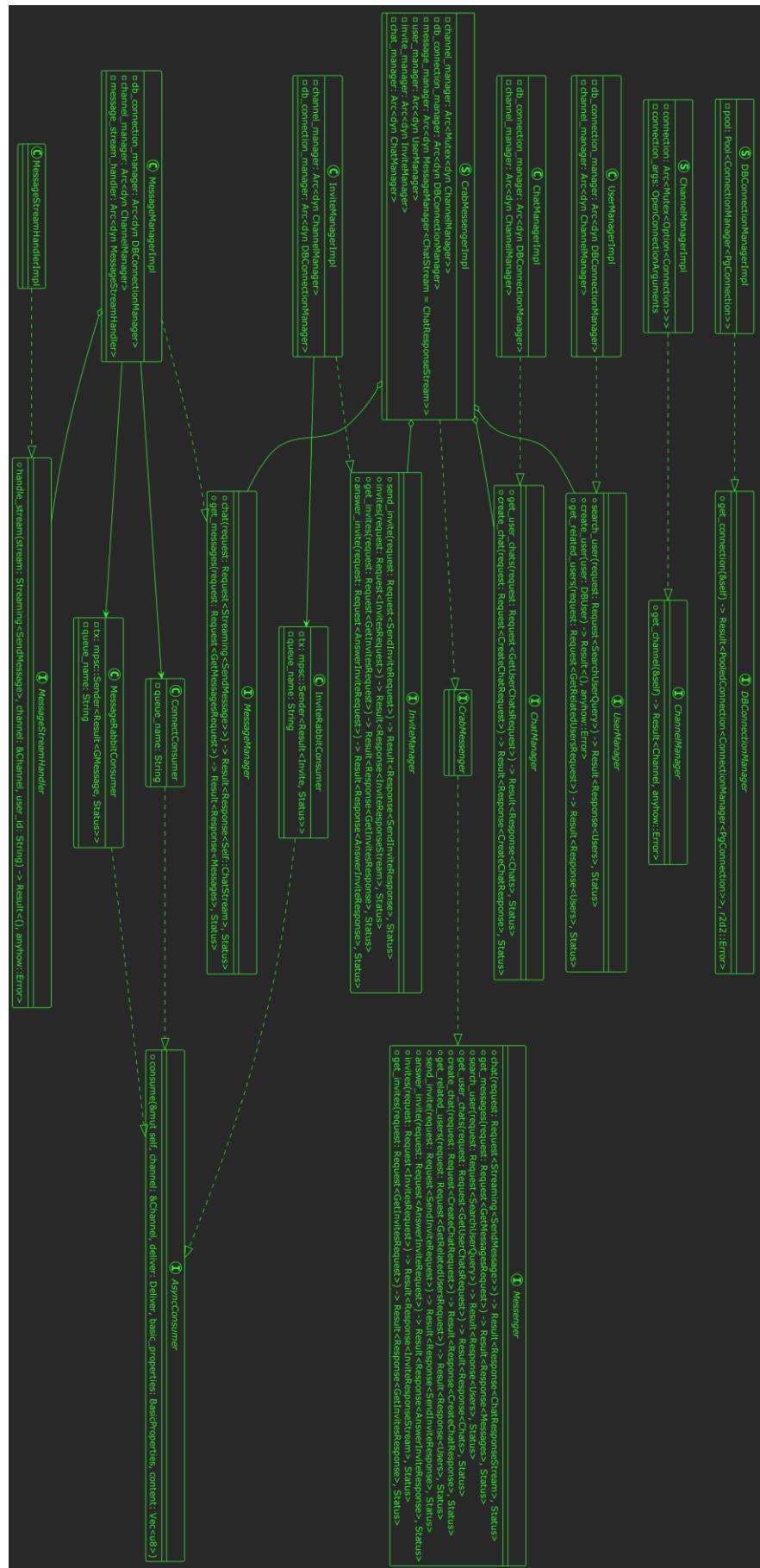


Рисунок 7.3 - Діаграма класів месенджера сервера

7.2. Додаток Б

```
#[tokio::main]
async fn main() -> Result<()> {
    let subscriber : Subscriber<EnvFilter> = fmt::Subscriber::builder()
        .with_env_filter(EnvFilter::from_default_env()) : SubscriberBuilder<...>
        .finish();

    tracing::subscriber::set_global_default(subscriber).expect( msg: "setting default subscriber failed");

    let module : Arc<WorkerModule> = build_worker_module();
    let worker: Arc<dyn Worker> = module.resolve();
    worker.run_worker().await?;
    Ok(())
}
```

