# РЕФЕРАТ

Пояснювальна записка до курсового проекта на тему «Месенджер» складається з трьох розділів, висновків, списку використаної літератури та додатку.

Текстова частина викладена на 54 сторінках друкованого тексту.

Пояснювальна записка має 81 сторінку додатків. Список використаних джерел містить 11 найменувань і займає 1 сторінку. В роботі наведено 33 рисунка. Загальний обсяг роботи – 136 сторінок.

У першому розділі було обґрунтувано створення програми для обміну повідомленнями між користувачами.

У другому розділі проведено проектування і розробка програмнго продукту.

У третьому розділі проведено тестування програмнго продукту.

Висновок містить в собі результати виконаної роботи створення програми месенджер за допомогою веб-технологій.

У додатку представлений лістинг розробленого програмного продукту.

Ключові слова: КЛІЄНТ, СЕРВЕР, ЮЗЕР, СОКЕТ, БАЗА ДАНИХ, ХЕШУВАННЯ, ОБМІН ІНФОРМАЦІЄЮ, АСИНХРОННІСТЬ, ПОТІК.

ЗМІСТ

[ВСТУП 3](#_Toc10087733)

[РОЗДІЛ 1 АНАЛІЗ ПРОБЛЕМАТИКИ, МЕТОДІВ ТА ЗАСОБІВ ВИРІШЕННЯ ЗАДАЧІ 4](#_Toc10087734)

[1.1 Аналіз задачі, засобів та методів її вирішення 4](#_Toc10087735)

[1.2 Аналіз існуючого програмного забезпечення за тематикою курсового проекту 6](#_Toc10087736)

[1.3 Технічне завдання на курсовий проект 9](#_Toc10087737)

[РОЗДІЛ 2 ПРОЕКТУВАННЯ ТА РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 12](#_Toc10087738)

[2.1 Проектування загального алгоритму роботи програми 12](#_Toc10087739)

[2.2 Розробка функціональних алгоритмів роботи програми 15](#_Toc10087740)

[2.3 Розробка програмного забезпечення 16](#_Toc10087741)

[РОЗДІЛ 3 ОПИС РОБОТИ З ПРОГРАМНИМ ДОДАТКОМ ТА ЙОГО ТЕСТУВАННЯ 46](#_Toc10087742)

[3.1 Опис роботи з програмним додатком 46](#_Toc10087743)

[3.2 Тестування роботи програмного забезпечення 47](#_Toc10087744)

[ВИСНОВКИ 54](#_Toc10087745)

[СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 55](#_Toc10087746)

[ДОДАТКИ 56](#_Toc10087747)

# ВСТУП

**Головна мета:** проаналізувати поставлене, згідно з варіантом, завдання. Віднайти, оглянути та порівняти аналогічні розробки. Спроектувати структуру та на основі здобутих за другий семестр навчання знань реалізувати програмний додаток.

**Актуальність теми:** в цьому курсовому проекті будемо використовувати методологію ООП, а саме попрацюємо з класами, об’єктами класів. На основі об’єктів будемо нарощувати програмний функціонал. Про що обов’язково потрібно сказати: ні функціональні мови програмування, ні те ж ООП, не є чимось новим, так само як і багато інших «нові підходи», знову стали популярними останні роки. Багато з них беруть початок в 60-х роках, коли програмістів було мало, і прийшли вони, в основному, з середовища математиків і фізиків. Тоді було легко винаходити нове, і це швидко підхоплювалося допитливими умами, розвивалося і застосовувалося. Зараз кількість програмістів значно зросла, і як наслідок, рівень середнього програміста став значно нижче. Саме тому, що вже завоювала популярність, нескладна концепція об'єктного проектування як і раніше залишається такою поширеною. Кілька слів на захист ООП, по-перше, це працює. Так чи інакше, програмісти навчилися розуміти один-одного, будувати складні системи і створювати абстракції за допомогою патернів. Безліч коду написано і протестовано. Це те, що робить ООП актуальним ще довгий час, особливо для вирішення класичних задач. Але якщо ми говоримо про тривіальні завдання, потрібно постійно застосовувати нові підходи. Більш того розуміння підходів функціонального програмування, алгоритмів, пристрої компіляторів-інтерпретаторів дають програмісту незаперечні переваги для написання якісного і гнучкого коду.

**Об’єкт дослідження:** алгоритми підключення декількох користувачів до сервера з можливістю передачі даних.

**Предмет дослідження:** використання веб-технологій для підключення та обміну повідомленнями між клієнтами та сервером.

# РОЗДІЛ 1 АНАЛІЗ ПРОБЛЕМАТИКИ, МЕТОДІВ ТА ЗАСОБІВ ВИРІШЕННЯ ЗАДАЧІ

## Аналіз задачі, засобів та методів її вирішення

Месенджер – програма для миттєвого обміну повідомленнями (в тому числі файлами, зображеннями, голосовими повідомленнями та іноді дзвінками). Це означає, що потрібно будь-яким чином з’єднати користувачів   
між собою в одній системі.

Є декілька методів реалізації цієї ідеї. Перша це зв’язати користувачів на пряму, без використання сервера, але думку це не коректне рішення, і тут краще підійде другий метод – зв’язування користувачів за допомогою сервера. У цьому випадку всі дії з боку клієнтів (такі як реєстрація, надсилання повідомлень і т.д.) будуть оброблюватися через сервер (наприклад стаціонарний комп’ютер). Кожен користувач при реєстрації повинен буде вказати логін, свою пошту, яку ми будемо використовувати для ідентифікації, та пароль.

Якщо реєстрація проходить успішно, дані разом з захешованим паролем потрапляють на сервер, де зберігаються в SQLITE – це бібліотека мови С, яка реалізує невеликий, швидкий, автономний, високонадійний, повнофункціональний движок бази даних SQL. SQLite є найбільш часто використовуваним механізмом бази даних у світі. SQLite вбудований у всі мобільні телефони та більшість комп'ютерів і поставляється в безлічі інших програм, які люди використовують щодня. У цьому випадку доступ до повідомлень можна буде дістати з будь-якого пристрою (на якому працює наш месенджер), просто ввівши логін та пароль.

З’єднання між сервером та клієнтами буде відбуватися за технологією Ethernet та протоколом TCP/IP. TCP/IP – це стек протоколів. TCP (на відміну від UDP) – відповідає за однозначну передачу пакетів даних в місце призначення. Прикладом використання цього протоколу є електронна пошта, адже якщо відправляємо лист, то потрібно щоб він в повному обсязі був доставлений потрібному користувачу. UDP використовують в онлайн іграх, бо там потрібна швидкість передачі, якщо декілька пакетів не приходить нічого страшного не станеться, ми просто втратимо декілька кадрів. IP – це протокол адресації пристроїв.

Навіть якщо зловмисник зможе дістати доступ до акаунтів, то йому важко буде прив’язати повідомлення до будь-якої людини (звісно якщо користувач не використав реальне ім’я, як логін), адже ми не використовували номера для реєстрації акаунтів. Багато відомих месенджерів мають цей недолік. За допомогою номерів вони збирають інформацію про користувачів, можливо для таргетованої реклами, і додають можливість отримати повідомлення, якщо хтось з вашої телефонної книги зареєструвався в месенджері.

Для написання нашої програми ми будемо використовувати мову програмування C# та WPF (Windows Presentation Foundation), тому що ця технологія дає широкі можливості. Розроблена мова розмітки XAML (Extensible Application Markup Language) дозволяє відділити програмний код від інтерфейсу, що збереже купу часу, коли буде потрібно змінити вигляд програми, або навіть при перенесенні месенджера на іншу платформу. Порівняно з Windows Forms, WPF підтримує DirectX. Це означає, що стало можливим створення складних графічних елементів з тінями та накладанням шарів один на одного, а також це дає приріст в продуктивності.

Для того щоб ефективно розробляти наш програмний додаток поділимо весь процес на частини. Це дозволить організувати роботу, та в будь-який час розуміти на якому етапі ми знаходимося. Створення поділимо на такі етапи:

* Аналіз роботи програми, функцій які будуть присутні в програмі;
* Розробка структури коду (побудова діаграм та блок-схем до кожної; функції з попереднього пункту). Розробка класів та схем наслідування (у тому числі і для серверної частини);
* Створення дизайну програми в графічному редакторі;
* Написання основного коду та перевірка функціонування;
* Написання графічної частини програми WPF з використанням XAML.
* Налагодження коду;

## 1.2 Аналіз існуючого програмного забезпечення за тематикою курсового проекту

Для порівняння будемо використовувати три популярних месенджера: Telegram, Facebook Messenger, Viber.

Почнемо з Telegram – це кроссплатформенна програма для обміну повідомленнями та файлами створена Павлом Дуровим у 2013 р. Зв’язок між користувачами здійснюється через сервер. Усі повідомлення кодуються за допомогою криптографічного протоколу MTProto. Він включає в себе низку інших алгоритмів кодування, хеш-функцій. В системі використовується алгоритм кодування RSA (абревіатура від фамілій Rivest, Shamir і Adleman) , в якому за допомогою певних операцій над числами створюються публічні та приватні ключі. Повідомлення потрапляють на сервер а потім до потрібного користувача де текст декодується.

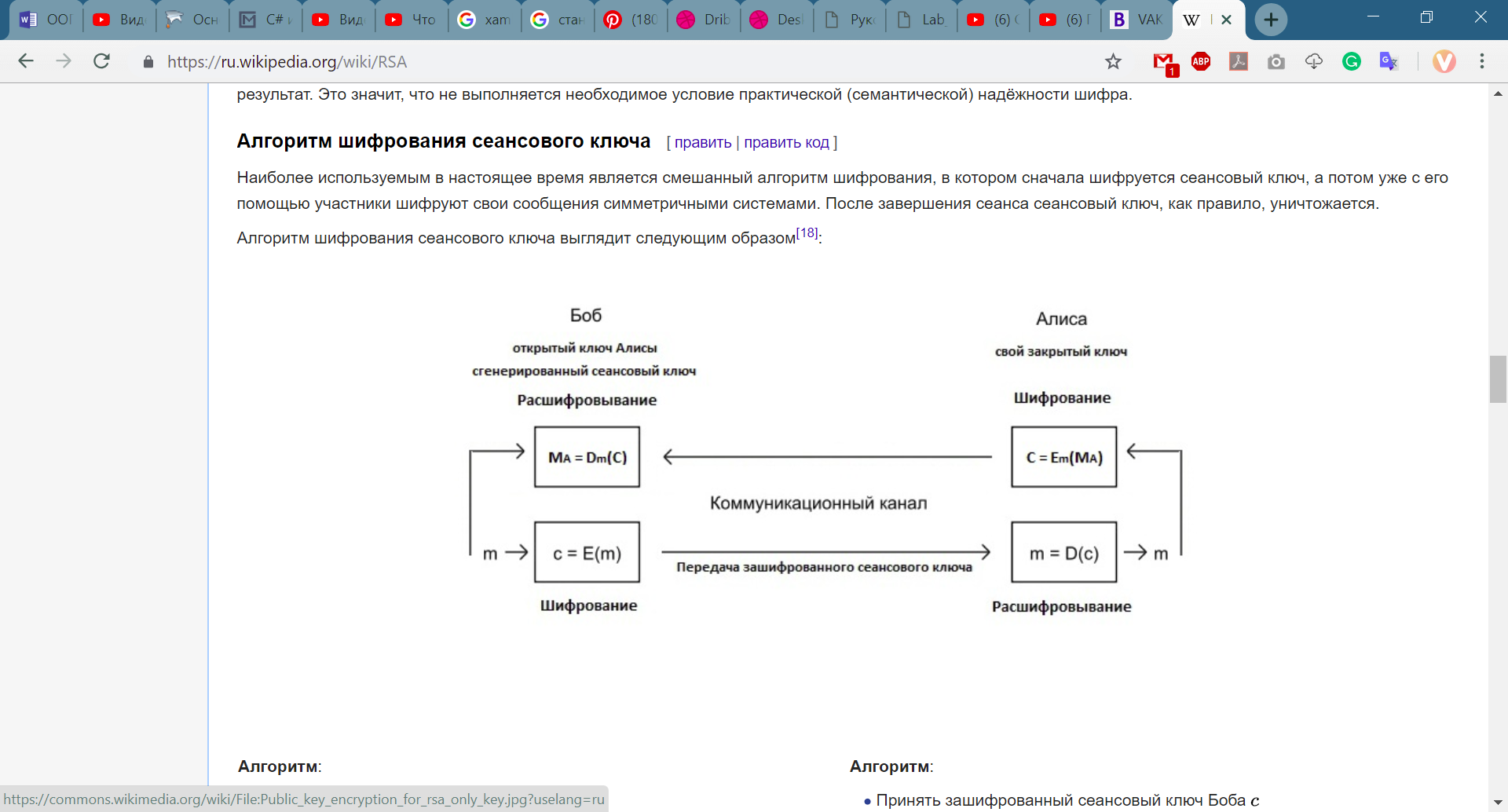


Рис. 1.1. Алгоритм кодування за допомогою сеансного ключа

Вся функціональність в Telegram розділена на вкладки. Кожна вкладка оформляється у вигляді чату. У Telegram є 5 видів таких чатів:

* Диалоги;
* Групи;
* Збереженні повідомлення;
* Канали;
* Чати з ботами

Діалоги:

Оформлення і функціональність діалогів не надто відрізняється від інших месенджерів. Присутні стандартні можливості: голосові повідомлення, відео повідомлення, прикріплення файлів, стікери, gif-анімації і емодзі, можливість побачити, що співрозмовник прочитав повідомлення, попередній перегляд посилань і т. д.

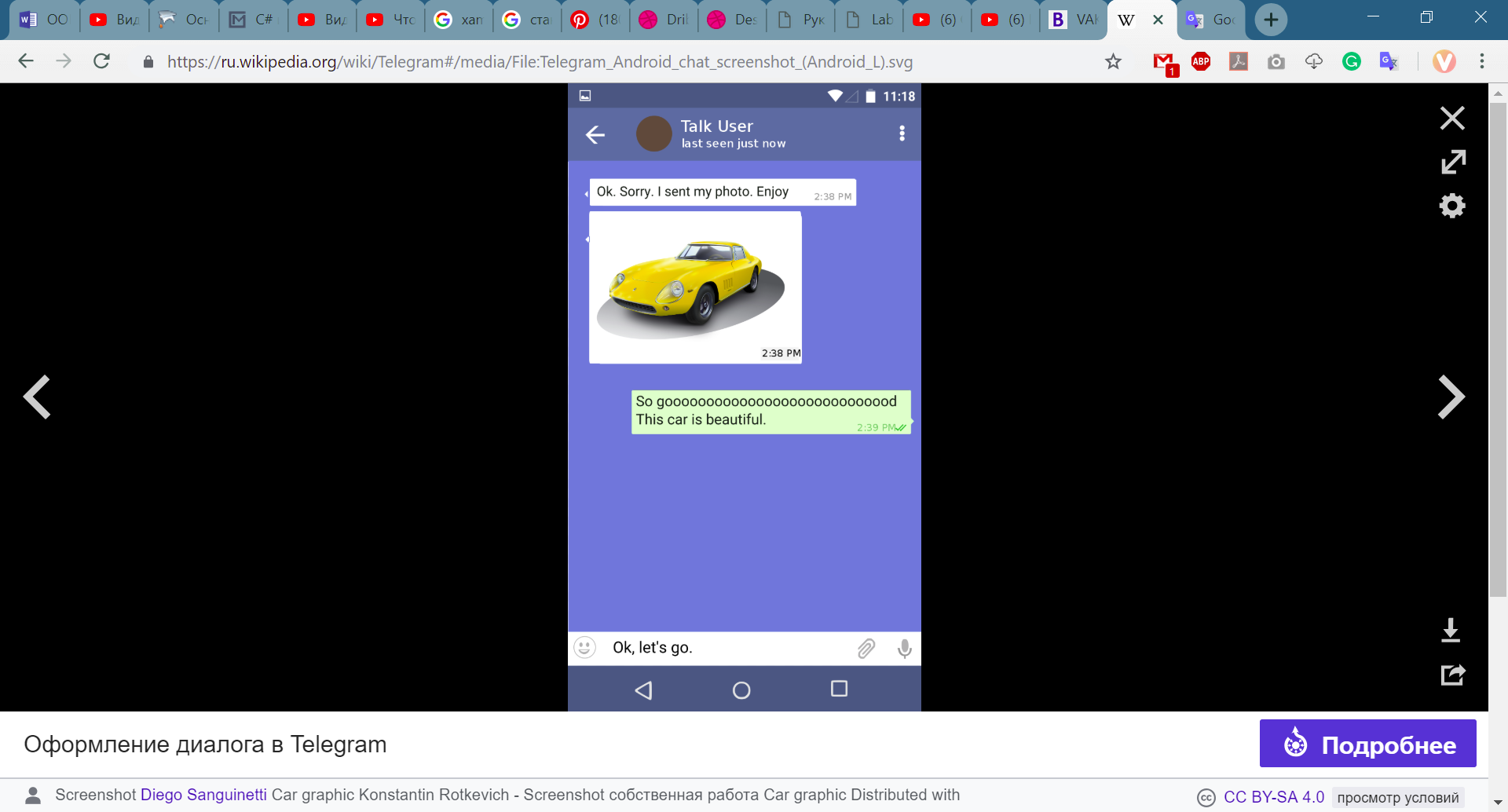


Рис. 1‑2 Оформлення діалогу в Telegram

Групи:

Є можливість організовувати групи до 200 учасників, починаючи з листопада 2015 року, супергрупи до 1000 учасників, з 14 березня 2016 - супергрупи до 5000 учасників. З 30 червня 2017 року розмір супергруп збільшився до 10000 учасників, з 30 січня 2018 року - супергрупи до 100000 учасників.

Збереження повідомлень:

Всі потрібні програми можна зберегти в окрему вкладку. Також туди можна завантажити необмежену кількість файлів, тобто месенджер надає нескінченне хмарне сховище.

Канали:

Найважливішою особливістю, що відрізняє Telegram від своїх конкурентів, можна вважати інструмент комунікації в форматі публічних каналів. Такий спосіб дозволяє автору або групі авторів ділитися інформацією з необмеженим колом осіб з мінімальною дистанцією між читачем і контентом, але і зберігати при цьому анонімність.

Telegram-канали мають три ключових відмінності від стандартних мікро блогів (таких, як Twitter, Facebook, Tumblr ...):

* відсутність алгоритмічної стрічки новин;
* відсутність зворотного зв'язку;
* анонімність.

Viber - додаток для смартфонів, що дозволяє здійснювати відео і голосові VoIP дзвінки. Здійснювати дзвінки між користувачами з встановленим Viber безкоштовні (оплачується тільки інтернет-трафік за тарифом оператора зв'язку). Також в Viber є можливість передачі текстових повідомлень, зображень, відео- та аудіо повідомлень, документів і файлів. Для авторизації користувачів і пошуку контактів додаток використовує номер телефону і передає вміст телефонної адресної книги (імена і телефони всіх контактів) на сервери корпорації Viber Media

На відміну від того ж Telegram в Viber всі діалоги між користувачами зберігаються на пристрої (телефон, комп’ютер і т. д.). Це дуже корисний хід, якщо сервера мають обмежену кількість пам’яті. З’явилася функція збереження резервної копії в хмарне сховище.

Робота чатів та груп аналогічна до Telegram.

Facebook Messenger - додаток для обміну миттєвими повідомленнями і відео, створене Facebook. Він інтегрований з системою обміну повідомленнями на основному сайті Facebook (Facebook Chat) і побудовано на базі відкритого протоколу MQTT, який працює на базі TCP/IP, орієнтований для обміну повідомленнями між пристроями за принципом видавець-передплатник.

В усьому іншому Facebook Messenger подібний до Telegram та Viber

На мою думку ці месенджери мають один мінус. При реєстрації потрібно вводити діючий номер телефона, для якого приходить SMS повідомлення, за які платить компанія. Це шкодить анонімності, адже за номером телефона зловмисник може індитифікувати людину, а також це не найкращій шлях витрачати гроші.

Таблиця 1.1

**Порівняння аналогів**

|  |  |  |  |
| --- | --- | --- | --- |
| **Назва додатку** | Безпека даних | Функціональність | Доступ офлайн |
| Telegram | + | + | + |
| Viber Messenger | +- | +- | +- |
| Facebook Messenger | +- | +- | + |

Висновок до порівняльної таблиці: з вищезазначених месенджерів Telegram має найбільший функціонал. Окрім звичайних груп та чатів він має канали, секретні чати, а також можливість створювати своїх ботів, вже не кажучи про те, що кожному користувачу надається безмежне хмарне сховище. Це швидкий, стильний, зручний та безпечний месенджер.

## 1.3 Технічне завдання на курсовий проект

Введення

Ми живимо в 21-му столітті, інформаційній ері, коли обсяги інформації настільки великі, що потрібно вчасно реагувати. Виникає велика потреба в швидкому комунікуванні з іншими людьми. Для вирішення цих проблем було придумано багато пристроїв. Найрозповсюдженіший з них є телефон. Кожен день ми використовуємо його для зв’язку з іншими людьми, але чи правильно ми це робимо? На мою думку використання PSTN або GSM є застарілим. На це є декілька причин:

1. Це вичерпний ресурс, кількість номерів, виділених для кожної країни, обмежена.

2. Це регульований ресурс - видається тільки державними уповноваженими організаціями, причому в кожній державі свої правила.

3. Так як номери з геоприв’язкой до певної країни, місту і т.д., то для дзвінків на ці номери потрібно платити гроші.

Тому є сенс розробляти такі програмні додатки, з допомогою яких ми зможемо безкоштовно обмінюватися повідомленнями, дзвонити.

Найменування та область застосування:

Месенджер - спосіб миттєвої комунікації. З його допомогою ми можемо зв’язатися з людиною, яка знаходиться в будь-якому місці безкоштовно, з урахуванням присутності інтернет з’єднання.

Інший спосіб використання месенджерів – застосування для бізнесу:

* сфера послуг (бронювання авіаквитків, виклик таксі, запис або консультація у лікаря);
* фінансовий сектор (здійснення грошових переказів);
* e-commerce (підтвердження доставки, клієнтська підтримка 24/7).

Підстава для розробки:

Розробка проводиться на основі завдання курсового проекту за спеціальністю 121 «Інженерія програмного забезпечення» Житомирського державного технологічного університету.

Призначення розробки:

Функціональним призначенням програмного додатку є миттєвий обмін повідомленнями між користувачами мережі.

Технічні вимоги до програми чи програмного продукту

Завершений програмний продукт повинен володіти такими функціональними можливостями:

* Реєстрація нових користувачів (збереження даних в базу даних);
* Вхід користувачів в систему;
* Синхронізація даних з сервером;
* Обмін повідомленнями, у тому числі і файлами або зображеннями, з зареєстрованими користувачами;
* Створення публічних та приватних груп з можливостями додавання, видалення користувачів.

