Хмельницький національний університет
Факультет програмування
та комп'ютерних і телекомунікаційних систем
Кафедра інженерії програмного забезпечення

ДИПЛОМНИЙ ПРОЕКТ

Мобільний додаток для організації медіа-файлів та їх автоматичного резервного копіювання у хмару з використанням API Telegram

Назва теми

Рівень вищої освіти Перший (бакалаврський)

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

Шифр ДПІПЗ.170106.01.06.ПЗ

Виконав студент IV курсу група ІПЗ-17-1 _____ С. О. Капітанець
Підпис Ініціали, прізвище

Керівник канд. техн. наук, доцент _____ Г. І. Радельчук
Науковий ступінь, звання Підпис Ініціали, прізвище

Нормоконтролер канд. техн. наук, доцент _____ Г. І. Радельчук
Підпис Ініціали, прізвище

До захисту допускаю:
Завідувач кафедри інженерії
програмного забезпечення _____ Л. П. Бедратюк
Підпис Ініціали, прізвище

15 06 2021 р.

Хмельницький 2021

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Програмування та комп'ютерних і телекомунікаційних систем
Кафедра Інженерії програмного забезпечення
Рівень вищої освіти Перший (бакалаврський)
Галузь знань 12 «Інформаційні технології»
Спеціальність 121 «Інженерія програмного забезпечення»
Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Л. П. Бедратюк
05 02 2021 р.

ЗАВДАННЯ
НА ДИПЛОМНИЙ ПРОЕКТ (РОБОТУ)

Капітанцю Степану Олександровичу
Прізвище, ім'я, по батькові студента

1. Тема проекту (роботи) Мобільний додаток для організації медіа-файлів та їх автоматичного резервного копіювання у хмару з використанням API Telegram

Керівник проекту (роботи) Радельчук Галина Іванівна, канд. техн. наук, доцент
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 05.02.2021 р. № 11
2. Строк подання студентом проекту (роботи) на кафедру 01.06.2021 р.
3. Вихідні дані до проекту (роботи) Матеріали переддипломної практики

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити) Дослідження предметної області та постановка задачі, проектування програмного забезпечення, програмна реалізація, тестування програмного забезпечення

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень) Презентаційні матеріали (слайди, 18 шт.)

6. Консультанти розділів дипломного проекту

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання пр
Нормоконтроль	Радельчук Г. І., доцент кафедри ІПЗ		
Антиплагіат	Гурман І. В., доцент кафедри ІПЗ		

7. Дата видачі завдання « 05 » лютого 2021р.

КАЛЕНДАРНИЙ ПЛАН

Назва етапів (розділів) дипломного проекту (роботи)	Строк виконання етапів проекту (роботи)	Приміт
1 Ознайомлення з тематикою дипломного проектування (ДП), визначення та узгодження індивідуальних там ДП	01.12 – 30.12.2020	
2 Дослідження предметної області, в якій планується використання програмного засобу (ПЗ), визначення задач та вимог, розробка технічного завдання	02.01 – 31.01.2021	
3 Проектування програмного забезпечення	01.02 – 28.02.2021	
4 Програмна реалізація	01.03 – 10.04.2021	
5 Тестування програмного забезпечення	11.04 – 30.04.2021	
6 Написання вступу, загальних висновків, оформлення джерел посилання та додатків. Оформлення пояснювальної записки ДП згідно вимог стандартів	01.05 – 25.05.2021	
7 Попередній захист ДП	Травень 2021 (згідно графіка)	
8 Перевірка ДП на плагіат, нормконтроль, отримання відгуків та рецензій. Брошурування (зшиття) пояснювальної записки	26.05 – 30.05.2021	
9 Підготовка до захисту та захист ДП	з 01.06.2021	

Студент _____ С. О. Капітанець
Підпис Ініціали, прізвище

Керівник проекту (роботи) _____ Г. І. Радельчук
Підпис Ініціали, прізвище

АНОТАЦІЯ

Тема дипломного проекту: Мобільний додаток для організації медіа-файлів та їх автоматичного резервного копіювання у хмару з використанням API Telegram.

Автор проекту: Капітанець Степан Олександрович.

Керівник проекту: Радельчук Галина Іванівна.

Пояснювальна записка: 165 с., 21 рис., 6 табл., 4 дод., 20 джерел.

Графічна частина: 18 слайдів.

РЕЗЕРВНЕ КОПІЮВАННЯ, МОБІЛЬНИЙ ДОДАТОК, ХМАРНЕ СХОВИЩЕ, XAMARIN, TELEGRAM.