Рисунок 7.4 - main воркера

На зображені відображено фрагмент вихідного коду з моого проекту, який демонструє ініціалізацію асинхронного воркера за допомогою Tokio. У коді налаштовується система логування з фільтром середовища. Після цього інстанціюється модуль воркера, який відповідає за робочі процеси в архітектурі сервера. Використання `Arc<dyn Worker>` дозволяє управляти спільними ресурсами між асинхронними задачами. Метод `run_worker()` асинхронно запускає основну бізнес-логіку воркера, що підтримує роботу сервера, обробляючи задачі в фоновому режимі.

Важливо відмітити, що інші застосунки будуть запущені за допомогою цього самого принципу, тому я не буду їх сюди додавати.

```

#[derive(Queryable, Selectable, Deserialize, Serialize, Insertable)]
#[diesel(table_name = crate::utils::persistence::schema::messages)]
#[diesel(check_for_backend(diesel::pg::Pg))]
pub struct Message {
    pub id: i32,
    pub text: String,
    pub created_at: chrono::NaiveDateTime,
    pub user_id: String,
    pub chat_id: i32,
}

7 usages  ✎ Michael Molchanov
#[derive(Queryable, Selectable, Deserialize, Serialize, Insertable)]
#[diesel(table_name = crate::utils::persistence::schema::messages)]
#[diesel(check_for_backend(diesel::pg::Pg))]
pub struct InsertMessage {
    pub text: String,
    pub user_id: String,
    pub chat_id: i32,
}

✎ Michael Molchanov *
impl From<ProtoMessage> for Message {
    ✎ Michael Molchanov *
    fn from(proto_msg: ProtoMessage) -> Self {
        let timestamp : Timestamp = proto_msg.created_at.unwrap();

        Message {
            id: proto_msg.id,
            user_id: proto_msg.user_id,
            chat_id: proto_msg.chat_id,
            text: proto_msg.text,
            created_at: chrono::NaiveDateTime::from_timestamp_opt(
                timestamp.seconds,
                timestamp.nanos as u32,
            )
            .unwrap(),
        }
    }
}

```

Рисунок 7.5 - Структура доступу до бази даних

На зображеному фрагменті коду представлена дві структури Rust, які взаємодіють з базою даних через ORM бібліотеку Diesel. Структура `Message` визначена як тип для відображення повних записів з таблиці повідомень, тоді як `InsertMessage` оптимізована для вставки нових записів без ідентифікатора, який, зазвичай, генерується базою даних.

Анотації `#[derive(Queryable, Selectable, Deserialize, Serialize, Insertable)]` надають структурам можливість серіалізації, десеріалізації та взаємодії з базою даних. Атрибут `#[diesel(table_name = ...)]` вказує Diesel на те, з якою таблицею пов'язана структура.

Реалізація `From<ProtoMessage> for Message` вказує на використання конвертеру, який дозволяє конвертувати дані з протокольного буфера (`ProtoMessage`) у внутрішню структуру домену (`Message`). Цей підхід демонструє гнучкість системи типів Rust і її ефективність у забезпеченні типобезпечної взаємодії між різними рівнями аплікації.

```

#[async_trait]
impl AsyncConsumer for RabbitConsumer {
    #[allow(unused)]
    #[cfg_attr(feature = "tracing", tracing::instrument(skip(self, channel, content)))]
    async fn consume(
        &mut self,
        channel: &Channel,
        deliver: Deliver,
        _: BasicProperties,
        content: Vec,
    ) {
        debug!("Sending message to user");
        let db_message: DBMessage = match serde_json::from_slice(&content) {
            Ok(msg : Message ) => msg,
            Err(e : Error ) => {
                error!("Failed to deserialize message: {:?}", e);
                return;
            }
        };

        let grpc_message : Message = db_message.into();

        let send_result : Result<(), SendError<Result<...>>> = self.tx.send(Ok(grpc_message)).await;
        debug!("Send result: {:?}", send_result);

        if let Err(e : Error ) = channel
            .basic_ack(BasicAckArguments::new(deliver.delivery_tag(), multiple: false))
            .await
        {
            error!("Failed to acknowledge message: {:?}", e);
        }
    }
}