Стадії й етапи розробки:

* Аналіз роботи програми, функцій які будуть присутні в програмі;
* Розробка структури коду (побудова діаграм та блок-схем до кожної; функції з попереднього пункту). Розробка класів та схем наслідування (у тому числі і для серверної частини);
* Створення дизайну програми в графічному редакторі;
* Написання основного коду та перевірка функціонування;
* Написання графічної частини програми WPF з використанням XAML.
* Налагодження коду;

# РОЗДІЛ 2 ПРОЕКТУВАННЯ ТА РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

## 2.1 Проектування загального алгоритму роботи програми

Розробку загальної структури програми ми розпочнемо з «найвищої», найбільш абстрактної точки. Весь процес ми поділимо на два етапи: розробка клієнтської та серверної частини.

Сервер:

Ми знаємо, що потрібно створити сервер, що буде зв’язною точкою між усіма клієнтами. Усі відправлені команди будуть потрапляти на сервер, після чого оброблятися, а всі зміни, що відбуваються в базі даних надсилатимуться усім юзерам, які в даний момент під’єднані до сервера. З цього випливає, що сервер повинен одночасно підтримувати зв’язок з деякою кількістю клієнтів. Крім того, потрібно мати можливість в цей момент відправляти команди клієнтам.

Це означає що потрібно буде використовувати потоки в нашій програмі, тобто розробляти нашу програму багатопоточною. Потік виконання (тред; від англ. thread — нитка) – це найменша одиниця обробки, виконання якої може бути назначене ядром операційної системи. Кілька потоків виконання можуть існувати в рамках одного і того ж процесу і спільно використовувати ресурси, такі як пам'ять, тоді як процеси не поділяють цих ресурсів. На одному процесорі багатопоточність зазвичай відбувається шляхом тимчасового мультиплексування процесор перемикається між різними потоками виконання. Це дозволить на кожний важливий процес, такий як очікування з’єднання та надсилання команд з клієнта, виділити окремий потік. Для з’єднання, отримання та надсилання інформації клієнтам ми створимо окремий клас, який буде використовувати сокети для виконання поставлених завдань. Сокети (англ. socket - заглиблення, гніздо, роз'єм) — назва програмного інтерфейсу для забезпечення обміну даними між процесами. Процеси при такому обміні можуть виконуватися як на одній ЕОМ, так і на різних ЕОМ, пов'язаних між собою мережею. Сокет - абстрактний об'єкт, що представляє кінцеву точку з'єднання. В C# реалізація цього об’єкта знаходиться в просторі імен System.Net.Sockets. Сокети дозволять достатньо швидко налаштувати систему для обміну інформацією по мережі. Після того, як ми під’єднаємо та отримаємо інформацію для обробки, вона попаде в чергу, яка потрібна для того, щоб ми могли накопичувати інформацію, без ризику втрати. Далі інформація буде находити в основний модуль для обробки, де буде відбуватися робота з базою даних. Після чого буде надсилатися інформація усім клієнтам, якщо це звісно потрібно. Графічна система сервера буде представляти собою, вікно з великим текстбоксом та двома кнопками для виведення інформації роботи сервера та ввімкнення, вимкнення серверу. Сервер буде збирати інформацію усіх клієнтів, а при підключенні юзера, його дані будуть оновлюватися. На цьому функціонал серверу закінчується.

Клієнт:

Клієнт – основна програма, яка буде автоматично з’єднуватися з сервером та надавати можливості обміну повідомленнями. З’єднання з сервером буде відбуватися за допомогою класу, який нагадує схожий клас сервера, Він буде здійснювати підключення та надсилання та прийом інформації від серверної частини. Окрім основних функцій підключення, прийому та відправлення данийх, клас буде сигналізувати о статусі підключення, який ми будемо використовувати у майбутньому.

Після рівня, що відповідає за підключення до серверу, йде клас, який буде надавати можливості для додавання інформації, яка потім надсилатиметься на сервер, в чергу, а також отримання інформації, яка надійшла з серверу. У цьому класі доречно буде використати події, наприклад для сигналізації отримання нової порції даних від сервера. В ООП події грають значну роль, адже час від часу потрібно якимось чином передати інформацію з одного класу в інший, ви скажете, що це можна зробити і без системи сповіщення. Так можна, але з часом ваш код стане важким, не тільки для читання та перевірки, а й для заміни окремих модулів на інші. Вам потрібно буде заміняти усі виклики одних класів на інші, а це вже протирічить основним принципам ООП, а саме **SOLID.** Замість того щоб плутати свою програму краще використати події, тому що навіть при потребі замінити деякі класи на інші, все що вам потрібно буде зробити, це встановити нового підписника події.

Далі переходимо до основного класу, який буде виконувати роль керівника, направляючого. Саме він матиме інформацію о інших модулях програми. Якщо брати конкретний паттерн проектування, то для опису усієї програми клієнтської частини та ролі цього класу, краще всього підійде **Mediator**. Це паттерн який використовується в модульному проектуванні, для заміни зв'язків усі до усіх, на один до усіх. По принципу своєї роботи він нагадує башту, яка координує літаки у певному аеропорті. Наш клас буде знади де знаходиться потрібна інформація, і як її дістати.

Наступним йде клас відповідальний за доступ до локальної бази даних. В ньому будуть зберігатися класи, які описують таблиці бази даних, і з функціоналу мають головні методи, які можуть знадобитися для запису, видалення та пошуку інформації.

Останній шар являє собою представленням інтерфейсу користувача або UI Presenter. Так як ми будемо використовувати WPF, то тут ми будемо зберігати файли з XAML кодом, а саме: сторінки, юзер контроли, стилі та ін. Крім того тут ми будемо зберігати моделі представлення, так званні View Models. Це об’єкти які зберігають властивості та команди, до яких ми потім зможемо прив’язати графічне представлення. Цей паттерн програмування називають **MVVM** (Model View View Model). Перевага цього паттерна у порівнянні зі звичайним накопиченням властивостей та методів у класі зв’язаного коду (наприклад файл App.xaml.cs) дуже велика. По-перше, view model це клас, який нічого не знає о файлах з xaml кодом, і при зміні як самого xaml коду так і платформи розробки (наприклад на Xamarin для IOS і Android) головну логіку змінювати не знадобиться. Ми проводимо чітку грань між графічною частиною та бізнес логікою. По-друге, використовуючи цей паттерн стає набагато простіше писати юніт тести.

## 2.2 Розробка функціональних алгоритмів роботи програми

Спочатку наведемо загальну схему усіх модулів програми (включаючи сервер), далі будемо описувати кожен шар діаграми окремо.

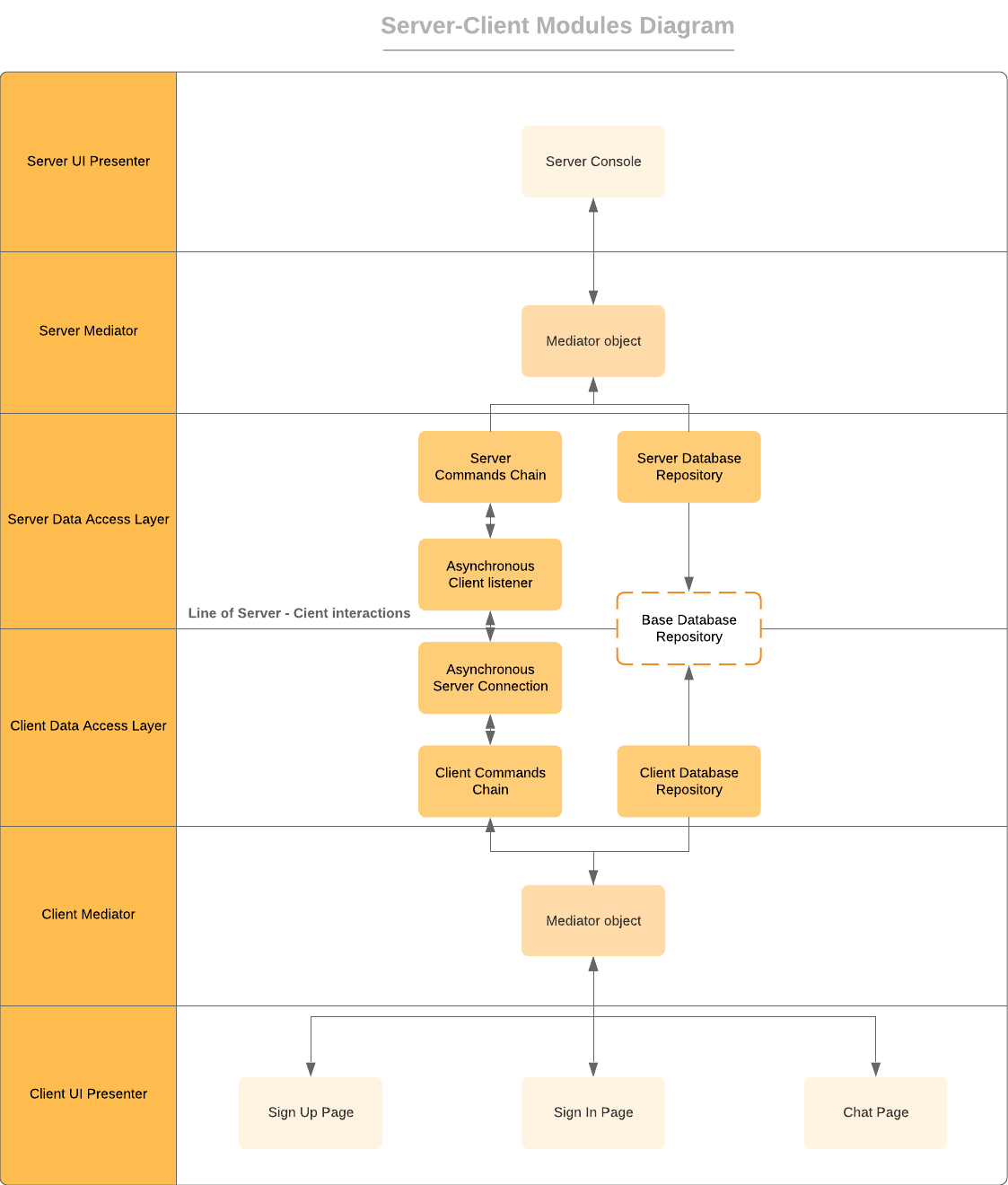


Рис. 2.1. Діаграма модулів клієнт-серверного додатку

По середині ми маємо лінію перетину, яка об’єднує серверну частину з клієнтською.

На діаграмі (рис. 2.1 – Client Data Access Layer, Server Data Access Layer) можна побачити репозиторій бази даних сервера та клієнта пов’язані з базовою абстракцією бази даних, яка інкапсулює в собі базові дії над таблицею. Вона знаходиться по середині серверної та клієнтської частини, тому що буде винесене в окрему бібліотеку спільних класів.

Далі ми можемо побачити два класи, які підтримують зв’язок між двома частинами, а саме: асинхронний прослуховувач клієнтів та асинхронне під’єднання до сервера. Вони тримають з’єднання в окремому потоці, тому не блокують основний. При отриманні команд вони передають їх у ланцюг команд. На цій діаграмі не видно але ланцюг команд клієнта не схожий на ланцюг команд сервера, так як в клієнта розташовано дві черги: для команд, які потрібно відправити на сервер, та команд, які прийшли з серверу. В сервера знаходиться лише одна черга: команд, які надходять з клієнта. Такі відмінності пов’язані з тим, що сервер при отриманні команди, миттєво починає обробляти її після чого розсилає відповіді, коли в клієнта команди, що приходять від сервера можуть чекати деякий час перед тим, як оброблятися, тому що локальна база даних в даний момент може бути зайнята іншим потоком програми. Разом усі ці класи утворюють модуль доступу до даних.

Наступними ідуть шари управління доступу до модулів (рис. 2.1 – Client Mediator, Server Mediator). В сервера та клієнта вони майже однакові і складаються з одного класу, до якого ми звертаємося, якщо знадобиться інформація, наприклад з локальної бази даних.

Останніми йдуть шари представлення інтерфейсів клієнта та сервера (рис. 2.1 – Client UI Presenter, Server UI Presenter). У сервера є одне головне вікно на якому буде відображатися інформація про поточні дії: підключення нового юзера, відправлення та прийняття інформації від клієнтів.

## 2.3 Розробка програмного забезпечення

Клієнт:

Загальна діаграма класів клієнтського додатку складається з трьох частин. А саме з бібліотек Common Lib, Client Lib та проекту WPF (Client UI Presenter)

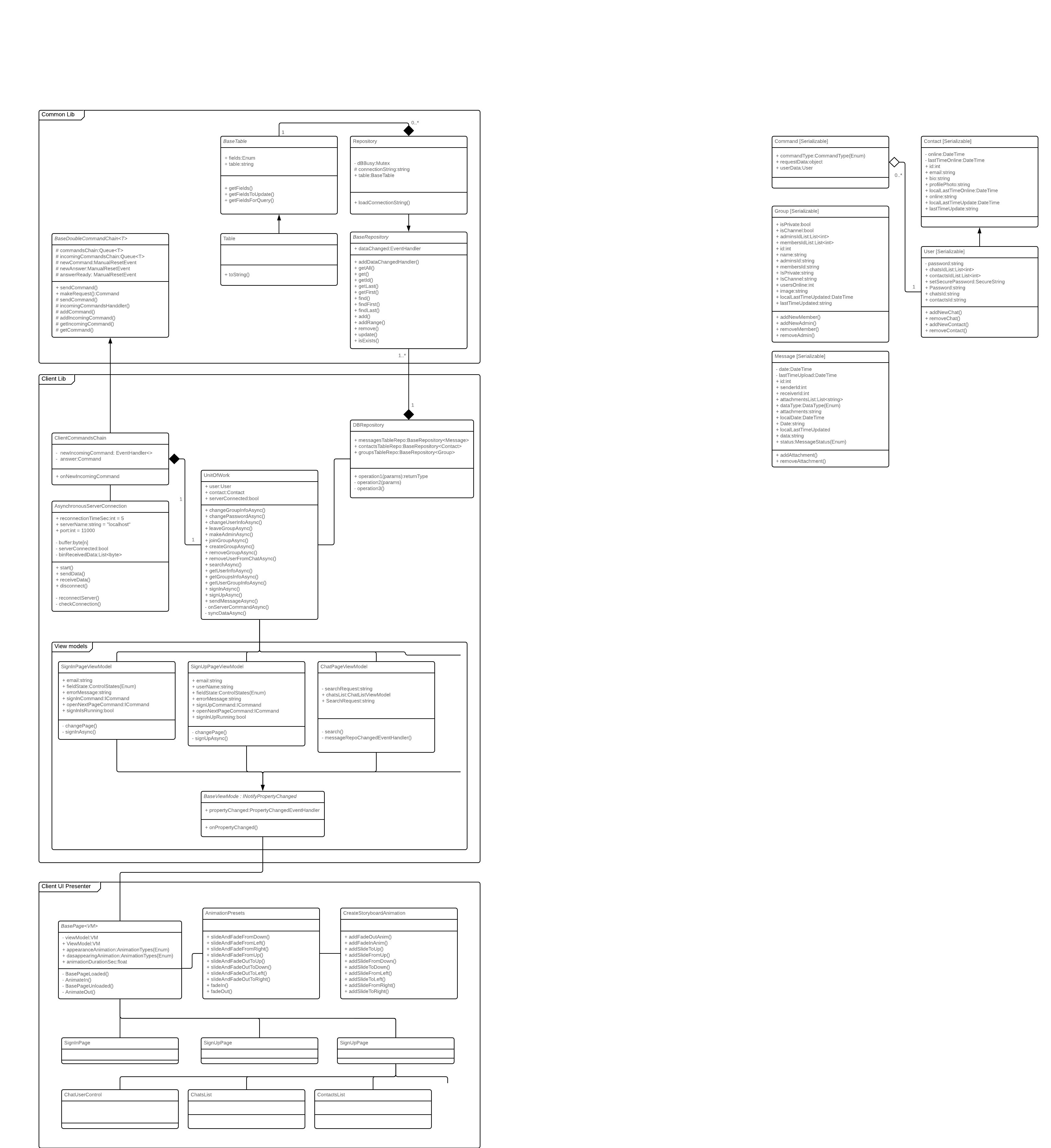


Рис. 2.2. Діаграма класів клієнтського додатку

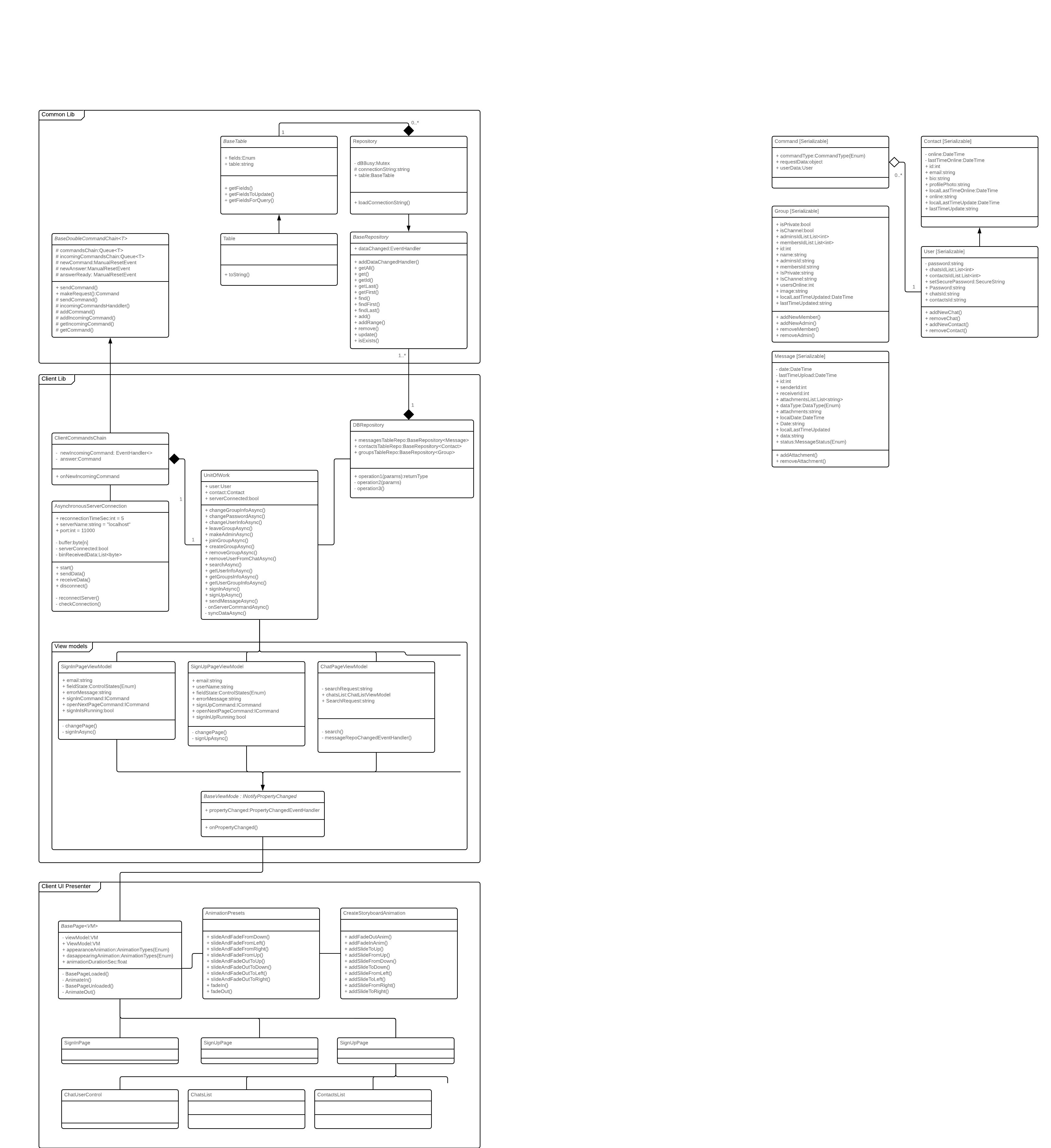


Рис. 2.3. Продовження діаграми класів клієнтського додатку

Розпочнемо огляд усіх класів з бібліотеки **Common Lib**. В ній знаходяться класи та абстракції, які ми використовуємо на клієнті та на сервері. Першими підуть класи, які не зображені на діаграмі (див. рис. 2.7), але представляють основні типи даних, з якими ми оперуємо.

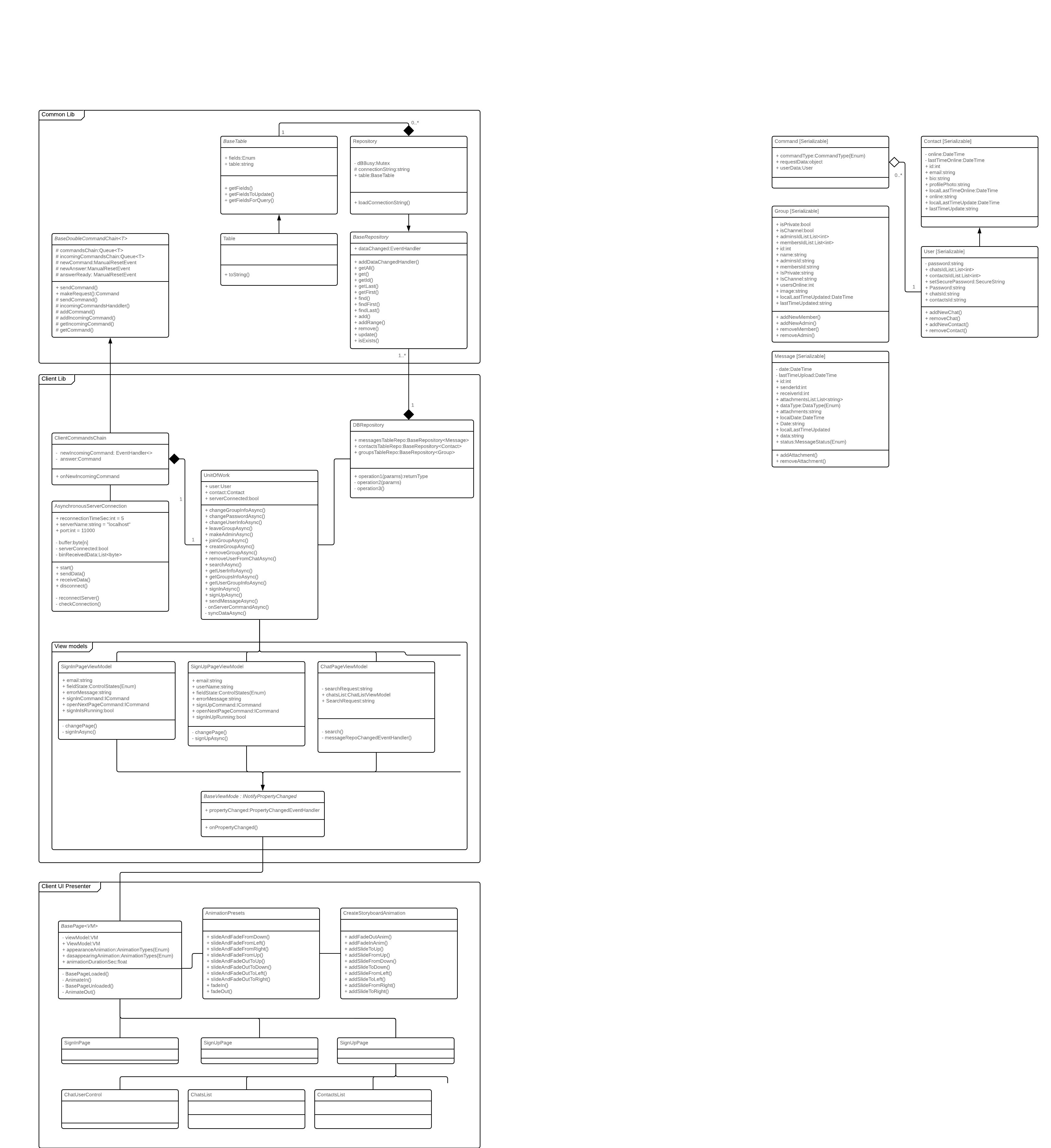


Рис. 2.4. Основні типи даних

**Contact** – клас, який зберігає найголовнішу інформацію користувача, він використовується у випадках, коли потрібно відобразити найголовнішу інформацію, але такі дані, як пароль передавати не хочеться.