Метою проекту є розробка мобільної галереї для ОС Android, яка дозволятиме її користувачам в автоматичному та ручному режимі вивантажувати медіа-файли в хмарне сховище Telegram, а також організовувати свою галерею шляхом додавання до них тегів, за якими потім можна буде виконувати пошук.

У дипломному проекті було проаналізовано предметну область, з'ясовані причини частого виникнення проблеми нестачі пам'яті у користувачів смартфонів, проведено аналіз схожого програмного забезпечення, були порівняні архітектурні стилі для розробки мобільних додатків, визначені модулі системи та здійснена їх програмна реалізація.

Для розробки мобільного додатку була використана мова програмування C#, фреймворк Xamarin та база даних SQLite.

В результаті було розроблено мобільний додаток для автоматичного резервного копіювання медіа-файлів у Telegram.

01.06.21
Дата

Підпис

ВІДОМІСТЬ ДОКУМЕНТІВ

№ рядка	Формат	Позначення документа	Найменування документа	К-сть аркушів	№ екз.	Примітка
			Текстові документи			
1	А4	ДПІПЗ.170106.01.06.ПЗ	Пояснювальна записка	165		
2	А4		Завдання на дипломний	1		
			проект			
3	А4		Анотація	1		
			Графічні документи			
4	А4		Презентаційні матеріали	18		

Змн.	Арк.	№ докум.	Підпис	Дата	ДПІПЗ.170106.01.06.ВД		
Виконав		Капітанець С.О.		01.06	Мобільний додаток для організації медіа-файлів та їх автоматичного резервного копіювання у хмару з використанням API Telegram	Літ.	Арк. 1 / Акрушів 1
Керівник		Радельчук Г.І.		14.06			
Н. Контр.		Радельчук Г.І.		14.06		ХНУ, ІПЗ-17-1	
Зав. Каф.		Бедратюк Л.П.		15.06	Відомість документів		

ЗМІСТ

ВСТУП

Мобільна фотографія викликає все менше відрази у професійних фотографів. Можна фотографувати усе підряд або ж зберігати у галереї лише фото лічильників води та газу, але це не відміняє того факту, що сучасні середньобюджетні смартфони навчилися робити знімки, які за своєю якості здатні конкурувати навіть із професійними камерами. Це спонукає все більше людей тягнутися за своїм телефоном, коли вони бачать красивий захід сонця, зустрічаються із друзями чи просто помітили щось, варте уваги. Зберегти момент, щоби потім повернутися до нього та знову відчути ті емоції, ще ніколи не було настільки просто.

Тим не менше, всі користувачі смартфонів рано чи пізно стикаються із проблемою нестачі пам’яті на їхньому пристрої. Це змушує їх обирати, якими додатками і/або медіа-файлами (адже зазвичай саме вони є найбільш вимогливими до сховища) доведеться пожертвувати для відновлення нормальної працездатності смартфону. Видалення лише невдалих дублів іноді може виявитись недостатнім, а тому користувачі, не бажаючи знищувати дорогі для них фото та відео, змушені знову й знову тягнутися за кабелем синхронізації, щоби перемістити медіа-файли на комп’ютер, де вони хоча й збережуться, але вже будуть недоступними зі смартфону. Це дуже незручно, адже в такому випадку не можна буде переглянути або відправити комусь збережений таким чином медіа-файл, якщо тільки ви не знаходитесь поряд із комп’ютером, на якому він зараз зберігається. Пошуки розумних альтернатив приводять користувачів до використання хмарних сховищ.

Актуальність теми дипломного проектування полягає в тому, що вищезгадана проблема є надзвичайно популярною, а більшість з існуючих рішень для резервного копіювання з використанням хмарного сховища є незручними або повністю не придатними для роботи із мобільною галереєю. Окрім того, мало який подібний сервіс може похвалитись привабливою ціновою

політикою, в той час як Google Photos (найпопулярніший на даний момент додаток для резервного копіювання медіа-файлів), починаючи із 1 червня 2021 року прибирає безкоштовний тариф, який надавав користувачам можливість вивантаження у хмару безлімітної кількості медіа-файлів. Ці два фактори разом вже найближчим часом спричинять «голод» на дешеві та надійні альтернативи у даному сегменті мобільних додатків.