```

Рисунок 7.6 - Надсилання повідомлень клієнту

Цей код є частиною асинхронного споживача RabbitConsumer, який реалізує інтерфейс AsyncConsumer. Він відповідає за прийняття повідомлень з RabbitMQ exchange та подальшу їх обробку. Функція consume асинхронно читає вміст повідомлення, десеріалізує його з JSON формату у структуру DBMessage, перетворює у формат, придатний для gRPC (за допомогою into()), та надсилає його в канал передачі повідомлень. У випадку успішної операції відбувається підтвердження обробки повідомлення. Цей механізм є критично важливим для доставки повідомлень користувачеві.

```

// Michael Molchanov
#[async_trait]
impl AsyncConsumer for ConnectConsumer {
    // Michael Molchanov
    async fn consume(
        &mut self,
        channel: &Channel,
        deliver: Deliver,
        basic_properties: BasicProperties,
        content: Vec<u8>,
    ) {
        let user_id : String = String::from_utf8(content).unwrap();

        if let Err(e : ()) = declare_messages_exchange(channel, &user_id)
            .await
            .map_err(|e : Error | {
                error!("Failed to declare exchange: {:?}", e);
            })
        {
            let _ = channel
                .basic_reject(BasicRejectArguments::new(deliver.delivery_tag(), requeue: false))
                .await
                .map_err(|e : Error | {
                    error!("Failed to reject message: {:?}", e);
                });
            return;
        }

        if let Err(e : ()) = channel
            .queue_bind(QueueBindArguments::new(&self.queue_name, &messages_exchange_name(&user_id), routing_key: ""))
            .await
            .map_err(|e : Error | {
                error!("Failed to bind queue: {:?}", e);
            })
        {
            let _ = channel
                .basic_reject(BasicRejectArguments::new(deliver.delivery_tag(), requeue: false))
                .await
                .map_err(|e : Error | {
                    error!("Failed to reject message: {:?}", e);
                });
        }

        if let Err(e : Error) = channel
            .basic_ack(BasicAckArguments::new(deliver.delivery_tag(), multiple: false))
            .await
        {
            error!("Failed to acknowledge message: {:?}", e);
        }
    }
}

```

Рисунок 7.7 - Під'єднання черги клієнта до нового чату

Ця асинхронна функція consume реалізована для ConnectConsumer і відповідає за процес під'єднання черги користувача до нового чату в месенджері на базі RabbitMQ. Споживач отримує ідентифікатор користувача з вмісту повідомлення, оголошує новий exchange і прив'язує чергу користувача до цього exchange. Якщо процес зазнає невдачі на будь-якому етапі, повідомлення відхилено.

```

fn process(&self, handle: Handle) -> anyhow::Result<()> {
    let dispatch_rc : Receiver<Action> = self.dispatch_rc.clone();
    let app_reducer : Arc<dyn AppReducer> = self.app_reducer.clone();
    let dispatch_tx : Sender<Action> = self.dispatch_tx.clone();
    let select_tx : Sender<State> = self.select_tx.clone();
    self.select_tx.send(self.state.clone())?;
    let mut state : State = self.state.clone();
    thread::spawn(move || {
        while let Ok(action : Action) = dispatch_rc.recv() {
            let reduce_result : ReduceResult =
                app_reducer.reduce(&action, &state.clone(), dispatch_tx.clone(), handle.clone());

            if let ReduceResult::Consumed(new_state : State) = reduce_result {
                state = new_state.clone();
            }

            select_tx.send(state.clone()).unwrap();
        }
    });
    Ok(())
}

```

Рисунок 7.8 - Сердце патерну Redux

Ця функція `process` є ключовою частиною імплементації модифікованого паттерну Redux для моого проекту. Вона запускає окремий потік, який слухає дій (actions), отримані через канал `dispatch_rc`. Кожна дія передається до функції `reduce` разом із поточним станом `state`. Якщо результатом виклику `reduce` є новий стан, цей стан заміщує попередній і розсилається всім зацікавленим компонентам через канал `select_tx`. Такий підхід забезпечує централізоване управління станом додатку і реактивне оновлення стану відповідно до користувачьких дій.