**User** – він наслідується від класа Contact, і додає додаткові поля для пароля, списку груп, а також методи для додавання та видалення елементів списку. Звернемо увагу на деякі властивості. Для збереження пароля ми використовуємо два поля: **SetSecurePassword** та **Password**.

/// <summary>

/// Convert secured password to hash string

/// </summary>

public SecureString SetSecurePassword { set => password = value.GetHash(); }

/// <summary>

/// Store hashed version of secured password

/// </summary>

public string Password {

get

{

if (password != null)

return password;

return null;

}

set

{

password = value;

}

}

При реєстрації користувач вводить пароль в захищений PasswordBox звідки ми зчитуємо SecureString – це спеціальний тип даних розроблений для того, щоб зберегти вміст захищеним, а зоб отримати звичайний текст потрібно виконати деякі операції з показниками. Після введення усіх даних створюється новий об’єкт класа User куди і передається захищений пароль. За допомогою властивості **SetSecurePassword** пароль ми дістаємо пароль з пам’яті та зразу ж хешуємо його, за допомогою функції.

Лістинг класу SecureStringHelper:

public static class SecureStringHelper

{

/// <summary>

/// Usecures a secure string to plane text

/// </summary>

/// <param name="secureString">secure string</param>

/// <returns></returns>

public static string Unsecure(this SecureString secureString)

{

if (secureString == null)

return "";

// get pointer for an unsecure string in memory

var unmanagedString = IntPtr.Zero;

try

{

//Unsecure string

unmanagedString = Marshal.SecureStringToGlobalAllocUnicode(secureString);

return Marshal.PtrToStringUni(unmanagedString);

}

finally

{

//Clean up any memory allocations

Marshal.ZeroFreeGlobalAllocUnicode(unmanagedString);

}

}

/// <summary>

/// Gets hash string from secured password and salt string

/// </summary>

/// <param name="secureString"></param>

/// <param name="salt">Random constant to make decrypting more dificult</param>

/// <returns></returns>

public static string GetHash(this SecureString secureString, string salt = "I(<cOol>" )

{

// usually i will use User Name as a value and some ford for a salt

// get salted byte[] buffer, containing value, password and some (constant) salt

byte[] buffer;

using (MemoryStream stream = new MemoryStream())

using (StreamWriter writer = new StreamWriter(stream))

{

writer.Write(salt);

writer.Write(secureString.Unsecure());

writer.Flush();

buffer = stream.ToArray();

}

// create a hash

SHA1 sha1 = SHA1.Create();

return Encoding.Default.GetString(sha1.ComputeHash(buffer));

}

}

Метод **Unsecure** приймає SecureString та повертає звичайний текст. Метод **GetHash()** приймає SecureString та ще одну строку під назвою **salt**, вони конвертуються в потік байтів після чого хешуються за алгоритмом SHA1, хешування відбувається в одному напрямку, тобто назад текст отримати вже не вийде. Але і не потрібно, при вході в акаунт ми таким же чином захешуємо пароль та відправимо на сервер, де ми порівняємо його з захешованим оригіналом.

Далі ідуть два поля які конвертують списки з ідентифікаторами чатів та контактів в текстовий формат за допомогою класу **DataConverter**. Це потрібно для того щоб зберігати дані в базу даних.

/// <summary>

/// Return string representation of chats list

/// </summary>

public string ChatsId {

get

{

return DataConverter.ListToString(chatsIdList);

}

set

{

chatsIdList = DataConverter.StringToIntList(value);

}

}

/// <summary>

/// Return string representation of contacts list

/// </summary>

public string ContactsId {

get

{

return DataConverter.ListToString(contactsIdList);

}

set

{

contactsIdList = DataConverter.StringToIntList(value);

}

}

**Group** – цей клас зберігає усю необхідну інформацію групи.

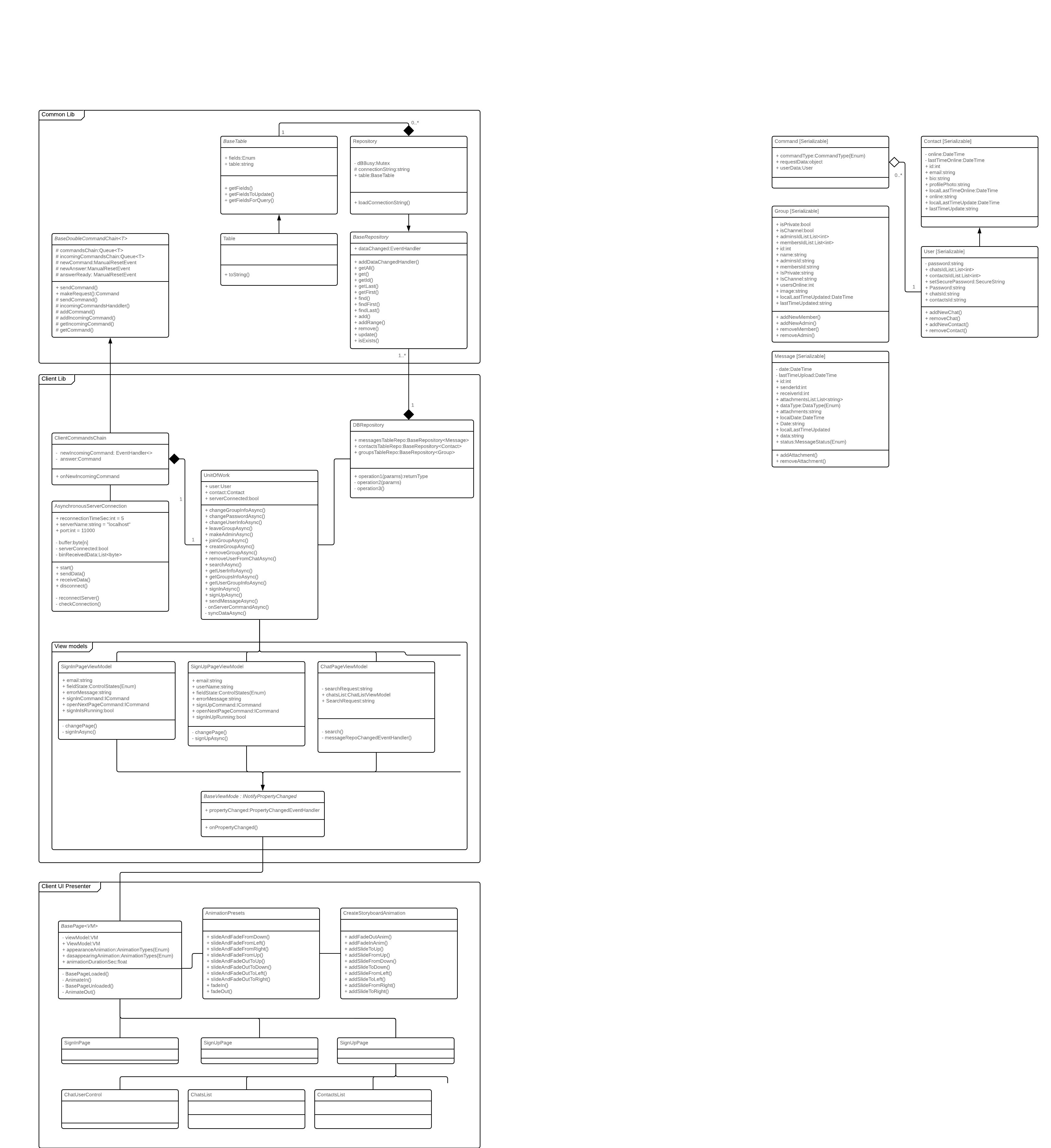


Рис. 2.5. Схема класу Group

На схемі ми можемо побачити такі поля, як **isPrivate** і **isChannel**. Вони відповідають за налаштування групи. Якщо група буде приватною, то при пошуку ця група не буде відображатися. Щоб підключитися до неї потрібно, щоб один з членів групи запросив вас. Якщо група буде каналом, то публікувати повідомлення зможуть лише адміністратори групи. Також ми можемо побачити властивості: **localLastTimeUpdated** та **lastTimeUpdated**.

/// <summary>

/// Get and set local last time updated DateTime

/// </summary>

public DateTime LocalLastTimeUpdated

{

get

{

return lastTimeUpdated;

}

set

{

lastTimeUpdated = value;

}

}

/// <summary>

/// Converted binary representation of universal time

/// </summary>

public string LastTimeUpdated

{

get

{

return lastTimeUpdated.ToUniversalTime().ToBinary().ToString();

}

set

{

lastTimeUpdated = DateTime.FromBinary((long)Convert.ToUInt64(value)).ToLocalTime();

}

}

Вони відповідають за збереження часу останнього оновлення групи. **LocalLastTimeUpdated** простоприсвоює значення в приватну змінну, в той час коли **LastTimeUpdated** конвертує з бінарного представлення глобального часу в DateTime та навпаки. Такі махінації потрібні, якщо наприклад користувач підключається до серверу, який знаходиться в іншому часовому поясі.

**Message** – клас, який надає необхідні поля для опису повідомлення.

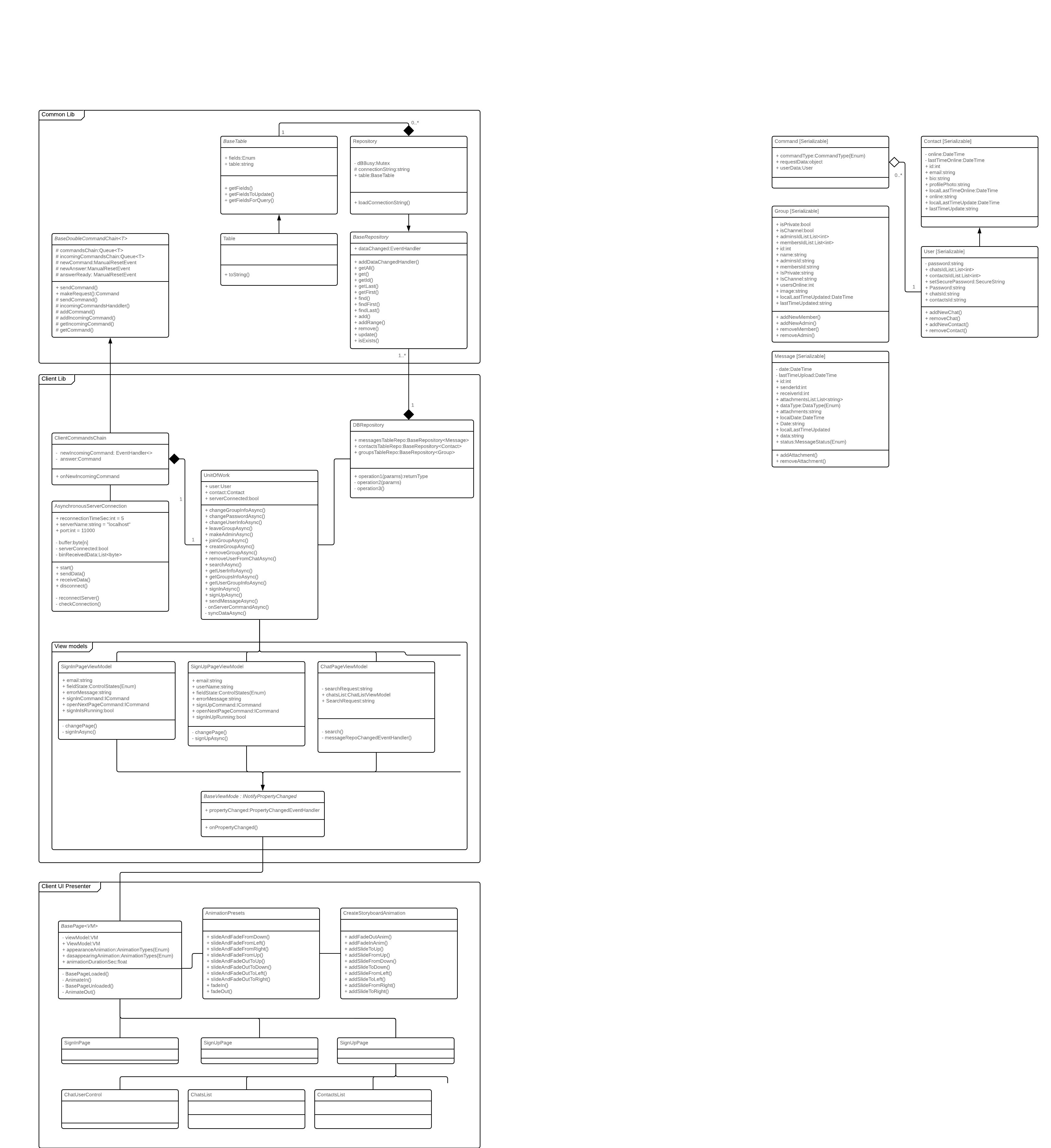


Рис. 2.6. Схема класу Message

Поле **SenderId** – це Id юзера, який відправив повідомлення, **ReceiverId** – це Id групи, в яку було відправлено повідомлення. Також мы тут можемо побачити **AttachmentsList** – це список усіх прикріплених до повідомлення файлів (фотографій, файлів). Властивість **Status** відповідає за поточний статус повідомлення: в процесі відправлення, відправлено, прочитано.

Останнім базовим класом є клас Command.

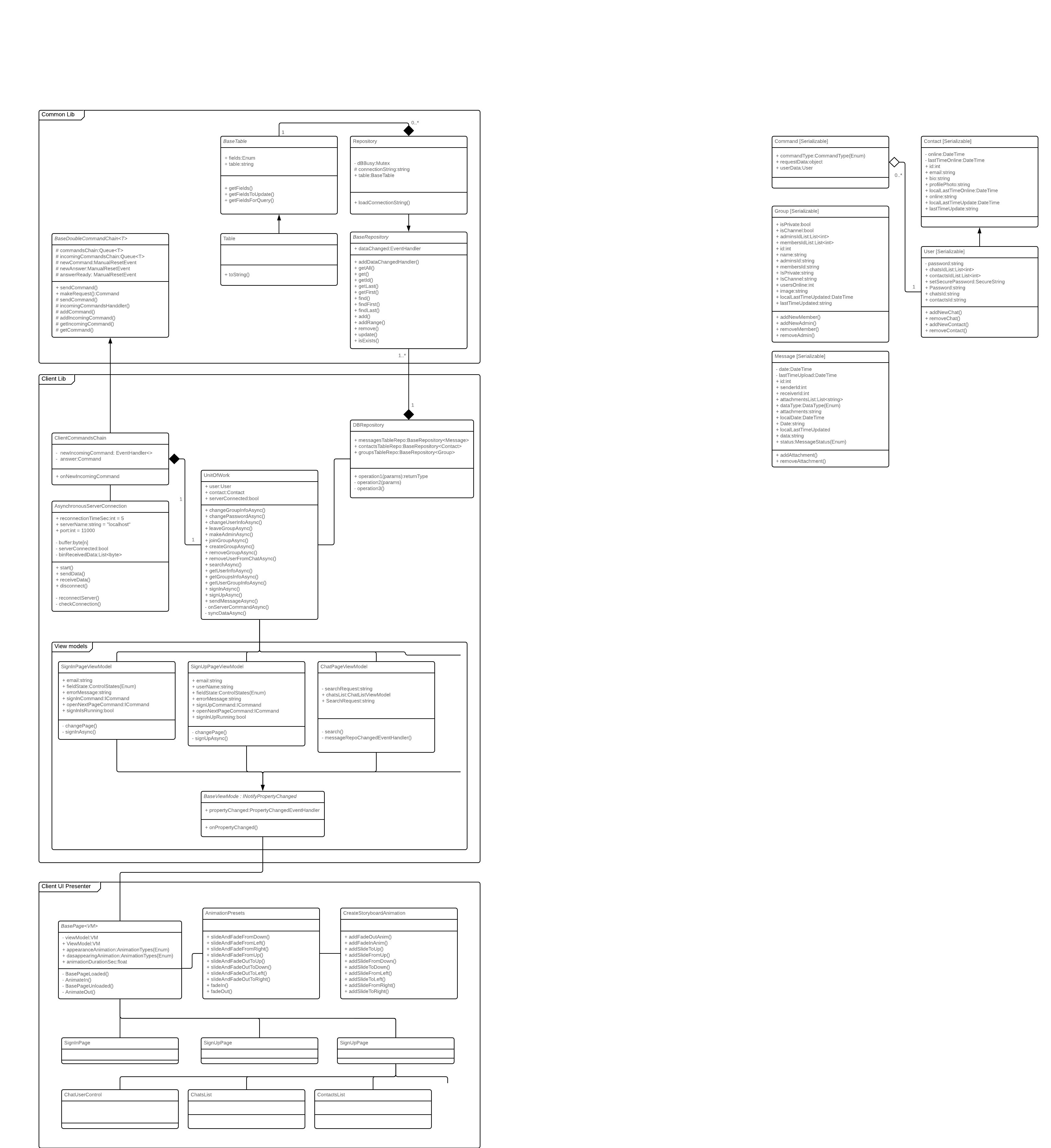


Рис. 2.7. Схема класу Command

Він представляє команду, за допомогою якої, сервер та клієнт обмінюються інформацією. В класі зазначено три поля. Перше – **CommandType**, воно представляє тип команди (SignUp, SignIn, SendMessage…), перелік типів досить великий і він буде зазначений в додатках. **RequestData** – поле типу об’єкт, для передачі потрібної інформації. **UserData –** поле для передачі інформації клієнта.

Всі вищезазначені класи мають атрибут **Serializable**, який дає можливість перетворювати класів в масив байтів і навпаки. Ця здібність буде використовуватися при обміні клієнта та сервера.

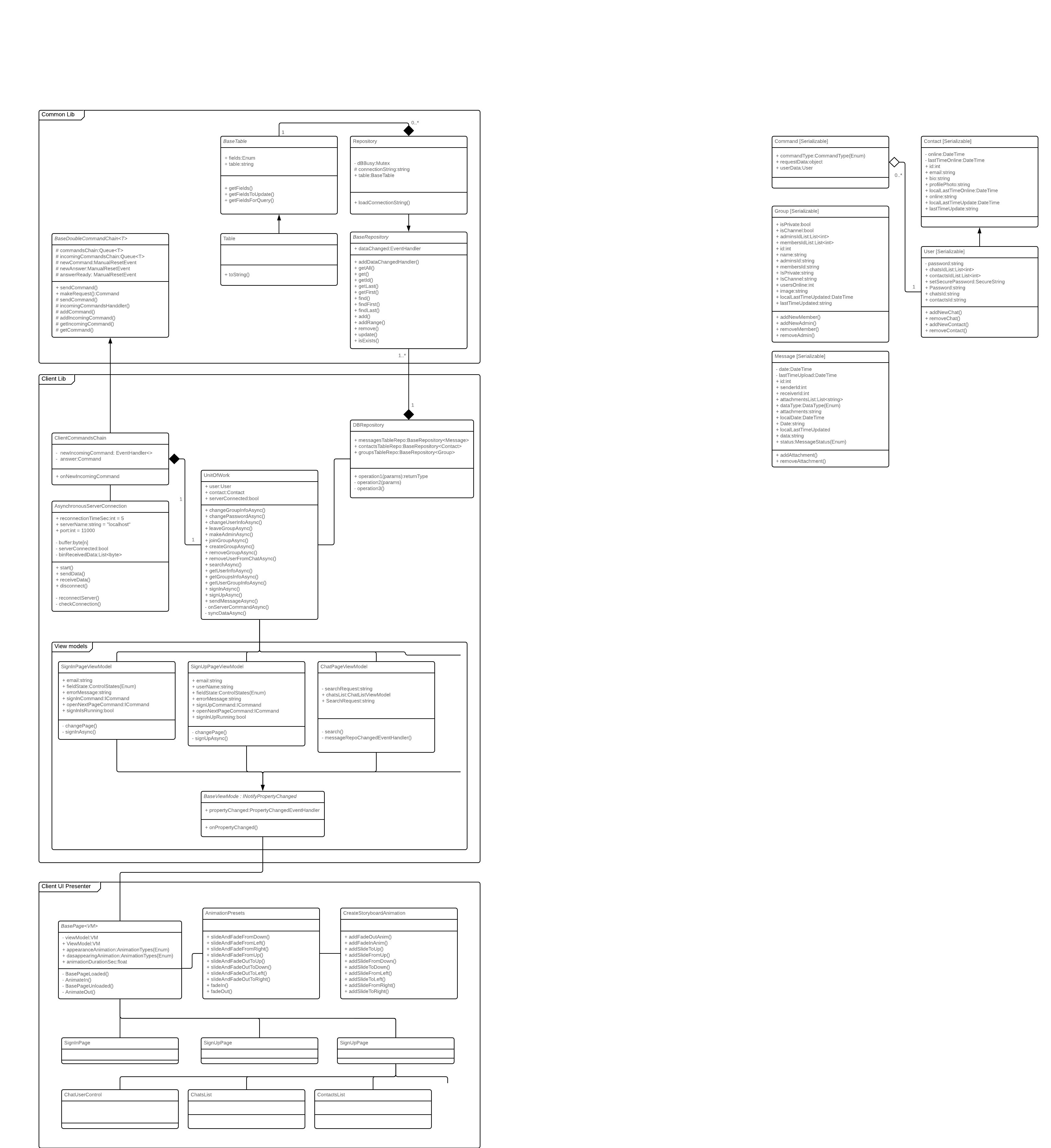


Рис. 2.8. Діаграма класів Common Lib

Після того, як ми описали всі звичайні класи, переходимо до головних класів бібліотеки Common Lib. Почнемо з класу **BaseDoubleCommandChain<T>** - загальний клас, про що свідчить літера **T** після назви. Це означає, що при створенні нового екземпляра класа, ми повинні задати тип даних, який буде представлятися цією літерою. В переліку полів ми бачимо дві захищені черги, які зберігають дані типу **T**. Вони потрібні для збереження команд, які чекають обробки або відправки на сервер. Далі ідуть три поля типу **ManualResetEvent** які потрібні для сигналізування завершення дії.