Мета проекту – розробка мобільної галереї для ОС Android, яка дозволятиме її користувачам в автоматичному та ручному режимі вивантажувати медіа-файли в хмарне сховище Telegram, а також організовувати свою галерею шляхом додавання до них тегів, за якими потім можна буде виконувати пошук.

Завдання, які необхідно вирішити для досягнення мети:

– провести аналіз предметної області для визначення її головних особливостей;

– провести аналіз наявного програмного забезпечення;

– сформувати технічне завдання;

– спроектувати програмний продукт;

– ознайомитися та провести аналіз Telegram API для визначення методів, які будуть використовуватись у розроблюваному мобільному додатку;

– виконати програмну реалізацію проекту використовуючи при цьому фреймворк Xamarin;

– провести тестові випробування додатку;

– опублікувати додаток на платформі Google Play Market.

1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Змістовний аналіз предметної області, її структурних та функціональних особливостей

Мобільні телефони вже багато років слугують для нас вірними помічниками як вдома, так і на роботі. Перші телефони з'явилися ще на початку минулого століття, але з того часу багато чого змінилося. Так, із них досі можна подзвонити та прийняти виклик, але їхні можливості, як нашого постійного помічника та супутника у щоденній рутині, значно збільшились, причому збільшились настільки, що для таких пристроїв було придумано навіть окреме слово – смартфон.

Людство оцінило переваги мобільності, а тому ринок смартфонів розвивається шаленими темпами. Так само як до середини минулого століття було важко уявити комп'ютер, який зміг би поміститись під столом, так і ще якихось десять років тому мало хто міг подумати, що настільки тонкі корпуси мобільного телефону будуть вміщувати такі великі потужності. За прогнозами, уже до 2025 року понад 50% мобільних процесорів будуть мати підтримку та активно використовуватимуть у своїй роботі штучний інтелект [1], який донедавна вважався прерогативою лише стаціонарних комп'ютерів. Збільшення обчислювальних потужностей процесора також є критично важливим для ще одного обов'язкового для сучасного смартфону модуля – камери.

Багато років цифрові фотоапарати й близько не мали конкурентів серед мобільних камер. Зараз ними мало хто користується – у професійній зйомці вони були витіснені дзеркальними фотоапаратами, а для масового споживача смартфон виявився набагато кориснішим. Наведемо декілька його переваг:

– легкість та компактність;

– смартфон завжди під рукою;

– хороше апаратне забезпечення;

– якісне програмне забезпечення;

– широкі можливості для постобробки;

– продуктивність.

Розглянемо ці переваги детальніше.

Невеликі розміри смартфону, середня вага до 200 г та тонкий корпус роблять смартфон придатним для носіння у звичайних кишенях, не доставляючи при цьому особливого дискомфорту його власнику.

Потреба постійно бути на зв'язку, заодно із згаданою вище перевагою, призвели до того, що людина не розлучається зі смартфоном ні на мить (а цифровий фотоапарат вона бере з собою лише тоді, коли точно знає, що вона буде щось фотографувати).

Виробники смартфонів постійно покращують характеристики використовуваних модулів камери. Вони часто вдаються до встановлення декількох сенсорів, кожен з яких має своє власне призначення (датчик глибини, основна, широкоформатна і/або макрокамера). Згідно з дослідженням, проведеним Counterpoint, майже 20% проданих за перший квартал 2020 року смартфонів містили в собі блок із чотирьох камер [2].

Виробники смартфонів вже давно зрозуміли, що для того, щоб розкрити весь потенціал камери та отримувати чудові знімки, потрібне не лише хороше апаратне, але й якісне програмне забезпечення. На даний момент навіть камери бюджетних смартфонів здатні самостійно розпізнавати до декількох десятків сценаріїв зйомки, використовуючи при цьому штучний інтелект. Такі камери можуть аналізувати, обробляти та покращувати захоплені зображення і відео, виходячи з того, що саме на даний момент знімає користувач (небо чи деякий пейзаж, людину чи домашню тварину тощо) та при яких умовах він це робить (при хорошому чи поганому освітленні, на вулиці чи в будівлі). Виробники також часто розробляють для своїх смартфонів своє власне програмне забезпечення, яке стає їхньою своєрідною візитною карткою. Наприклад, лінійка смартфонів Google Pixel поставляється із додатком Google Camera, який вважається однією із кращих камер для ОС Android.

На смартфонах доступна величезна кількість програм для редагування вже готових фото, а наявність великого екрану дозволяє відразу переглянути й оцінити результати.