Далі іде черга наслідувань зв’язаних з репозиторієм. Почнемо з абстрактного класу **BaseTable**, в якому ми бачимо поле під назвою **Fields.** Воно представляє перелік полів таблиці бази даних. Далі йде поле **Table** – це назва таблиці. Наступні три методи потрібні для того, щоб повертати рядок, який ми будемо використовувати для формування запиту до бази даних.

**BaseRepository –** абстрактний клас, який зберігає перелік найпростіших функцій для роботи з таблицею бази даних, а також подію яка викликається під час зміни таблиці бази даних. Якщо знадобиться створити клас з конкретним розширеним функціоналом, то ми створимо ще один абстрактний який унаслідуватиметься від **BaseRepository**. Далі від цієї абстракції наслідується клас **Repository**, який реалізує усі ці методи. Для роботи з базою даних **SQLite** я використовую плагін під назвою **Dapper**. Це Дуже простий Об'єктний реляційний маппер, який відповідає за відображення між базою даних і мовою програмування. Його роботу ми перевіримо на парі методів з класу **Repository**.

Лістинг методів класу Repository:

/// <summary>

/// Adds data to the table

/// </summary>

/// <param name="data">Info to add</param>

public override bool Add(T data)

{

DBBusy.WaitOne();

try

{

using (var cnn = new SQLiteConnection(LoadConnectionString()))

{

cnn.Open();

string request = "INSERT INTO " + table.ToString() + " ( " + table.GetFields() + " ) VALUES ( " + table.GetFieldsForQuery() + " )";

cnn.Execute(request, data);

}

}

catch (Exception e)

{

Console.WriteLine(e);

DBBusy.ReleaseMutex();

return false;

}

var ls = new List<T>();

ls.Add(data);

//Fires INotifyDataChanged event

DataChanged?.Invoke(this, new DataChangedArgs<IEnumerable<T>>(ls, table.ToString(), RepositoryActions.Add));

DBBusy.ReleaseMutex();

return true;

}

/// <summary>

/// Finds all the matches in table

/// </summary>

/// <param name="column">Columnt to search</param>

/// <param name="value">Search parameter</param>

/// <returns></returns>

public override IEnumerable<T> Find(string column, string value)

{

DBBusy.WaitOne();

var dp = new DynamicParameters();

dp.Add(column, value);

try

{

using (IDbConnection con = new SQLiteConnection(LoadConnectionString()))

{

con.Open();

string request = "select \* from " + table.ToString() + " WHERE " + column + " =@" + column;

var query = con.Query<T>(request, dp);

DBBusy.ReleaseMutex();

return query.ToList<T>();

}

}

catch (Exception e)

{

Console.WriteLine(e);

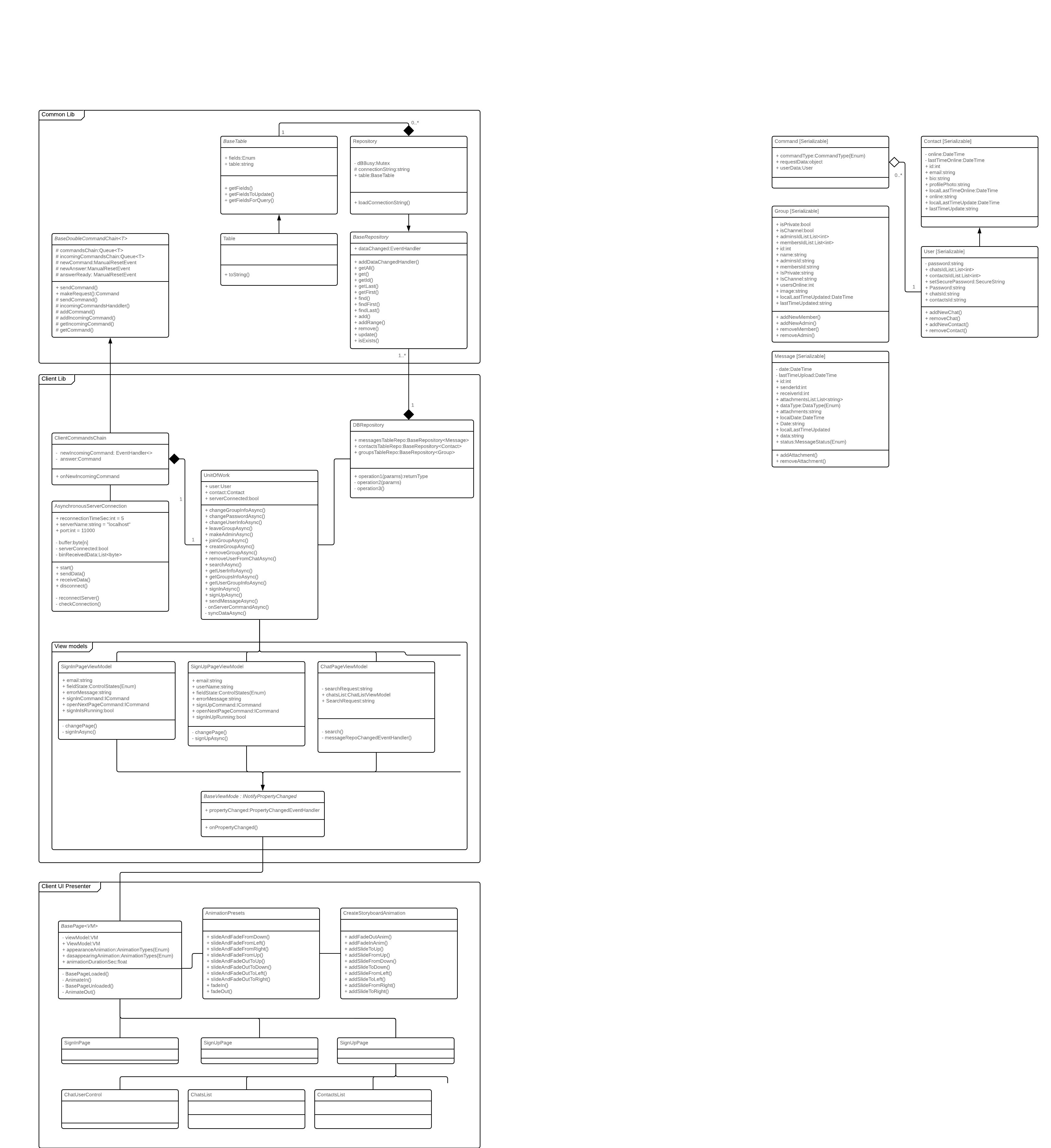
DBBusy.ReleaseMutex();

return null;

}

}

Перше, що ми можемо побачити, що цей клас, знову ж таки, є загальним – це означає, що ми можемо працювати з будь-якими типами даних. На перших рядках функцій ми можемо зустріти вираз DBBusy.WaitOne() - це вираз, що допомагає потокам однієї програми, співпрацювати без колізій, які можуть відбутися у випадку, коли ми захочимо відкрити базу даних, з різних потоків одночасно. Для вирішення такої проблеми в C# є багато класів, які відрізняються складністю використання. У циї функціях було використано мьютекси. Коли один потік заходить в функцію, він викликає метод під назвою **WaitOne()** – це означає, що всі інші потоки, які захочуть скористатися методом, будуть чекати доки попередній потік не викличе метод **ReleaseMutex**, який дозволить наступному потоку пройти всередину. Далі іде вираз try catch, який не дає програмі вилетіти у разі, якщо при роботі базою даних станеться помилка. Для того, щоб під ’єднатися до бази даних ми використовуємо using() а також **SQLiteConnection** з бібліотеки для роботи з sqlite. В зміну **request** ми записуємо рядок зі стандартним запитом до будь-якої бази даних стандарту sql. Далі ми викликаємо обробку запиту за допомогою функцій **Query<T>** або **Execute**. Головна відмінність між ними у тому, що **Query**, повертає значення які ми запитували, коли **Execute** просто виконує запит. Якщо метод вносить зміни в таблицю, то тоді в кінці ми побачимо виклик події **DataChanged** цієї таблиці. Він передає дані, які було змінено, також назву таблиці і дію, яка відбулася над таблицею. На цю подію підписані класи, які знаходяться у **ViewModel** і оновлюють інформацію, що демонструється клієнту.



2.9. Діаграма класів бібліотеки Client Lib

Наступною іде частина з бібліотекою Client Lib. Ця бібліотека використовується виключно клієнтським додатком. Огляд ми розпочнемо з класу **AsynchronousServerConnection**. Його назва каже все за себе, він займається підключенням та обміном даних з сервером при цьому в окремих потоках. Розглянемо декілька цікавих методів.

Лістинг методу Start класу AsynchronousServerConnection:

/// <summary>

/// Conects to server

/// </summary>

static public void Start()

{

//Get local ip addresses

//TODO change local IP addresses on server IP

IPHostEntry ipHost = Dns.GetHostEntry(ServerName);

IPAddress ipAddress = ipHost.AddressList[0];

//Specify end remote point

RemoteEndPoint = new IPEndPoint(ipAddress, Port);

//Create new socket

Socket socket = new Socket(RemoteEndPoint.AddressFamily, SocketType.Stream, ProtocolType.Tcp);

Socket = socket;

do

{

try

{

//Start connecting

var asyncResult = Socket.BeginConnect(RemoteEndPoint, null, null);

Socket.EndConnect(asyncResult);

ServerConnected = true;

//Notify connection changed

connectionChanged(null, ServerConnected);

//Start receiving data

Thread receiveData = new Thread(new ThreadStart(ReceiveData));

receiveData.IsBackground = true;

receiveData.Name = "Receiver From Server";

receiveData.Start();

//Start checking connection

Thread checkingConnection = new Thread(new ThreadStart(checkConnection));

checkingConnection.IsBackground = true;

checkingConnection.Name = "Check Connection";

checkingConnection.Start();

}

catch (Exception e)

{

DisplayMessageOnScreen($"Error on starting connectiong, try again after {ReconectionTimeSeconds} seconds \n " + e.ToString());

}

//If we didn't connect to server, than try again

if (!ServerConnected)

Thread.Sleep(ReconectionTimeSeconds \* 1000);

else

break;

} while (true);

}

Щоб під ‘єднатися до серверу, ми використовуємо сокети. Спочатку потрібно створити точку, до якої ми спробуємо під ‘єднатися. Вона складається з IP адресу сервера та порту, який слухає наш додаток, у цьому випадку це 11000. Створюємо сокет, після чого намагаємося підключитися, якщо виникає exception, то try catch ловить його, після чого перевіряється чи ми під ‘єдналися до сервера, якщо ні, то ми чекаємо ReconectionTimeSeconds секунд, після чого повторюємо спробу. Для підключення та обміну інформацією ми використовуємо асинхронні методи, про що свідчать приставки **Begin** та **End**. Після вдалого підключення о серверу ми створюємо два потоки, один з яких буде відповідати за отримання даних від сервера, а інший, за перевірку підключення до сервера, у разі відключення, він змінить змінну **ServerConnected** та знову спробує підключитися.

Лістинг методу ReceiveData класу AsynchronousServerConnection:

static public void ReceiveData()

{

string content = String.Empty;

int bytesRead;

try

{

var asyncResult = Socket.BeginReceive(buffer, 0, bufferSize, 0, null, null);

bytesRead = Socket.EndReceive(asyncResult);

}

catch (Exception)

{

DisplayMessageOnScreen("Error on receive data from server");

return;

}

if (bytesRead > 0)

{

//Check the end of file, if there isn't <EOF> label, then continue reading

content = Encoding.Default.GetString(buffer);

//Start of <EOF> label

var EOFIndex = content.IndexOf("<EOF>");

if (EOFIndex > -1)

{

//Create new temp buffer with size of EOFIndex

byte[] temp = new byte[EOFIndex];

//Delete part after <EOF> label

Array.Copy(buffer, temp, temp.Length);

//Copy buffer to bin recieved data

binReceivedData.AddRange(temp);

//Notify listeners about recieved data is ready to handle

answerDataReady.Invoke(binReceivedData, binReceivedData.ToArray());

//All the data has benn read from the client. Display it on the console

DisplayMessageOnScreen($"Read {content.Length} bytes from server");

//Clear binReceivedData to fill it with new data

binReceivedData = new List<byte>();

//Start again

ReceiveData();

}

else

{

//Copy buffer to bin recieved data

binReceivedData.AddRange(buffer);

//Continue reading data

ReceiveData();

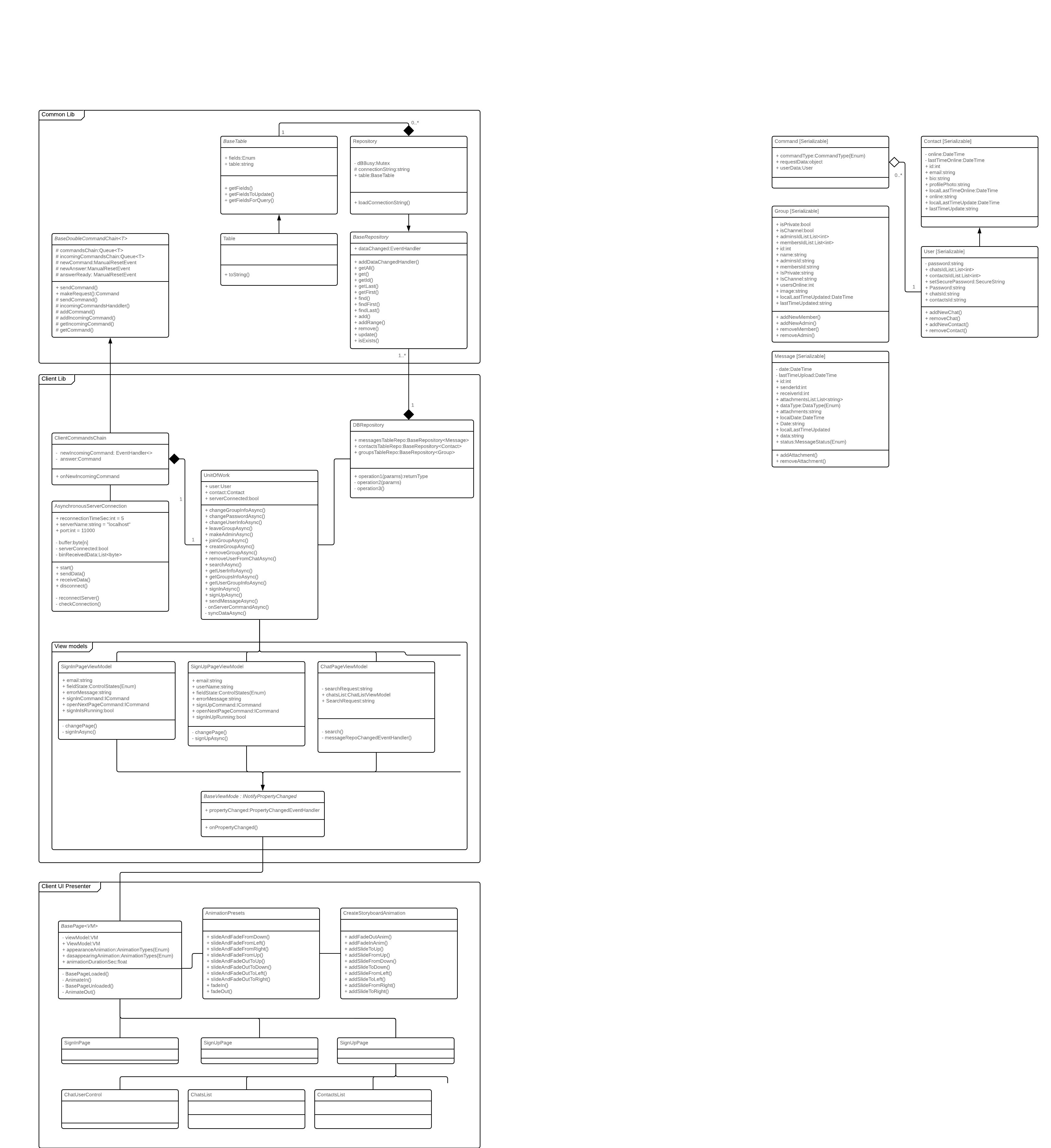
}

}

}

Наступний метод потрібний для того, щоб очікувати вхідних даних від сервера. Після того, як він отримав деяку кількість байтів потрібно обробити їх. Так як ми не знаємо скільки разів потрібно потрібно приймати масиви байтів від сервера, то в кінець кожного відправленого масива ми додаємо **<EOF>**. Це повідомлення означає, що ми отримали останній масив байтів від сервера, і що ми можемо почати обробляти отримані байти в повному обсязі. Щоб перевірити наявність лейбла, ми конвертуємо масив байтів в рядок, після чого пробуємо знайти ключове слово, якщо не знаходимо, то копіюємо отримані байти, та починаємо знову. Після того як ми знайдемо наше повідомлення, відділяємо непотрібну частину, далі ми викликаємо подію **answerDataReady** та передаємо отримані файли в повному обсязі.

Ці данні потрапляють у наступний клас, який ми розглянемо. **ClientCommandChain** (див. рис. 2.13), цей клас відповідальний за те щоб додавати команди, які очікують відправки, та команди, які щойно прийшли від сервера, в чергу. Це дає змогу чекати доки попередня команда виконається і не втрачати нові команди.



2.10. Схема класу ClientCommandChain

Лістинг методу addIncomingCommand класу ClientCommandChain:

/// <summary>

/// Addes new incoming command to the chain and signales <see cref="BaseDoubleCommandChain{T}.newAnswer"/>

/// </summary>

/// <param name="sender"></param>

/// <param name="args"></param>

protected override void addIncomingCommand(object sender, byte[] args)

{

//Get bytes from args and deserialize them

var data = (Command)DataConverter.DeserializeData(args);

incomingCommandsChain.Enqueue(data);

//Signal AnswersHandle to continue handle incoming commands

newAnswer.Set();

}

Саме цей метод є підписником події, що викликає сервер при отриманні останньої порції даних від сервера, а саме **answerDataReady**. Головна ціль методу – десереалізувати команду після чого додати в чергу і сигналізувати, що прийшла нова команда від сервера. Саме цим займається **newAnswer**, який представляє ManualResetEvent, як тип даних. Це ще один спосіб синхронізувати роботу окремих потоків.

Лістинг методу incomingCommandsHandler класу ClientCommandChain:

protected override void incomingCommandsHanddler()

{

while (true)

{

//If there isn't any command, then wait for a signal from addIncomingCommand method

if (incomingCommandsChain.Count == 0)

newAnswer.WaitOne();

newAnswer.Reset();

//Check all the commands

while (incomingCommandsChain.Count != 0)

{

var incomingCommand = getIncomingCommand();

//if command has type answer, then save it

if (incomingCommand.CommandType == CommandType.Answer)

{

//copy command

this.answer = incomingCommand;

//Signal MakeRequest method about answer is ready

answerReady.Set();

}

//else raise an newCommandFromTheServer event to send it further

else

{

newIncomingCommand?.Invoke(this, incomingCommand);

}

}

}

}

Цей метод після отримання сигналу, перевіряє чи в черзі є команди та по одній починає забирати їх з черги. Якщо тип команди відповідь, то це означає, що перед цим ми робили запит до сервера, і чекаємо на відповідь, яка щойно прийша, тоді ми присвоюємо команду в змінну відповіді і сигналізуємо потік, який чекає на відповідь. Якщо ж тип команди не відповідь, тоді ми викликаємо подію отримання нової і передаємо команду туди.

Лістинг методу MakeRequest класу ClientCommandChain:

/// <summary>

/// Make request to the server and wait for the answer

/// </summary>

/// <param name="comType"></param>

/// <param name="data"></param>

/// <param name="user"></param>

/// <returns>Answer command</returns>

public override Command MakeRequest(CommandType comType, object data, User user)

{

//Adding command to the queue

addCommand(new Command(comType, data, user));

//Wait for a signal, if we don't recieve any response, go further

answerReady.WaitOne(20000);

answerReady.Reset();

//Clear last command

var com = answer;

answer = null;

return com;

}

Це одна з команд, за допомогою яких ми відправляємо команди на сервер. Особливість цього методу в тому, що після додавання команди в чергу на відправлення, ми будемо чекати сигналу **answerReady** від функції incomingCommandsHandler, якщо ми в цей момент обірвемо з’єднання з сервером, то нічого поганого не станеться, тому що в нас стоїть таймер в 20 секунд, після чого ми просто продовжуємо роботу і повертаємо відповідь на команду або null.

Лістинг методу SendCommand класу ClientCommandChain:

/// <summary>

/// Send command without waiting for the answer

/// </summary>

/// <param name="comType"></param>

/// <param name="data"></param>

/// <param name="user"></param>

public override void SendCommand(CommandType comType, object data, User user)

{

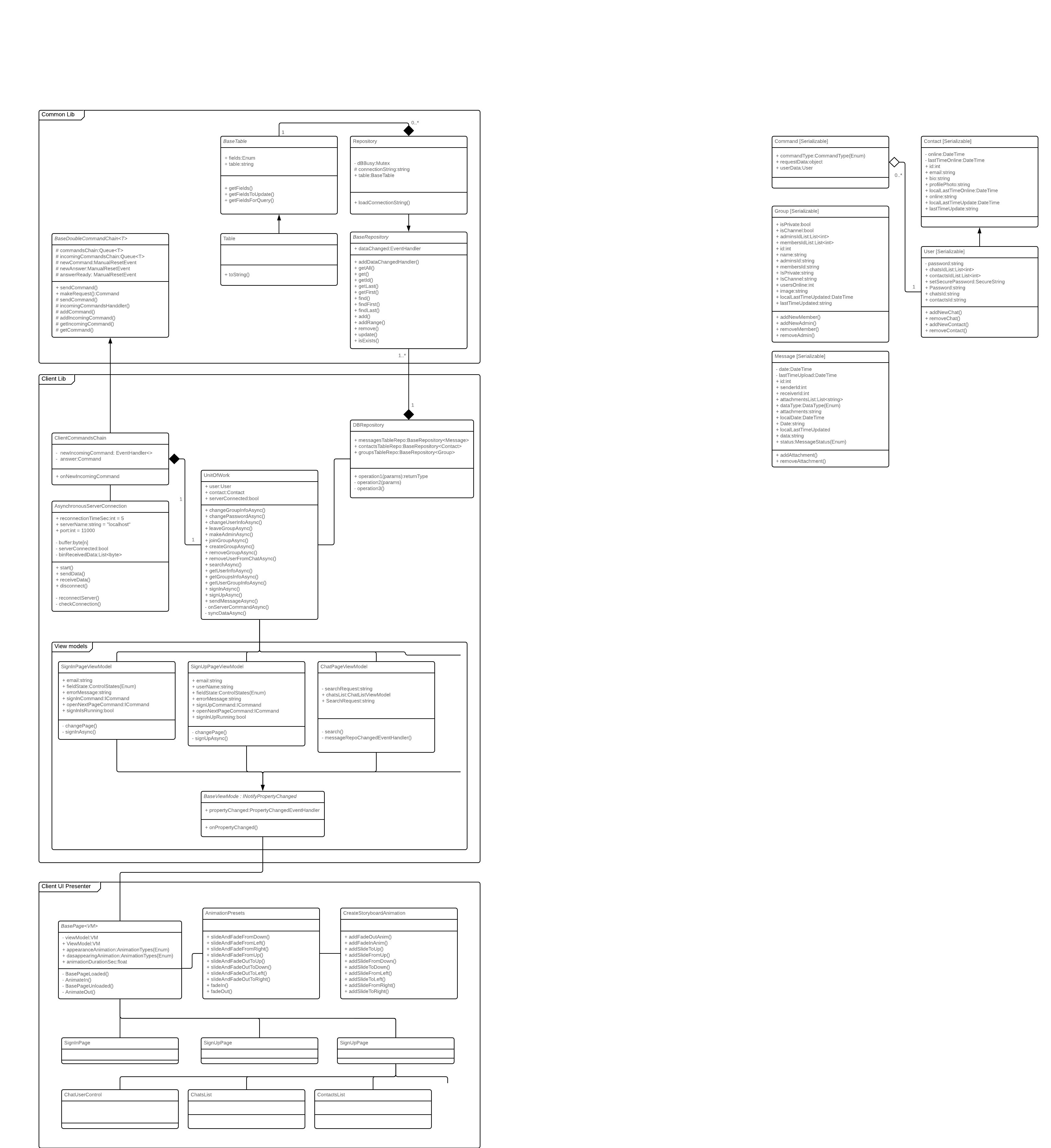
//Adding command to the queue

addCommand(new Command(comType, data, user));

}

Цей метод просто додає команду в чергу на відправлення без очікування відповіді.

Наступний клас, який ми швидко переглянемо буде **DBRepository** (див. рис. 2.14). Усе що віг робить - це агрегує в собі поля, які представляють усі таблиці локальної бази даних, та надає доступ до них.



2.11. Схема класу DBRepository

Наступний клас **UnitOfWork**. Це центральний клас, який збирає в собі можливості усіх попередньо розглянутих класів та надає їх **ViewModel**, які і будуть відображати усю потрібну інформацію користувачю.

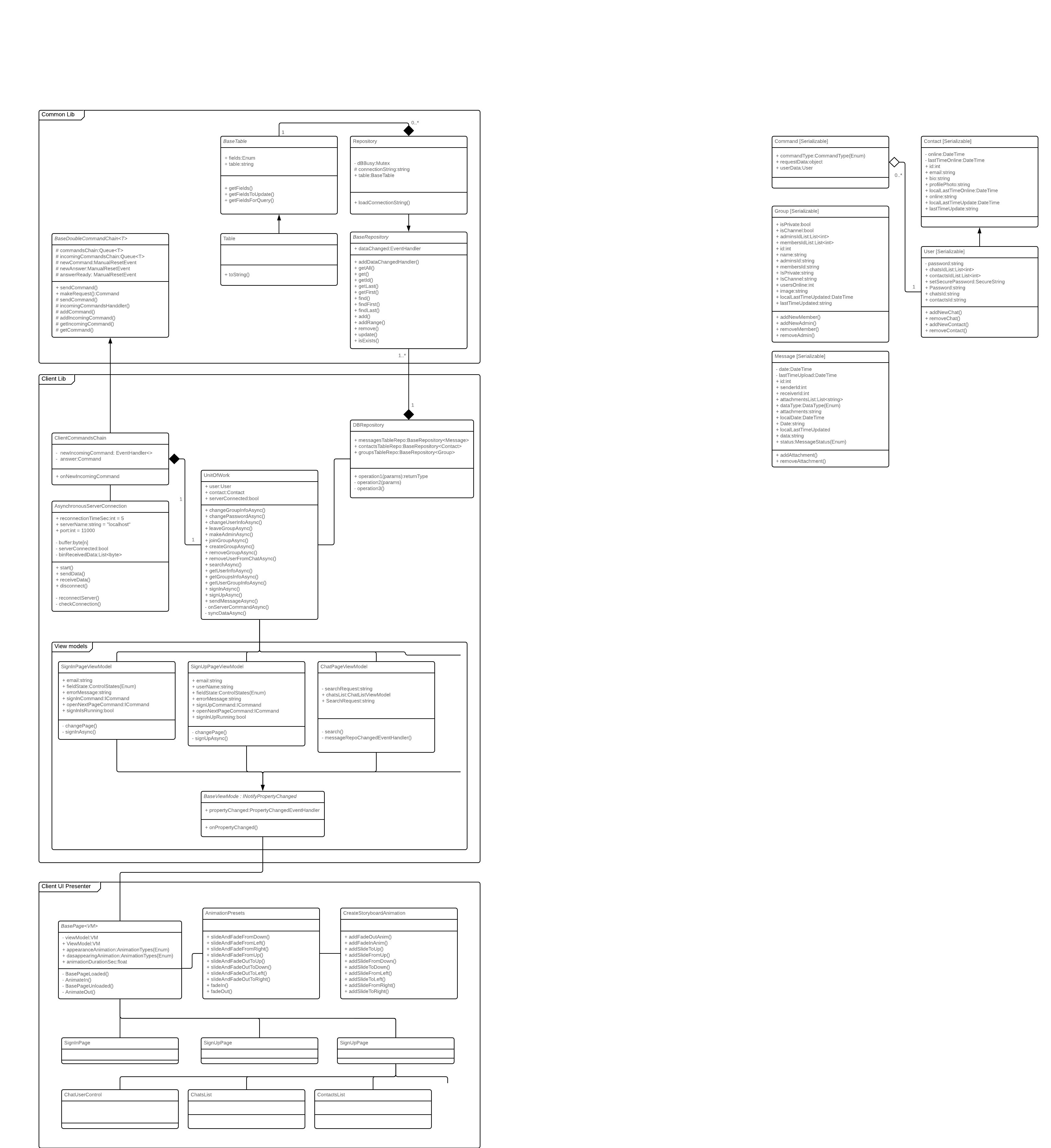


Рис. 2.12. Схема класу UnitOfWork

В класі розміщаються функції, які створюють команди та обробляють відповіді сервера. Для прикладу розглянемо декілька методів.

Лістинг методу SignIn класу UnitOfWork:

/// <summary>

/// Make Log In request to the server, return true if all ok

/// </summary>

/// <param name="userData"></param>

/// <returns></returns>

public static async Task<bool> SighIn(User userData)

{

if (!ServerConnected)

return false;

var result = await Task.Run(() =>

{

//Make request to the server

var response = commandChain.MakeRequest(CommandType.SignIn, null, userData);

//If response time is out

if (response == null)

return false;

// if all Ok save user data

if (response.RequestData != null)

{

User = response.UserData;

syncData((object[])response.RequestData);

return true;

}

return false;

});

return result;

}

Потрібно звернути увагу на те, що усі ці методи працюють асинхронно - це дуже важлива особливість, якщо для прикладу ви натиснете на кнопку і керування перейде у цей метод, а обробка команд займе деякий час, то увесь час роботи методу, ви не зможете взаємодіяти з вікном додатку.

Спочатку ми перевіряємо підключення до серверу, якщо його немає, ми просто виходимо з методу. Далі робимо запит до сервера і після отримання відповіді перевіряємо, чи відповідь прийшла. Якщо відповідь прийшла і **RequestData** не дорівнює нулю, то це означає що все пройшло нормально і логін та пароль підходять. Далі ми зберігаємо інформацію юзера і починаємо синхронізацію даних. Далі повертаємо заначення.

Лістинг методу CreateGroup класу UnitOfWork:

/// <summary>

/// Sends command to create a group

/// </summary>

/// <param name="group"></param>

public static async Task CreateGroup(Group group)

{

if (ServerConnected)

{

await Task.Run(() =>

{

var res = commandChain.MakeRequest(CommandType.CreateGroup, group, User);

//If response time is out

if (res == null)

return;

//If group created, add to local repo

if (res.RequestData != null)

{

//Add group to db

GroupsTableRepo.Add((Group)res.RequestData);

//Update user

User = res.UserData;

OnUserUpdated(null, new DataChangedArgs<IEnumerable<object>>(new List<object>() { ((Group)res.RequestData).Id }, UsersTableFields.ChatsId.ToString(), RepositoryActions.Add));

}

});

}

}

Наступна функція відправляє команду на створення нової групи, після чого додає її в локальну базу даних та оновлює інформацію юзера.

Після цього іде великий блок з об’єктами, які допомагають вивести інформацію на екран. Тут досить велика кількість об’єктів, у зв’язку з тим, що я створював окремий **ViewModel** для кожної нової сторінки, списку. Найбільші та найголовніші ви можете побачити на схемі (див. рис.)

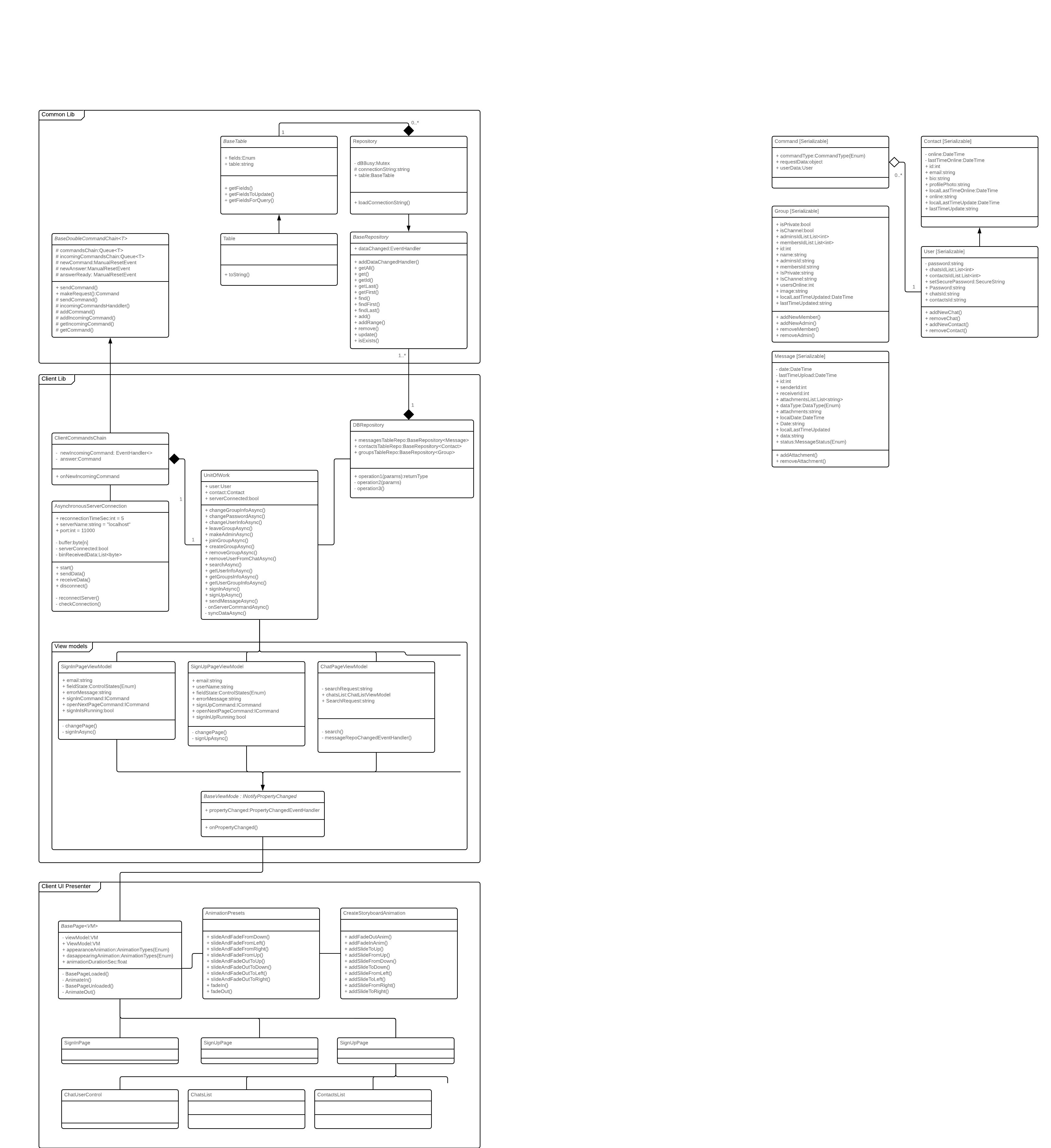


Рис. 2.13. Схема деяких об’єктів представлення інтерфейсу

Перше, що ми можемо побачити – це те, що всі вони наслідуються від **BaseViewModel**. Цей клас в свою чергу наслідується від інтерфейсу **INotifyPropertyChanged**, який гарантує наявність події **PropertyChanged** та методу **OnPropertyChanged**, який просто викликає подію і передає туди назву властивості, яка щойно змінилася. Вищезазначена подія потрібна для того, щоб інформувати інтерфейс про те, що інформацію потрібно оновити. Без цього ми не зможемо динамічно змінювати змінні і бачити результат. Взагалі, щоб інтерфейс автоматично змінювався, то потрібно після кожного присвоювання нового значення змінній, викликати функцію **OnPropertyChanged** і передати туди назву властивості, що змінилася. Але, щоб не писати багато лишнього коду, та зекономити час, ми використали додаток **PropertyChanged.Fody** від **Fody**. Він автоматично знаходить властивості, які потрібно оновлювати, та підставляє туди виклик методу **OnPropertyChanged**, але ми цього не побачимо.

Наша програма розділена на три сторінки, які відображаються в **MainWindow** і уміщують у собі контент. Це дає змогу костимізувати їх стандартні класи, додаючи анімацію зміни однієї сторінки на іншу. Сторінка **ChatPage** уміщує у собі головний контент, який також розділено на багато елементів, таких як **ChatList**, **ContactsList**, **UserInfo**, **GroupInfo**... Використовуючи такий підхід, ми можемо створити усі необхідні елементи і у разі потреби збираючи їх, як окремі модулі, при цьому не дублюючи код.

Розглянемо декілька **ViewModels** детальніше.

Лістинг властивостей класу SignUpPageViewModel:

#region Public Members

public ControlStates FieldState { get; set; } = ControlStates.NormalGray;

public string ErrorMessage { get; set; } = String.Empty;

public string UserName { get; set; } = String.Empty;

public string Email { get; set; } = String.Empty;

//Commands for system buttons

[DoNotNotify]

public ICommand SignInCommand { get; set; }

[DoNotNotify]

// Opens new page

public ICommand OpenNextPageCommand { get; set; }

public bool SignInIsRunning { get; set; } = false;

#endregion

З лістингу ми можемо побачити, що в нас є декілька властивостей, які пов’язані з інтерфейсом: **UserName**, **Email**. Коли ми вводимо свої данні в поля, вони потрапляють сюди. Також ми можемо побачити поля ErrorMessage і FieldState. Вони відповідають за відображення помилки, а також за зміну кольору потрібних полів. Після ідуть дві команди, які відповідають за вхід та зміну сторінки. Прив’язавши до команд потрібні функції ми можемо вказати їх в xaml коді, тоді при натисканні на кнопку буде спрацьовувати потрібна команда. Остання властивість **SignInIsRunning** використовується для того, щоб сигналізувати, що юзер вже натиснув кнопку входу і дія ще не завершилася.

Лістинг методів класу SignUpPageViewModel:

#region Private Methods

void changePage(object parametr)

{

ApplicationService.ChangeCurrentApplicationPage((ApplicationPages)Enum.Parse(typeof(ApplicationPages), (string)parametr, true));

}

/// <summary>

/// Check data and registre user

/// </summary>

/// <param name="param">Password, RepeatPassword</param>

/// <returns></returns>

async Task signUpAsync(object param)

{

//If user few times chicks Sign Up, but the server does not response yet

if (SignUpIsRunning)

return;

//Get secure strings from passwrod boxes

var pass = (param as IHavePasswords).StringPassword;

var repeatPass = (param as IHavePasswords).RepeatStringPassword;

if (!FieldChecker(new SecureString[] { pass, repeatPass}))

return;

//Registration

try

{

SignUpIsRunning = true;

var res = await UnitOfWork.SignUp(new CommonLibs.Data.User(UserName, pass, Email, null));

if (res)

//If all right open SignIn pag

changePage("SignInPage");

ErrorMessage = "Something went wrong";

}

catch {

ErrorMessage = "Something went wrong";

}

finally

{

SignUpIsRunning = false;

}

}

#endregion

В кінці ідуть дві функції, які ми прив’язуємо до щойно розглянутих команд.

Для зберігання такої інформації, як поточна сторінка, ми використовуємо окремий клас-сервіс. Головна причина такого рішення полягає в тому, що до деякої інформації потрібно дістатися з різних частин програми. У лістингу функцій наочним прикладом може слугувати функція **ChangePage**, яка отримує змінює поточну відкриту сторінку за допомогою **ApplicationService**, який зберігає найголовнішу інформацію нашої програми.

Для кращого розуміння того, як відбувається робота з даними, ми додатково розглянемо **ChatPageViewModel**

Лістинг публічних властивостей класу **ChatPageViewModel**:

#region Public Members

/// <summary>

/// store сurrent Choosen page and eth.

/// </summary>

public ChatViewModel ChatViewModel { get; set; }=ApplicationService.GetChatViewModel;

public ChatListViewModel ChatList { get; set; } = new ChatListViewModel();

/// <summary>

/// Text to search

/// </summary>

public string SearchRequest {

get { return searchRequest; }

set {

if (value != null)

{

searchRequest = value;

Search(value);

}

}

}

#endregion

З лістингу ми можемо побачити, що тут використовується три властивості. **ChatViewModel** відповідає за елемент чату який відображається в правій частині додатку. Це може бути інформація юзера, чата, вікно чата або результати пошуку. Ця властивість підв’язується до об’єкту **Frame**, який знаходиться в файлі **ChatPage.xaml**. Далі йде **ChatList** – це ViewModel, який відповідає за список чатів, що знаходиться одразу ж під пошуком. І остання властивість відповідає за результати пошуку.

Всі дані, які надходять з серверу, автоматично завантажуються. Це відбувається за допомогою декількох методів та подій зміни у базі даних.

Лістинг обробників подій в класі ChatPageViewModel:

/// <summary>

/// Updates groups

/// </summary>

/// <param name="dataChanged"></param>

void UpdateGroupInChatList(List<Group> dataChanged)

{

if (dataChanged == null)

return;

//For each updated group

foreach (var data in dataChanged)

{

//Find the group with same Id

for (int i = 0; i < ChatList.Items.Count; i++)

{

//And Replace

if (ChatList.Items[i].GroupData.Id == data.Id)

{

//Becouse Items is ObservableCollection we should update elements from the main thread

App.Current.Dispatcher.Invoke(() =>

{

ChatList.Items[i] = new ChatListItemViewModel(data, ChatList.Items[i].LastMessage);

});

break;

}

}

}

}

/// <summary>

/// Deletes groups

/// </summary>

/// <param name="dataChanged"></param>

void RemoveGroupFromChatList(List<Group> dataChanged)

{

if (dataChanged == null)

return;

//For each updated group

foreach (var data in dataChanged)

{

//Find the group with same Id

foreach (var item in ChatList.Items)

{

//And Delete

if (item.GroupData.Id == data.Id)

{

//Becouse Items is ObservableCollection we should remove elements from the main thread

App.Current.Dispatcher.Invoke(() =>

{

ChatList.Items.Remove(item);

});

break;

}

}

}

}

/// <summary>

/// Adds new element to chat list

/// </summary>

/// <param name="dataChanged"></param>

void AddGroupToChatList(List<Group> dataChanged)

{

if (dataChanged == null)

return;

foreach (var group in dataChanged)

{

//Get last message from dataBase

var lastMess = UnitOfWork.Database.MessagesTableRepo.FindLast(MessagesTableFields.ReceiverId.ToString(), group.Id.ToString());

//Becouse Items is ObservableCollection we should remove elements from the main thread

App.Current.Dispatcher.Invoke(() =>

{

ChatList.Items.Add(new ChatListItemViewModel(group, lastMess));

});

//Sort list by the last message time

ChatList.SortChatList();

}

}

При зміні записів, ми звертаємося до функцій і в залежності від виконаної над даними дії ми тим чи іншим методом оновлюємо інтерфейс. Якщо в базу даних додати декілька нових груп, то спрацює метод **AddGroupChatList**, який дістане останнє повідомлення пов’язане з групою, після чого додасть новий чат у список. Оновивши декілька існуючих груп ми викличемо метод **UpdateGroupInChatList**, що перевірить усі чати зі списку та оновить їх. Схожим чином оновлюються усі повідомлення, сторінки з інформацією юзерів та груп.

**Сервер**

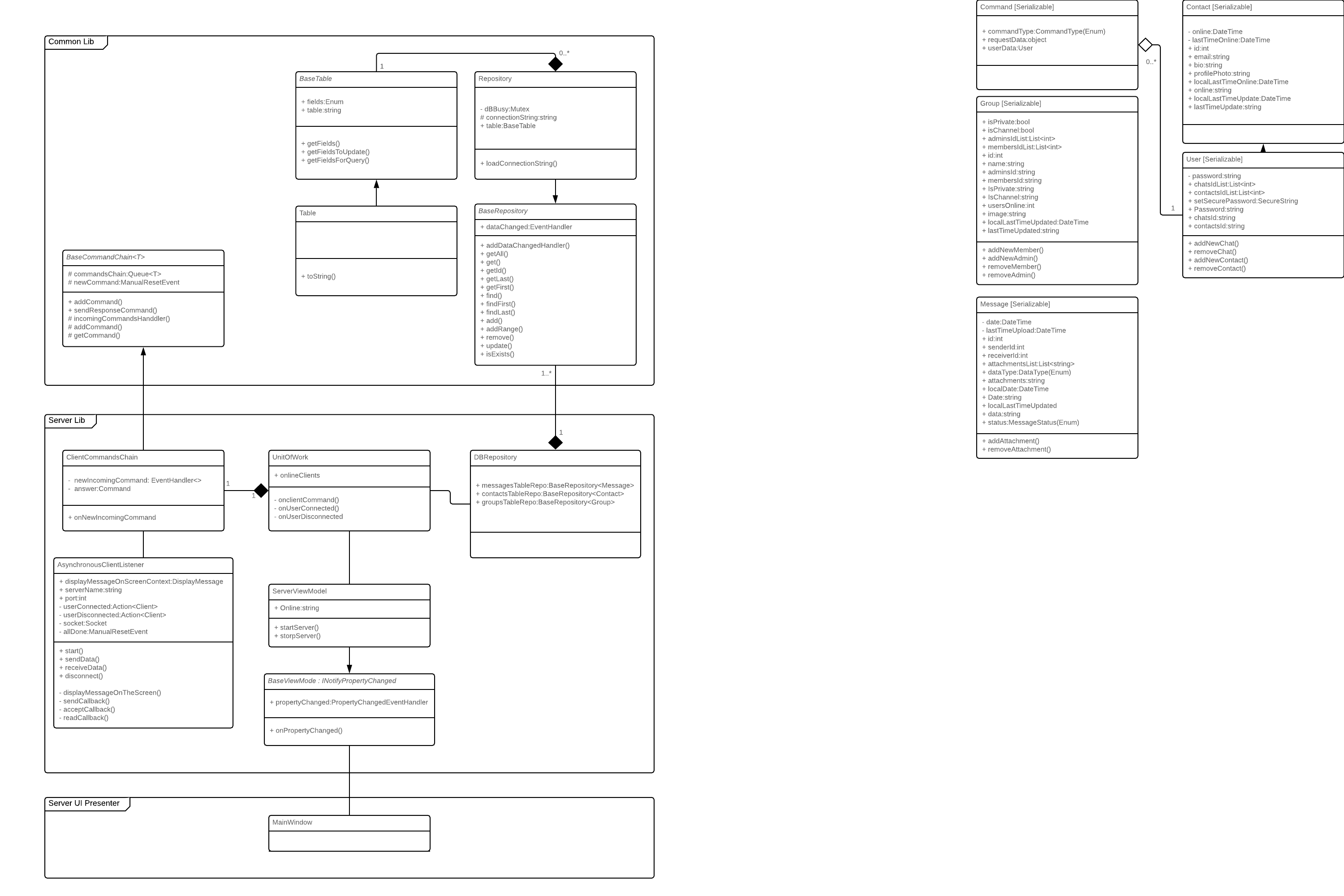


Рис. 2.14 Діаграма класів серверного додатку

Діаграма серверної частини дуже схожа на клієнтський додаток, але з деякими відмінностями, які ми зараз розглянемо.