Смартфон дозволяє швидко і без зайвих клопотів опублікувати фото у соціальній мережі чи просто надіслати його своєму знайомому в месенджер (в той час, як фото із цифрової камери доведеться попередньо перенести на телефон чи комп'ютер).

Вказані переваги постійно роблять мобільну фотографію ще доступнішою та якіснішою. Проте, якщо вдаватися в деталі, то виявиться, що не все так однозначно. До прикладу, камери сучасних смартфонів вже спроможні робити фото у роздільній здатності понад 100 мегапікселів. Так, деталізація таких світлин вражає, але й місця вони займають немало. Одна така фотографія може займати в середньому від 50 до 70 мегабайт вільного простору, якого на смартфонах часто не вистачає. Згідно з дослідженням Avast, середньостатистичний користувач зберігає на своєму смартфоні 952 фотографії [3]. А окрім фото, ще ж необхідно зберігати відео (у середньобюджетному сегменті вже нікого не здивуєш можливістю записувати відео у 4К), інші додатки та файли. Ось чому проблема нестачі вільного простору є особливо актуальною для власників смартфонів. Згідно з опитуванням, близько 42% користувачів iPhone стикаються із нестачею пам'яті один раз на місяць або частіше. Окрім того, в настільки обширній галереї стає досить складно орієнтуватися і швидко знаходити потрібні фото. І якщо першу проблему ще можна вирішити, наприклад, витративши час на переміщення фото зі смартфону на комп'ютер, то другу проблему такі дії навпаки зроблять ще гострішою. В такому випадку, якщо ми не матимемо доступу до пристрою, на який ми перенесли фото, то ми просто не зможемо ні переглянути його на своєму смартфоні, ні відправити його комусь ще. Найоптимальним рішенням для даної проблеми виглядає можливість використання хмарних сховищ.

Хмарне сховище – це місце, в якому можна зберігати власні файли таким чином, щоб у будь-який момент можна було отримати до них доступ через

мережу Інтернет. Більшість хмарних сховищ є універсальними та дозволяють зберігати будь-які типи файлів. Однак, за цю універсальність доводиться платити тим, що шукати серед всього іншого саме медіа-файли стає набагато складніше. Іноді такі сховища навіть не мають попереднього перегляду. Окрім того, існують обмеження на вхідний і вихідний трафік та на розмір самого сховища. Як правило, ці обмеження можна пом'якшити за щомісячну плату певної суми. Тим не менше, навіть платні тарифи багатьох подібних сервісів не дозволяють спільно використовувати це сховище, виконувати автоматичну синхронізацію та додавати до файлів теги, щоб їх потім можна було по них знайти. Окрім того, не всі хмарні сервіси мають свої власні додатки для смартфонів, а використовувати для подібних цілей мобільний браузер неефективно та незручно. Взагалі, потреба в модернізації цієї області назрівала вже давно, але навряд чи хтось колись міг подумати, що це частково вдасться зробити месенджеру.

Telegram – це швидка та зручна програма для обміну повідомленнями, у якої вийшло повністю переосмислити поняття месенджера. Чати з іншими користувачами, закриті й публічні групи та канали, групові конференції, можливість коментувати пости – це далеко не повний список функцій даної програми. Месенджеру Telegram вдалося практично повністю стерти грань між соціальною мережею та месенджером. Але з-поміж конкурентів його відрізняє не лише це, але й повністю хмарна організація, що означає можливість отримати доступ до списку своїх листувань та їх історії у будь-який момент та із будь-якого пристрою. Окрім того, не існує жодних обмежень на кількість фото, відео та інших типів файлів, які можуть бути відправлені через цей месенджер. Зручний та приємний інтерфейс, заодно із вищеперерахованими факторами, зробили Telegram приреченим на успіх. Лише за січень 2021 року додаток встановили понад 100 мільйонів нових користувачів [4].

Проблема резервного копіювання та нестачі пам'яті наразі є надзвичайно актуальною, а Telegram – це один із найпопулярніших месенджерів. Тому є доцільним поєднати ці два фактори та розробити мобільний додаток, який

дозволятиме його користувачам налаштувати автоматичне резервне копіювання медіа-файлів у безпечне хмарне сховище Telegram. Окрім того, користувачам буде надана можливість маркувати файли за допомогою тегів, що дозволить структурувати галерею та значно пришвидшить пошук по ній. Використання хмари Telegram, як основи, також надає додаткові переваги, які на перший погляд є не зовсім очевидними. Наприклад, можна буде організувати спільне із іншими користувачами сховище шляхом створення групового приватного чату та налаштувавши автоматичне резервне копіювання у нього. Окрім того, у разі виникнення необхідності поділитися із іншим користувачем деяким файлом, який був збережений у хмарі раніше, то це можна зробити набагато швидше, просто переславши його, а не завантажуючи його туди ще раз (особливо це буде корисно та помітно на файлах великого розміру).

Тим не менше, згадані вище проблеми явно не єдині, з якими стикаються користувачі в умовах сучасних реалій. Зараз, коли майже всю інформацію про людину можна отримати із її соціальних мереж, проблема безпеки даних стає особливо актуальною. Тому, не дивлячись на ореол безпеки, який сформувався навколо Telegram, багато користувачів відчували б себе набагато впевненіше, якби знали, що їхні конфіденційні дані захищено додатково, ще перед відправкою в хмару. Такі дії забезпечили б вищий рівень захисту та дозволили б вберегти конфіденційну інформацію навіть у тому випадку, якщо зловмиснику вдасться отримати доступ до акаунту користувача.

1.2 Аналіз наявного програмно-технічного забезпечення предметної області

На сьогодні існує безліч програм для резервного копіювання, але далеко не всім із них вдалося здобути популярність. Насамперед це пов’язано із умовами, на яких надаються послуги хмарного сховища, обмеженнями, функціоналом та ціновою політикою розробників. Проте, існують також інші

фактори, які впливають на популярність продукту. Наприклад, шалений успіх додатку Google Photos пов'язаний не лише з його багатим функціоналом, але й з політикою самої компанії Google, згідно з якою цей додаток входить до списку попередньо встановлюваних на всі пристрої із ОС Android. Із ключових можливостей додатку варто відзначити наступні:

– можливість зручного перегляду всієї колекції медіа-файлів, що дозволяє використовувати додаток замість стандартної галереї;

– автоматичне резервне копіювання фото та відео у хмару;

– розпізнавання об'єктів на фото, що значно спрощує їх пошук;

– можливість редагувати світлини прямо в додатку;

– можливість групування об'єктів у галереї за часом або за альбомом, до якого вони належать.

Резервне копіювання виконується у хмару, а користувачу на вибір дається два варіанти, як це можна зробити. За умовами першого варіанту усі фото та відео будуть вивантажуватись без стиснення, але використовуватимуть дисковий простір, якого безкоштовно надається лише 15 ГБ, які спільно використовуються відразу кількома сервісами, включаючи Google Диск та Gmail. Зрозуміло, що для середньостатистичного користувача цього простору буде замало, а тому за весь понаднормовий об'єм даних доведеться платити кожного місяця.

Альтернативним варіантом виступає можливість стиснення фото та відео в обмін на безлімітне сховище. Ясна річ, що при стисненні зображення втрачають свою початкову деталізацію, тому якою б хорошою камерою не володів смартфон, всі фото врешті-решт будуть стиснені до 16 мегапікселів. Якщо якість не є критичною, то із цим обмеженням все ще можна змиритися, адже велику різницю в деталізації можна помітити лише при кадруванні. Саме завдяки таким вигідним умовам додатку вдалося завоювати багато прихильників по всьому світу. Тим не менше, вже у найближчому майбутньому розробники Google Photos планують позбавити своїх користувачів можливості вивантажувати необмежену кількість стиснутих медіа-файлів. Починаючи з 1 червня 2021 року розмір абсолютно всіх вивантажуваних медіа буде

зараховуватись до безкоштовного сховища об’ємом 15 ГБ [5]. У разі, якщо користувач перевищить цей ліміт, він буде забов’язаний або звільнити місце, або збільшити об’єм сховища, оформивши щомісячну підписку.

В плані зовнішнього вигляду Google Photos відрізняється з-поміж інших додатків простим та приємним інтуїтивно зрозумілим матеріальним дизайном. У ньому досить легко орієнтуватися, а більшість дій можна виконати всього за декілька простих кроків. Зовнішній вигляд головного екрану додатку показано на рисунку 1.1.

Рисунок 1.1 – Головний екран додатку Google Photos

Dropbox – це досить популярний хмарний сервіс, який дозволяє зберігати будь-які типи файлів. І хоча ним мало користуються у Східній Європі, чомусь саме цей додаток вважається хорошою альтернативою для раніше згаданого

Google Photos після того, як він змінить свою політику зберігання медіа. Найбільшим недоліком Dropbox є об’єм безкоштовного дискового простору – всього 2 ГБ. При цьому тарифні плани є дорожчими, ніж у конкурентів. Тим не менше, багато хто вважає це достойною платою за ту безпеку й надійність, яку пропонує цей сервіс. До ключових особливостей додатку можна віднести:

– автоматичне резервне копіювання;

– наявність окремого сховища для найбільш конфіденційних документів;

– можливості зберігати паролі.