Найголовніші зміни відносяться до класу **AsynchronousClientListener**. Для того, щоб ми могли одночасно під’єднувати декілька користувачів, потрібно створити ще один допоміжний клас. У нашому випадку це буде клас Client.

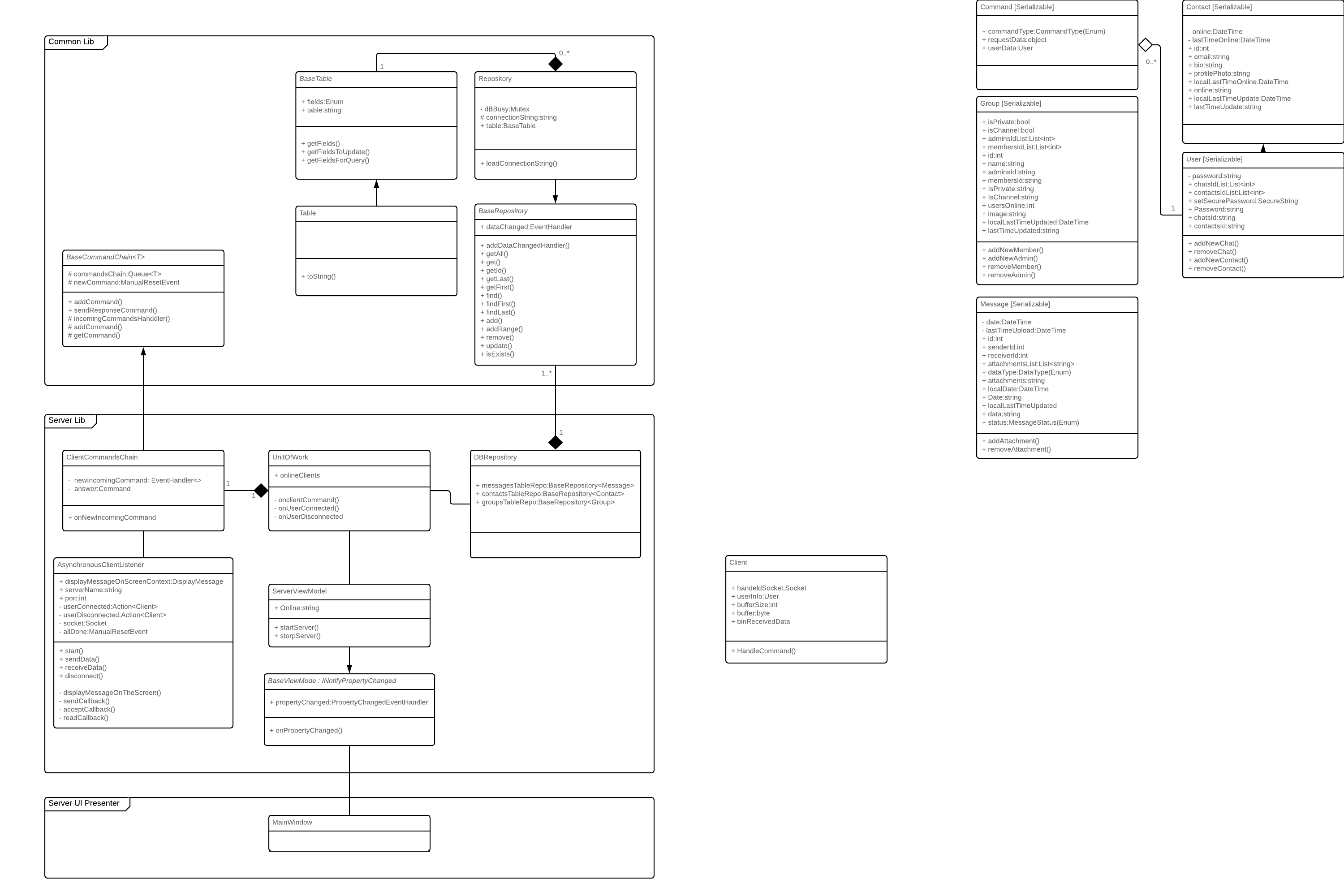


Рис. 2.15 Схема класу Client

Для кожного юзера, який під’єднується до серверу, створюється окремий екземпляр цього класу, куди передається оброблений сокет. Це потрібно для того щоб ми могли надсилати команди у відповідь.

Лістинг методу AcceptCallback класу AsynchronousClientListener:

static private void AcceptCallBack(IAsyncResult ar)

{

// Signal the main thread to continue

allDone.Set();

displayMessageOnScreen("New connection");

// Get socket that handles current client request

Socket handler = socket.EndAccept(ar);

// Create client object that will provide command handler

Client client = new Client(handler);

//Add new client to list of connected clients

UserConnected?.Invoke(client);

// Begin recieve data

handler.BeginReceive(client.Buffer, 0, Client.BufferSize, 0, new AsyncCallback(ReadCallback), client);

}

В момент коли команда потрапляє на сервер, вона передається в клас **ServerCommandChain**, після чого потрапляє в **UnitOfWork**, а саме в метод **OnClientChain**, де взалежності від типу обробляється. Усім іншим Сервер схожий на клієнтський додаток.

**База даних**

Для збереження даних ми використовуємо базу даних **SQLite** – вона достатньо функціональна та проста в застосуванні.

На сервері зберігаються усі повідомлення, створенні юзери, групи, а також файли. При підключенні вся інформація синхронізується з клієнтською базою даних.



Рис. 2.16 Схема серверної бази даних

В кожного юзера є набір інформації, яку маже переглянути кожен – це **Bio**, **Email**, **ProfilePhoto**, **UserName**, **ChatsId**, **Online**. Користувач може створювати та приєднуватися до груп, при цьому вони додаються до **ChatsId**, також користувач може вирішити додати когось у список контактів, які зберігаються в **ContactsId**. При зміні інформації користувача, групи або повідомлення змінюється час останньої зміни – **LastTimeUpdated**, що допомагає синхронізувати інформацію на клієнті. Усі дати зберігаються, попередньо конвертуючись у світовий час, а потім у бінарне представлення. Якщо сервер буде в іншому часовому поясі, то це не буде впливати на представлення часу у клієнта. Таблиця з групами зберігає інформацію про тип групи, доступність та приєднаних користувачів. Приватні групи не видно під час пошукую. А в каналах можуть писати повідомлення лише адміни. Повідомлення зберігає id відправника та отримувача. Відправник – це завжди юзер, у той час коли отримувач – це група. Також тут інформація про тип даних (текст, файл або зображення) час відправлення та статус: відправлено, доставлено та прочитано.

Аналогічно виглядає і клієнтська база даних, за винятком того, що замість таблиці користувачів тут знаходиться таблиця контактів, які зберігає загальнодоступну інформацію.



Рис. 2.17 Схема клієнтської бази даних

# РОЗДІЛ 3 ОПИС РОБОТИ З ПРОГРАМНИМ ДОДАТКОМ ТА ЙОГО ТЕСТУВАННЯ

## 3.1 Опис роботи з програмним додатком

Клієнт:

Запустивши додаток, ми можемо побачити головне вікно з полями для введення логіна та пароля. В верхньому лівому кутку знаходиться статус під’єднання до серверу. В нашому випадку сервер під’єднано.

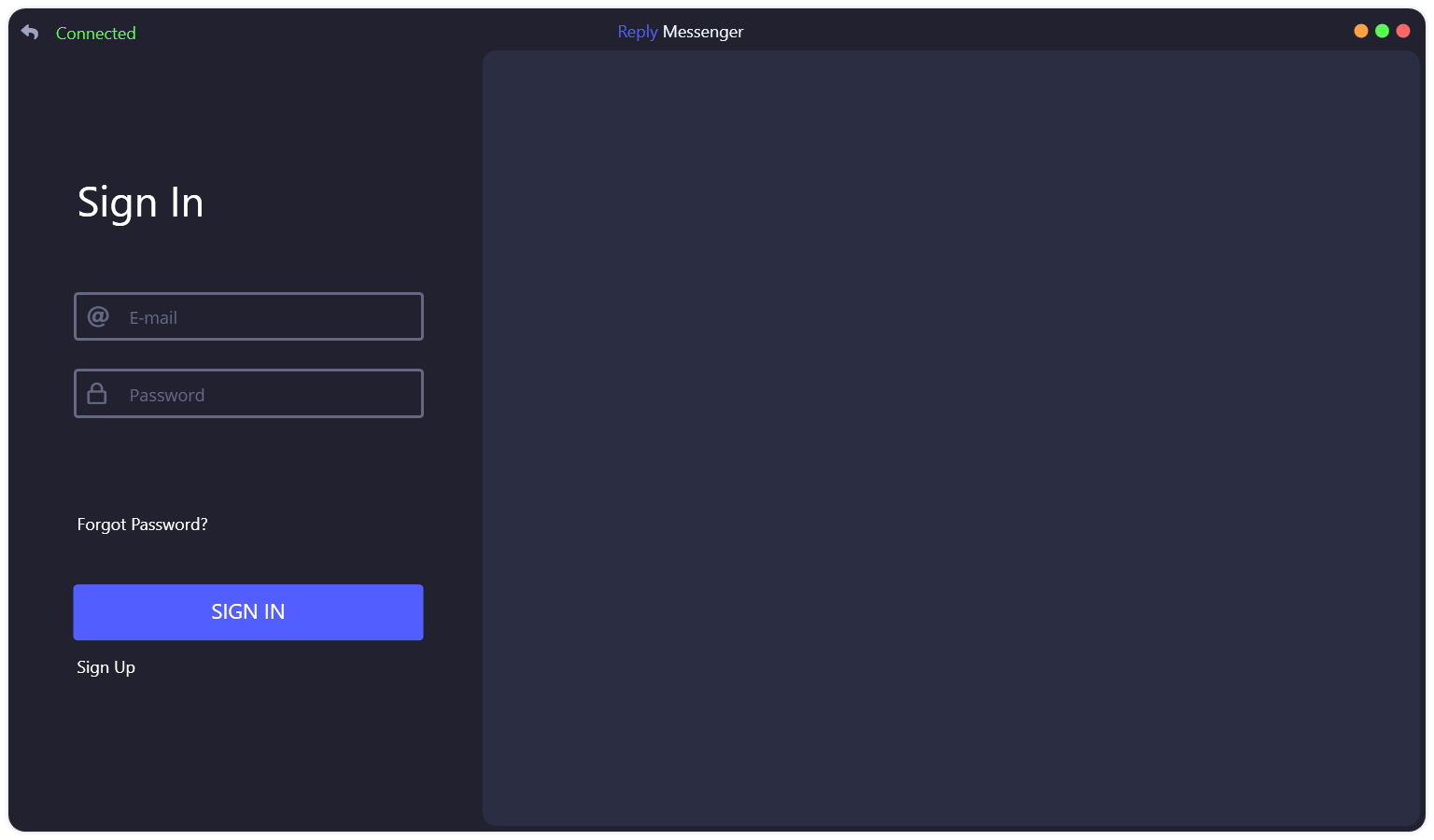


Рис. 3.1. Вікно входу

Якщо ви вже маєте акаунт, то потрібно просто ввести свою пошту та пароль, в іншому випадку натисніть кнопку Sign Up, що знаходиться одразу ж під синьою кнопкою входу.

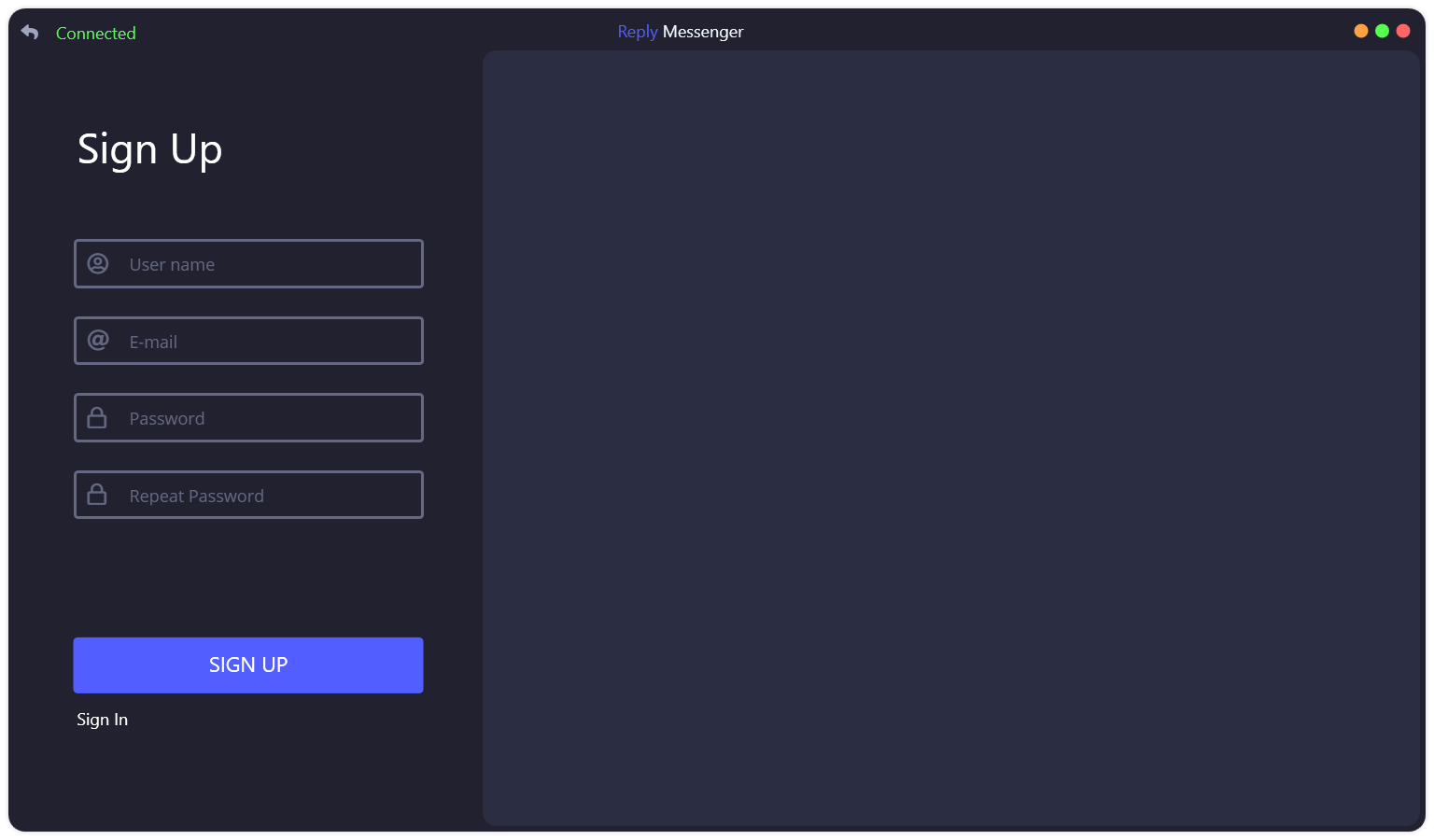


Рис. 3.2. Вікно реєстрації

У вікні реєстрації можна створити новий акаунт заповнивши усю необхідеу інформацію. Потрібно ввести логін, під яким вас будуть бачити інші користувачі, пошту, яку також можна буде побачити, пароль та його повторення. Після цього натисніть SIGN UP, щоб зареєструватися, якщо були введені некоректні дані, то ви побачите повідомлення про помилку, а також поля в яких буди введені дані змінять колір на червоний. У випадку, коли все пройде успішно, вас знову перекине у вікно входу (див. рис. 3.1). Введіть дані під якими ви щойно зареєструвалися, після чого натисніть кнопку SIGN IN. Якщо ви введете правильні дані, то вас відкриється вікно з чатами та вашим профілем.

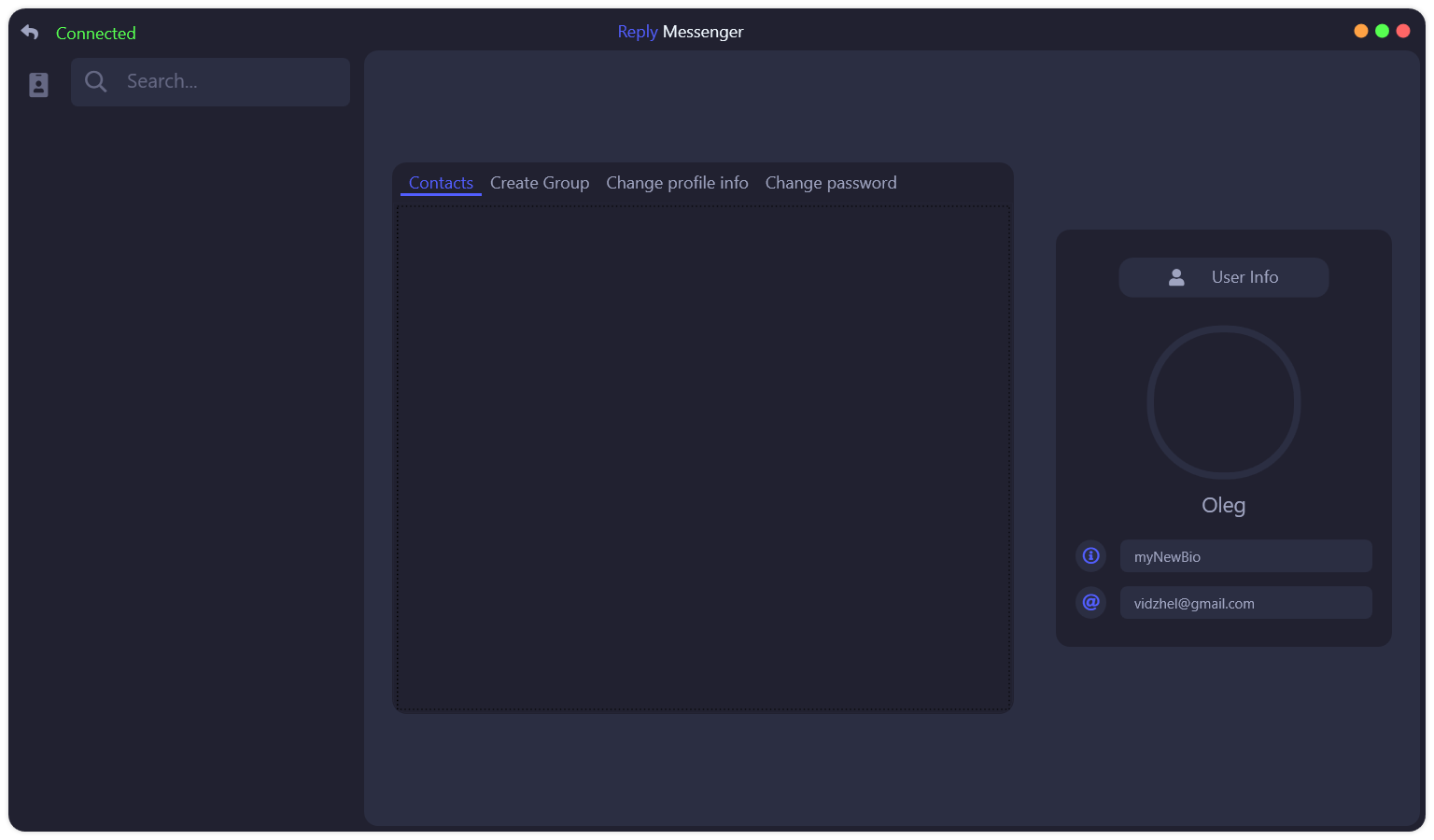


Рис. 3.3. Головне вікно чату

З лівого боку ми можемо побачити значок профіля – це кнопка, яка перемістить вас у ваш профіль. Також біля ми можемо побачити пошук. Через пошук ми можемо знайти користувача, або публічну групу. Приватні групи в пошуку не з’являються.

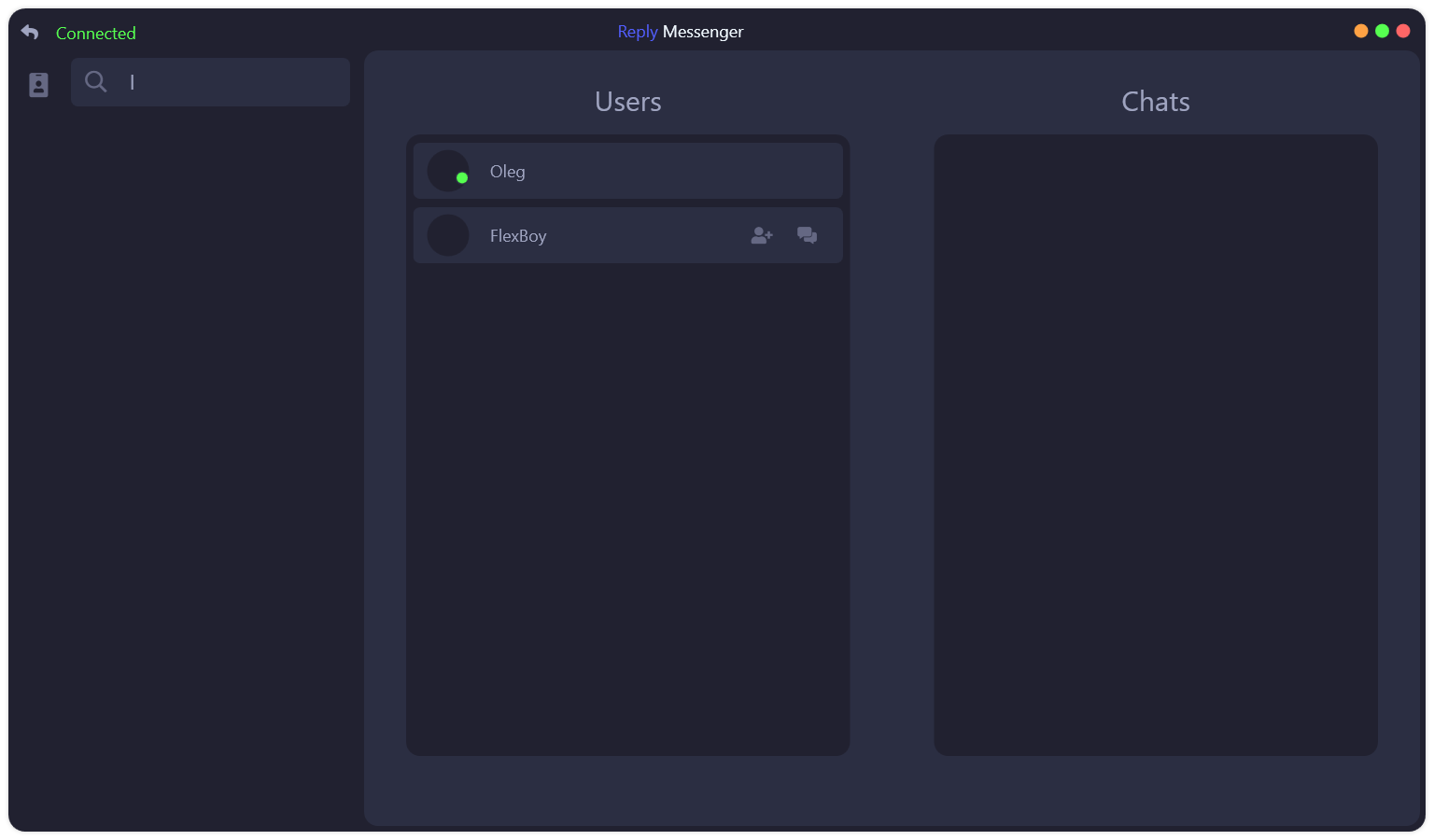


Рис. 3.4. Пошук груп та користувачів

В пошук введемо якусь літеру. Ми бачимо, що з’явилося декілька юзерів. Перший це ви, наступний це інший користувач. Ми можемо побачити дві іконки біля незнайомого користувача. Якщо наведемо на них курсор, то зможемо прочитати, що вони роблять. Перша додає користувача в список контактів, друга створює новий час з користувачем.

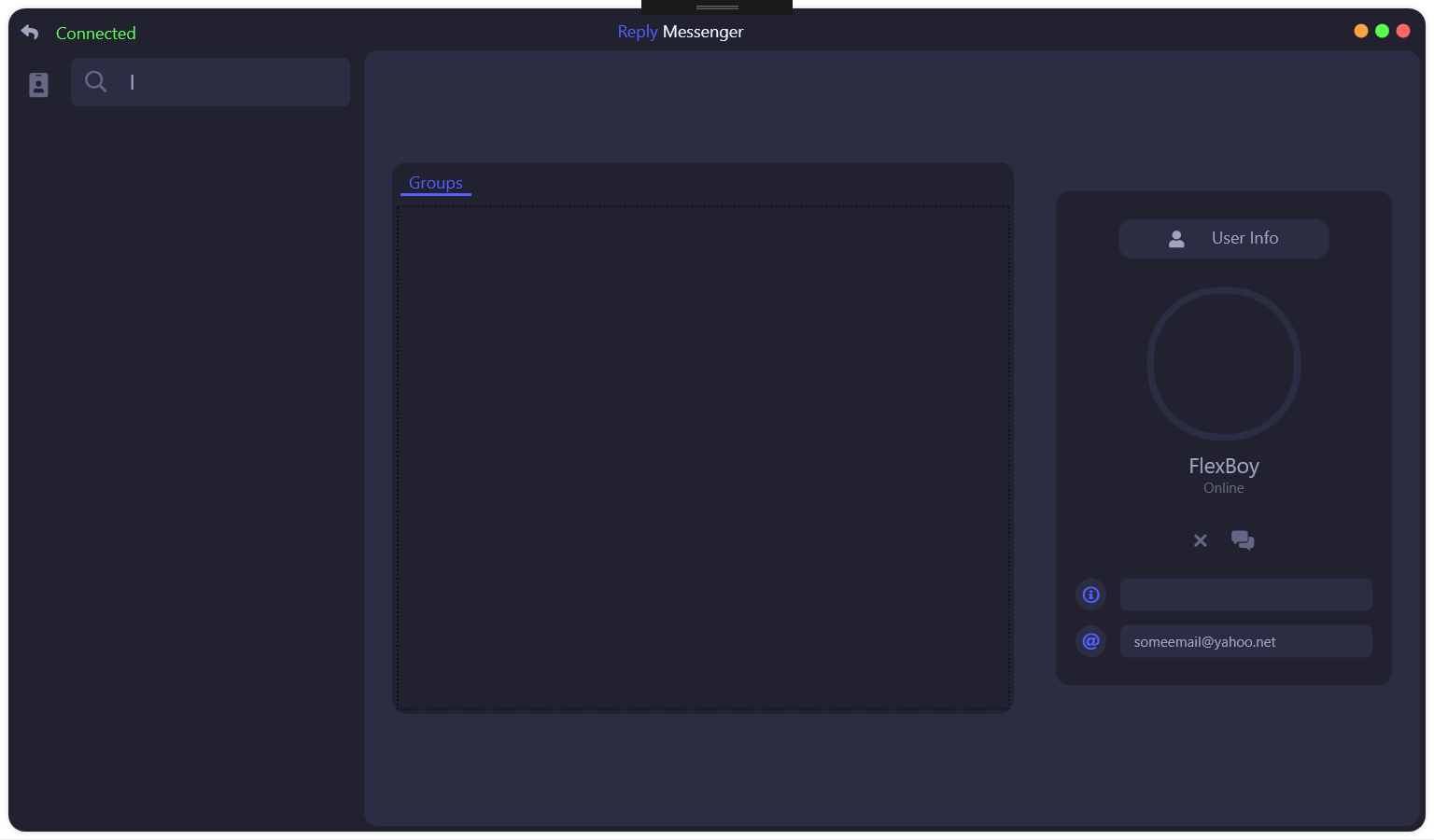


Рис. 3.5. Інформаційна сторінка користувача

Додамо користувача в список контакті, після цього перейдемо в свій профіль і побачимо, що він там з’явився. Тепер ми можемо видалити його з списку контактів, якщо захочемо. Якщо натиснути на користувача, то перейдемо в його профіль, де побачимо його публічні групи, пошту, ім’я та біо.

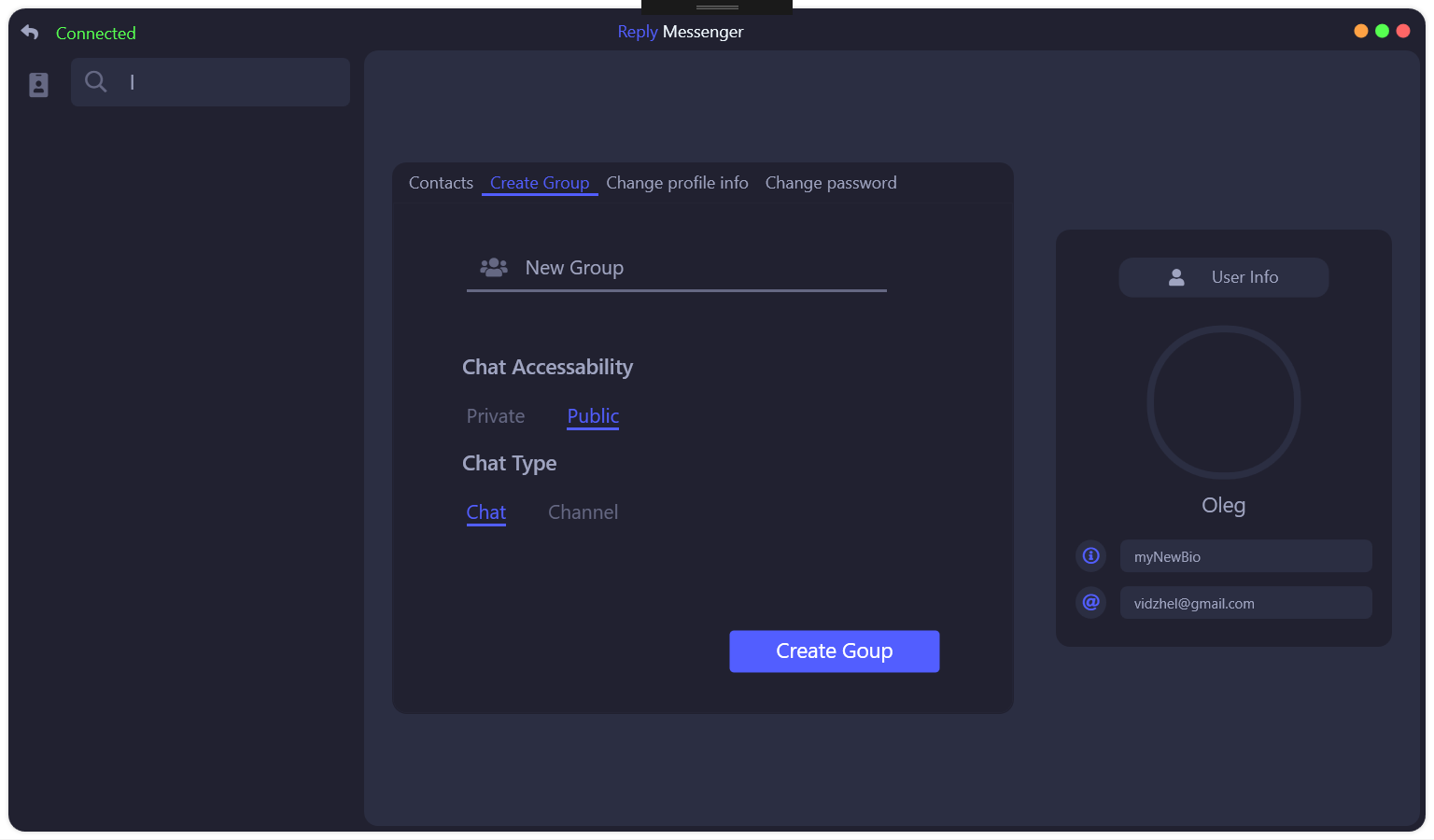


Рис. 3.6. Створення групи

Перейемо назад в свій профіль та створимо новий чат. Для цього потрібно заповнити поле з назвою, а також вибрати доступність чату і його тип. Якщо встановити тип на канал, то писати повідомлення зможуть лише адміни.

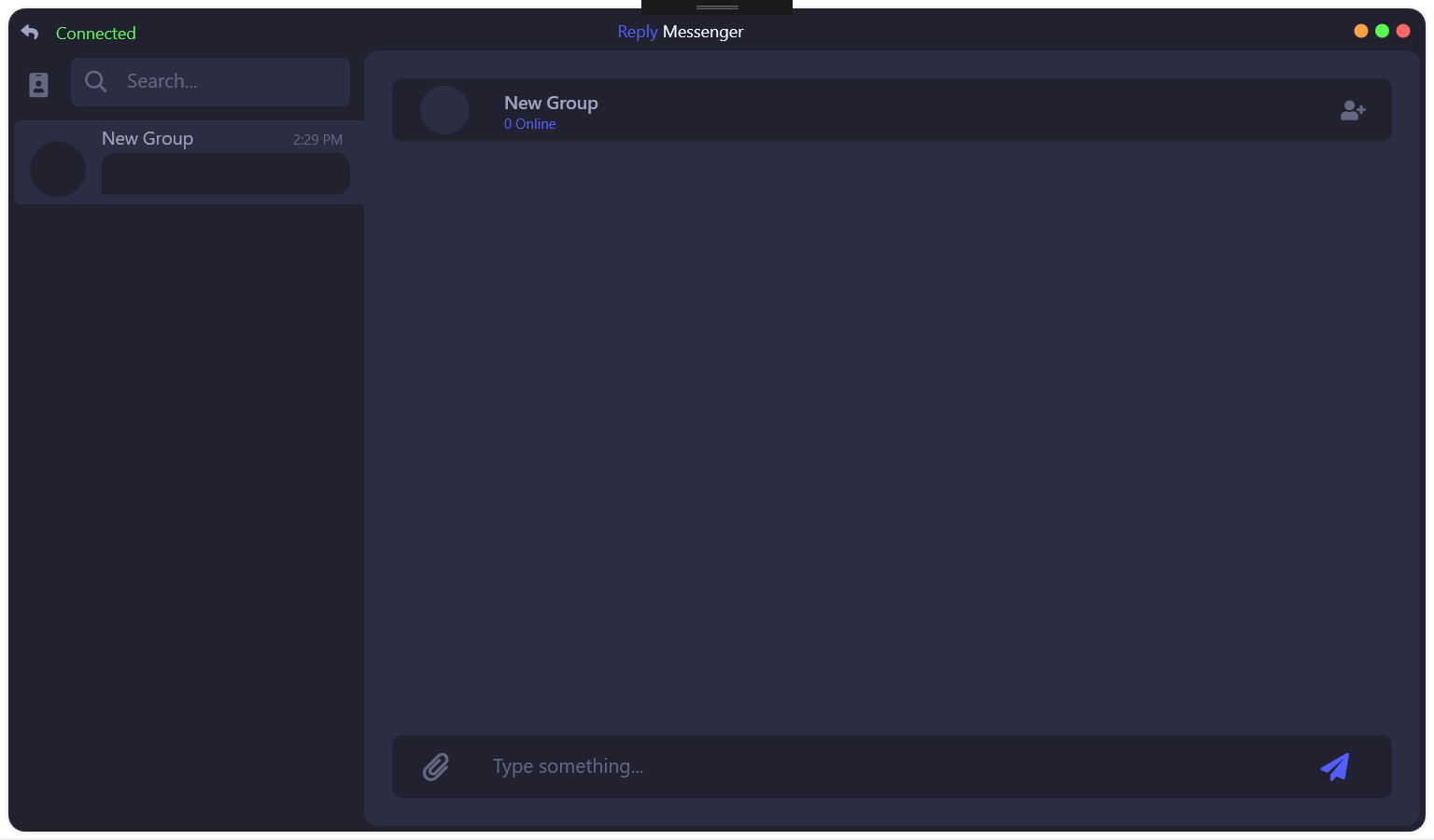


Рис. 3.7. Вікно чату

Після натиснення кнопки Create Group нас закине у вікно чату, де можемо писати повідомлення.

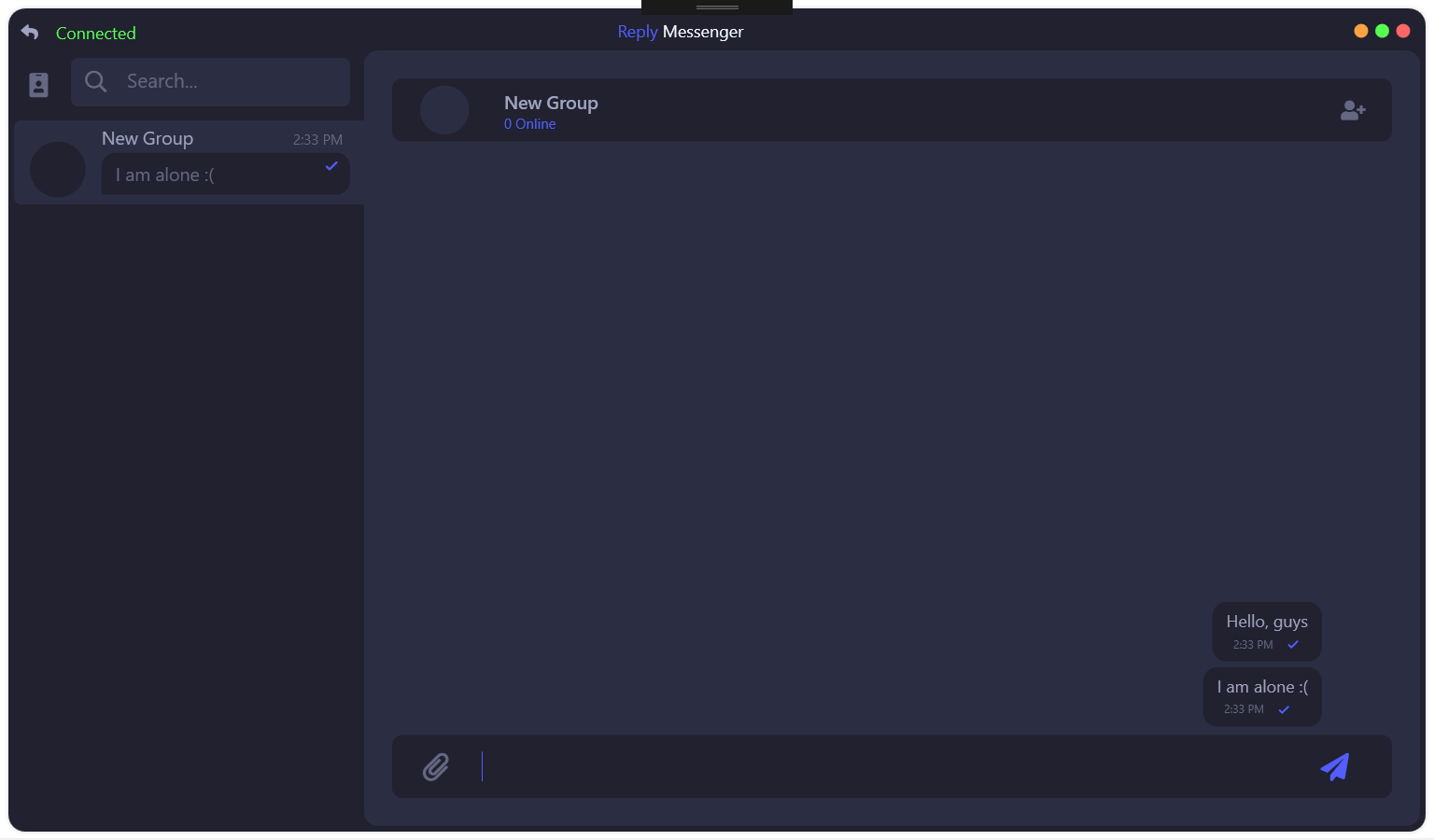


Рис. 3.8. Написання повідомлень

Також за необхідності можемо прикріпити файл. Це може бути і фонографія.

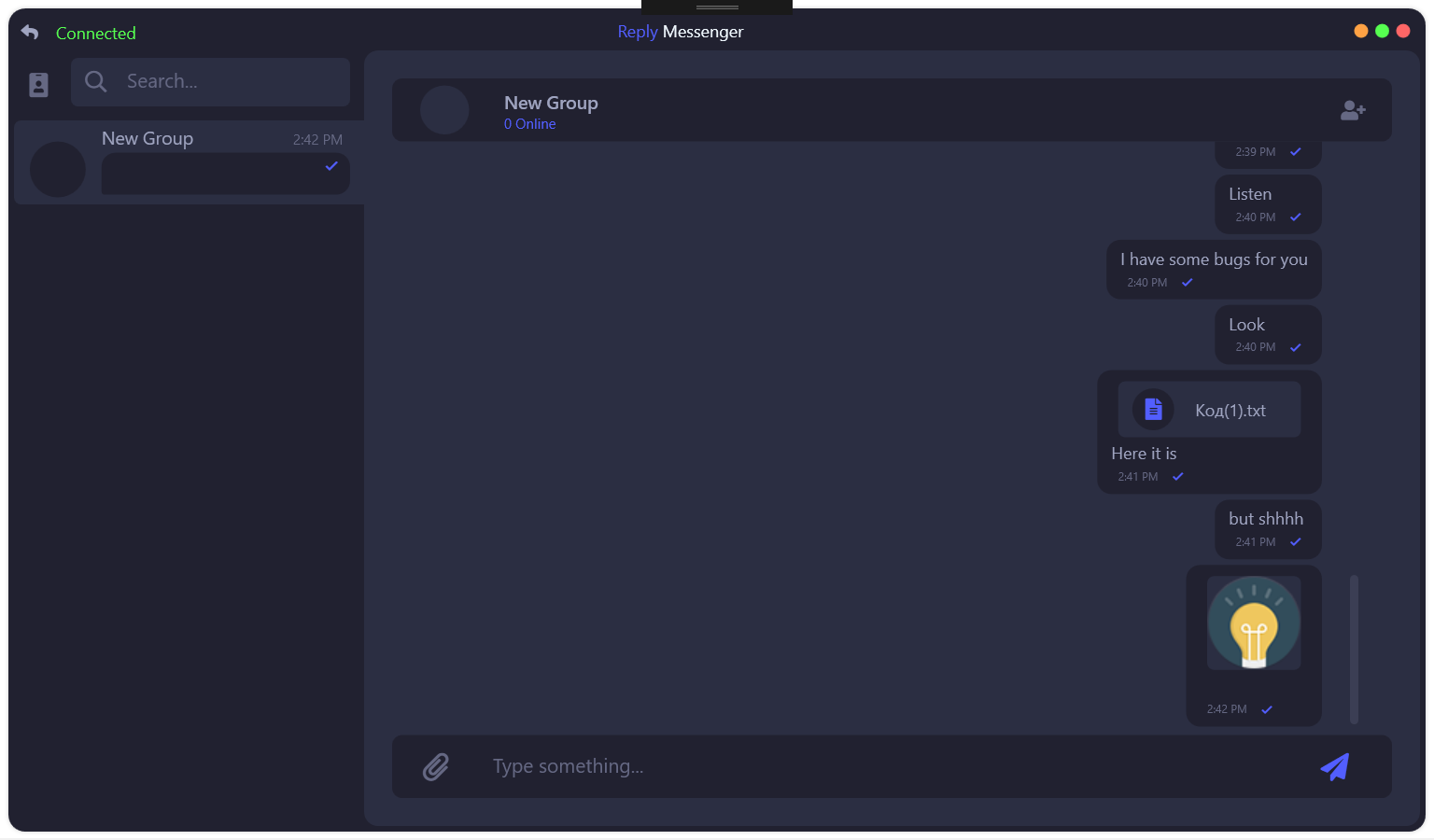


Рис. 3.9. Додавання файлів до повідомлень

Якщо натиснути на файл, то він відкриється. Щоб перейти в інформацію про групу потрібно натиснути на круг біля назви, або у списку груп.

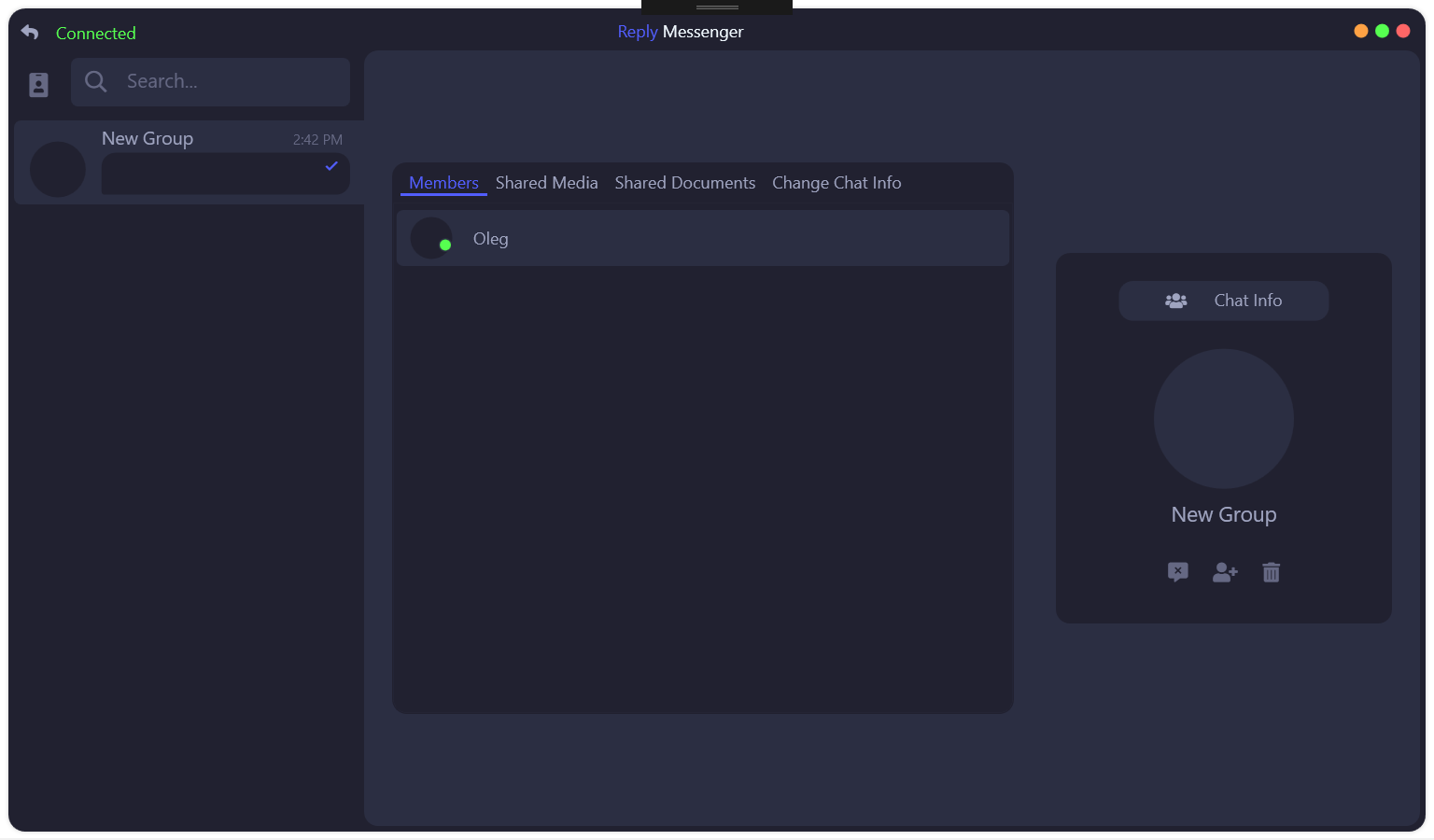


Рис. 3.10. Інформаційна сторінка групи

Тут можемо побачити усіх приєднаних до групи користувачі. Якщо ви адмін, то у вас буде можливість видаляти членів групи, змінювати інформацію групи, а також видалити її. Справа під назвою є три кнопки. Перша видаляє вас з групи, друга відкриває список, щоб запросити когось з списку контактів, а третя видаляє чат. Давайте редагуємо інформацію групи. Для цього перейдіть у вкладку Change Chat Info.

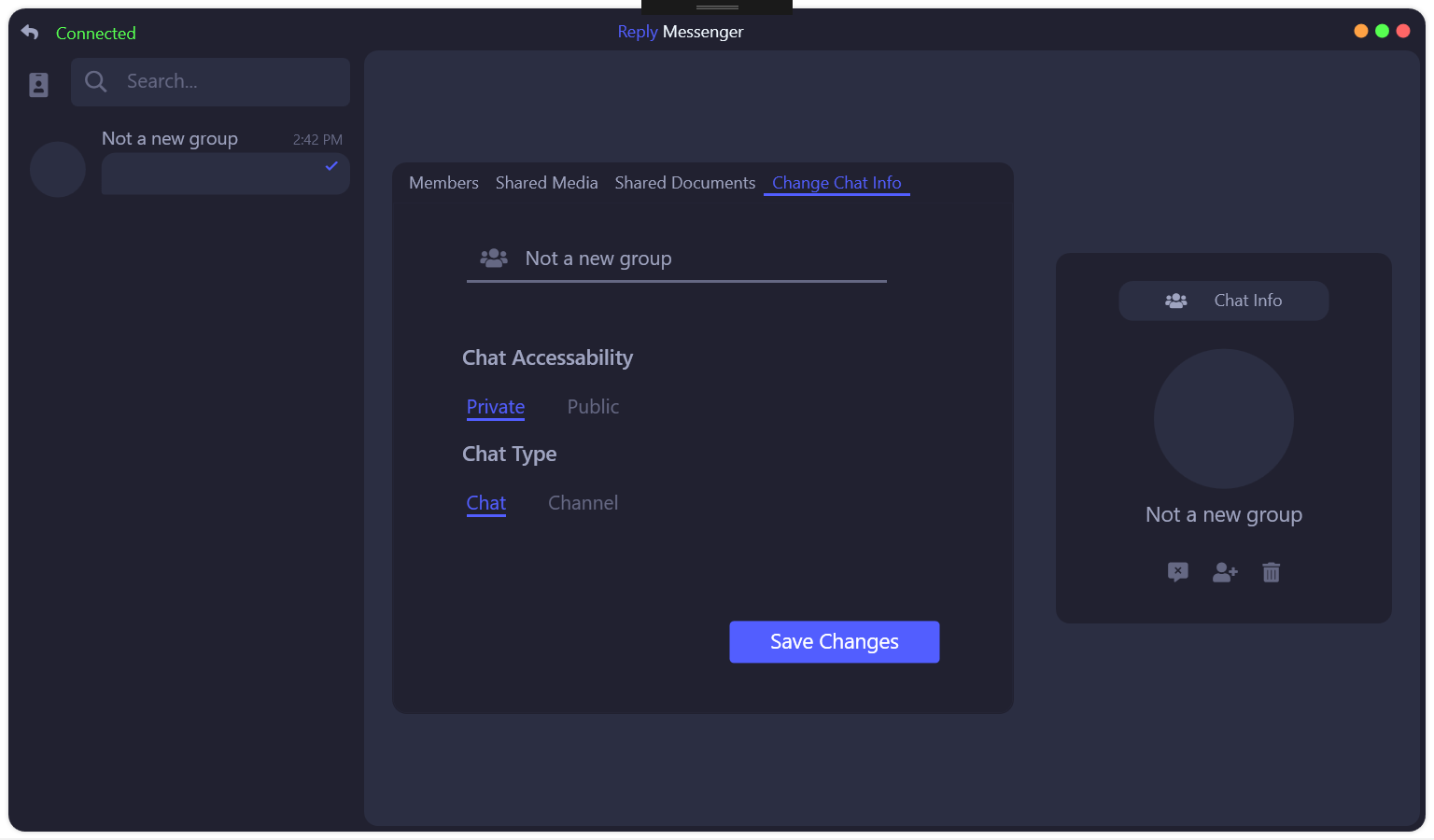


Рис. 3.11. Налаштування групи

Після натискання на кнопки Save Changes інформація зберігається та оновлюється. Як і з групою, можемо змінювати свою інформацію. Для цього перейдемо в свій профіль та відкриємо вкладку Change Profile Info. Тут можемо вказати інформацію, яку хочемо замінити. Якщо залишити поле пустим, то параметр не буде зміюватися.

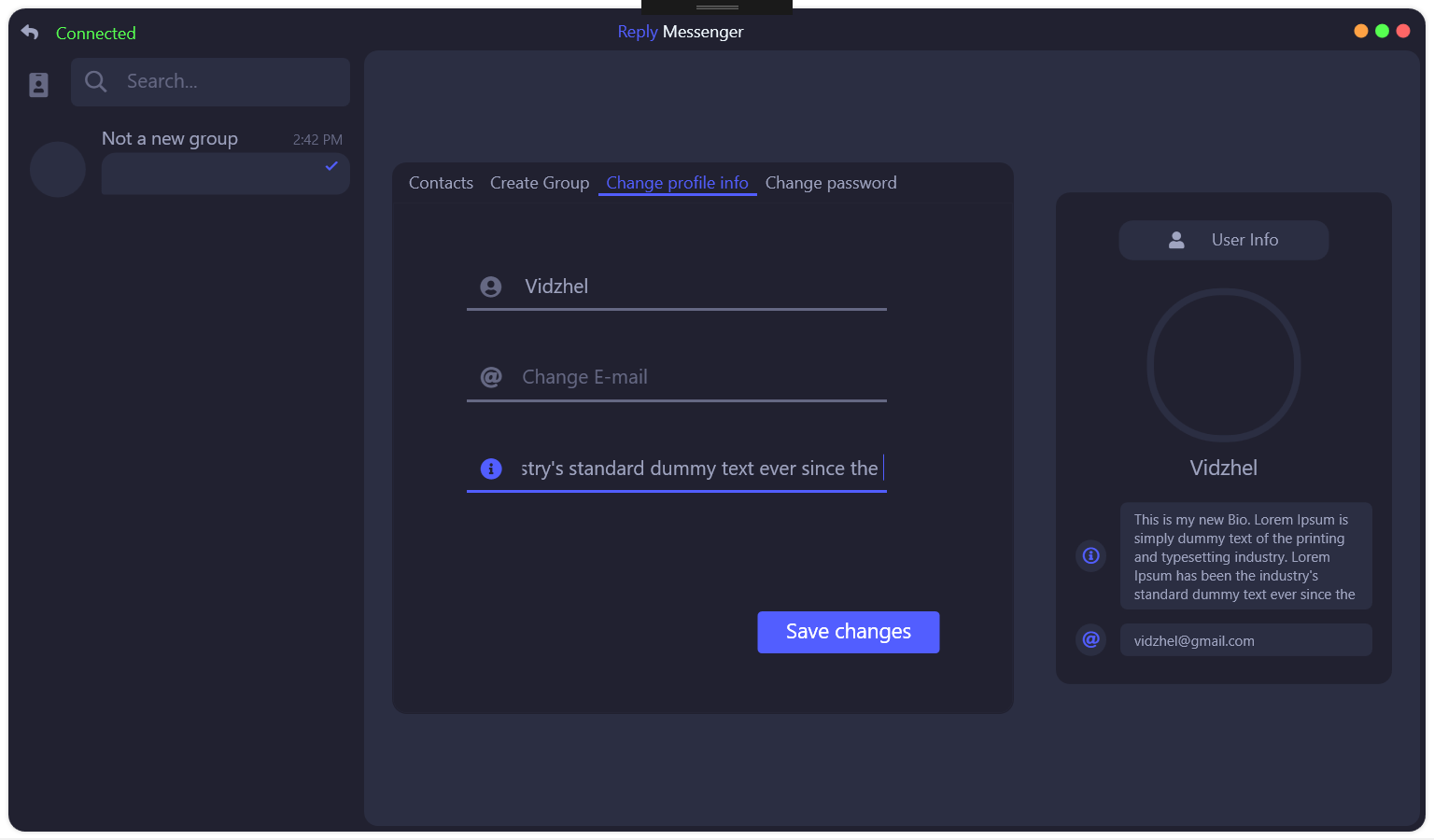


Рис. 3.12. Зміна публічної інформації користувача

Після збереження інформації, вона автоматично оновлюється. У вкладці Change password можемо змінити пароль

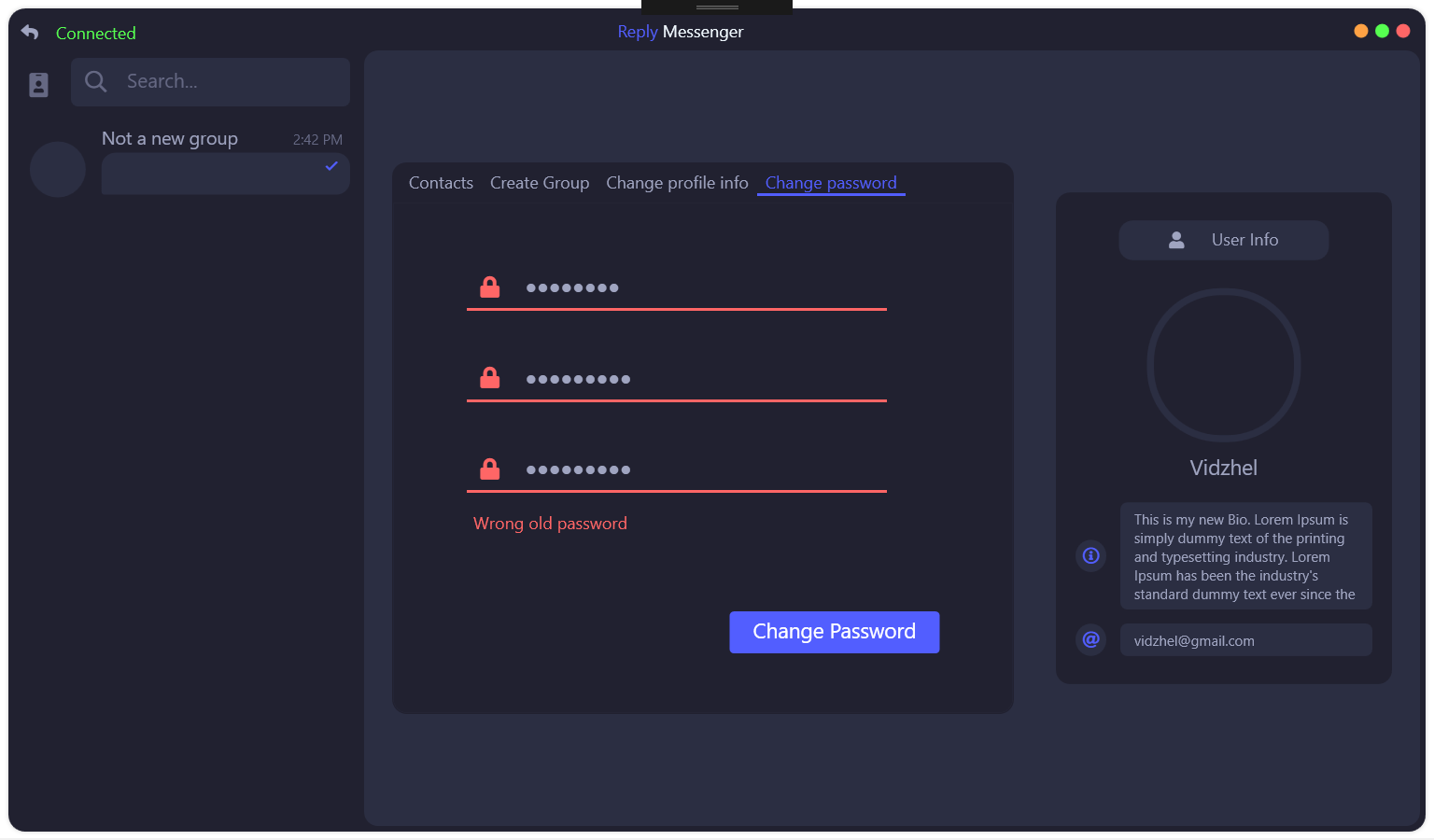


Рис. 3.13. Зміна пароля користувача

Щоб змінити пароль потрібно спочатку ввести старий, а потім два рази новий. Якщо виникне помилка, во буде виведена на екран. Після цього ви готові користуватися месенджером

Сервер:

Сервер має одне велике вікно для перегляду логів, а також дві кнопки для запуску та зупинки сервера.

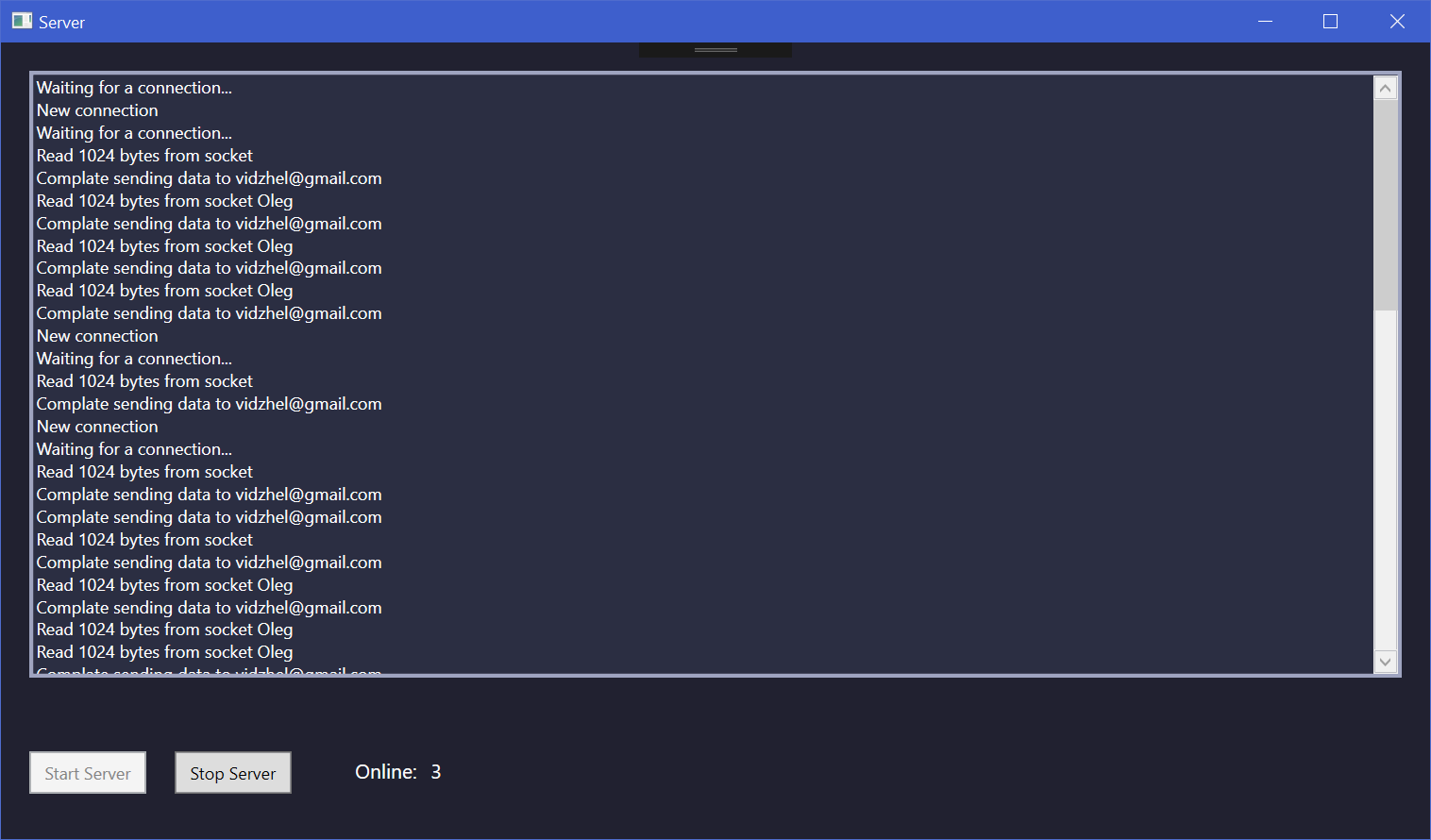


Рис. 3.14. Головне вікно сервера

## 3.2 Тестування роботи програмного забезпечення

Настав час для тестування. Для цього будемо використовувати метод функціонального тестування: виявлення невідповідностей між реальною поведінкою реалізованих функцій і очікуваною поведінкою відповідно до специфікації і вимог. В ТЗ (див. ст. 9) зазначили список функцій, які мають працювати в програмі:

* Реєстрація нових користувачів (збереження даних в базу даних);
* Вхід користувачів в систему;
* Синхронізація даних з сервером;
* Обмін повідомленнями, у тому числі і файлами або зображеннями, з зареєстрованими користувачами;
* Створення публічних та приватних груп з можливостями додавання, видалення користувачів.

Таблиця 3.1

**Тестування програмного додатку**

|  |  |  |
| --- | --- | --- |
| № | Функція | Успішність |
| 1 | Реєстрація користувачів | + |
| 2 | Вхід в акаунт | + |
| 3 | Синхронізація диних з сервером | + |
| 4 | Обмін повідомленнями між користувачами | + |
| 5 | Створення груп | + |
| 6 | Видалення груп | + |
| 7 | Під’єднання до груп | + |
| 8 | Додавання користувачів у список контактів | + |
| 9 | Видалення користувачів з списку контактів | + |
| 10 | Обмін файлами та зображеннями | + |
| 11 | Зміна налаштувань групи | + |
| 12 | Зміна інформації користувача | + |

# ВИСНОВКИ

В цьому курсовому проекті було досліджено алгоритм створення програмного додатку для обміну повідомленнями по локальній мережі. Месенджер – програма для миттєвого обміну повідомленнями (в тому числі файлами, зображеннями, голосовими повідомленнями та іноді дзвінками). Це означає, що потрібно будь-яким чином з’єднати користувачів між собою в одній системі.

Уся робота над додатком розділялась на декілька частин:

* Аналіз роботи програми, функцій які будуть присутні в програмі;
* Розробка структури коду (побудова діаграм та блок-схем до кожної; функції з попереднього пункту). Розробка класів та схем наслідування (у тому числі і для серверної частини);
* Створення дизайну програми в графічному редакторі;
* Написання основного коду та перевірка функціонування;
* Написання графічної частини програми WPF з використанням XAML.
* Налагодження коду;

У першому розділі визначені підстави для розробки курсового проекту, більш детально описані основні поняття.

У другому розділі була детально спроектована основна схема функціонування програмного продукту. Розроблено візуальну складову програми та систему взаємодії з користувачем.

У третьому розділі було проведено тестування, та виправленні деякі помилки які були виявлені у проекті, а також детально дана інструкція користування програмним продуктом.

Програмний додаток реалізований з використанням мови C# та WPF.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Michael A. Nielsen, "Neural Networks and Deep Learning", Determination Press, 2015

# ДОДАТКИ

Додаток А