Головний екран додатку Dropbox показано на рисунку 1.2.

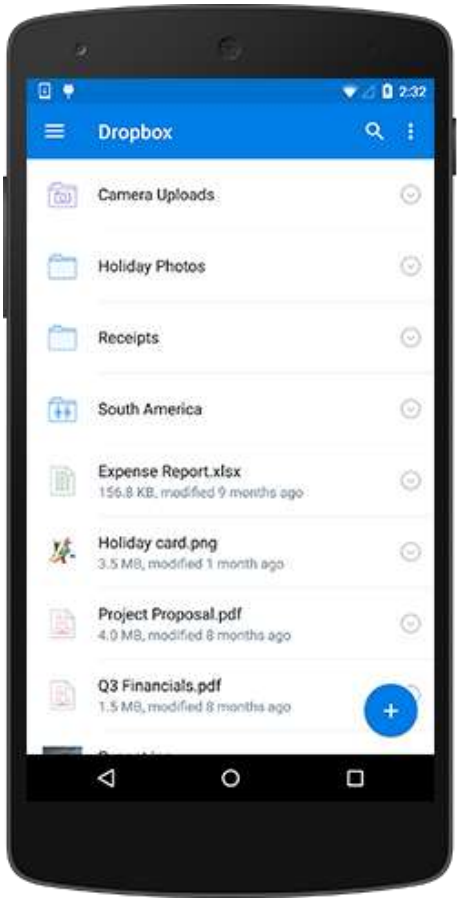


Рисунок 1.2 – Головний екран додатку Dropbox

До недоліків цього додатку також можна віднести систему пошуку у безкоштовному тарифному плані, яка дозволяє шукати файли лише за їх назвами. Перехід на професійний план дозволить шукати файли серед об’єктів,

які були розпізнані на медіа-файлах автоматично. Додавати свої власні теги до файлів не можна, а тому досягти чогось подібного, користуючись безкоштовним тарифним планом неможливо.

Mega – це хмарне сховище, яке відразу підкуповує своїми умовами любителів об'ємних дисків. Навіть безкоштовний тарифний план може похвалитися чималим обсягом дискового простору – цілих 50 ГБ, а платні тарифи, в перерахунку на вартість за кожен гігабайт, є дешевшими, ніж у конкурентів. Тим не менше, за подібні об'єми частково доводиться платити зручністю користування.

На Mega існують квоти на передачу даних. Раніше, згідно з умовами безкоштовного тарифу, для користувача встановлювався ліміт у розмірі 4 ГБ (кожні шість годин цей лічильник скидався) на отримання та передачу даних. Це дуже незручно у випадках, коли потрібно виконати резервне копіювання більших об'ємів даних, особливо, якщо часу, щоб почекати на оновлення квоти, немає. Зараз же умови надання послуг дещо змінились. Тепер квота розраховується динамічно в залежності від навантажень на сервери Mega, часу, країни та постачальника послуг Інтернет, яким користується користувач. Платні тарифи володіють чітко прописаними в умовах квотами, які надаються відразу на місяць. У разі перевищення цих лімітів користувачу пропонується перейти на дорожчий тарифний план, інакше він просто не зможе продовжити користуватись можливостями сервісу аж до моменту поновлення квоти (тобто до настання наступного місяця). До недоліків також можна віднести швидкість завантаження та вивантаження файлів, яка дуже часто залишає бажати кращого.

Сховище Mega є доступним як із браузера, так і з усіх популярних настільних та мобільних платформ. Окрім того, наявність зрозумілого відкритого API заохотила багатьох розробників створювати свої власні неофіційні клієнтські програми, підлаштовуючи їх під певні потреби. Якщо говорити про офіційний додаток, то він, як і Dropbox, не зможе замінити для користувача стандартну галерею (оскільки ця програма не призначена для роботи з медіа і переглядати та орієнтуватися серед фото та відеофайлів

незручно), а можливість якимось чином маркувати файли (щоб потім їх можна було простіше знайти) відсутня.

Головний екран офіційного додатку Mega показано на рисунку 1.3.

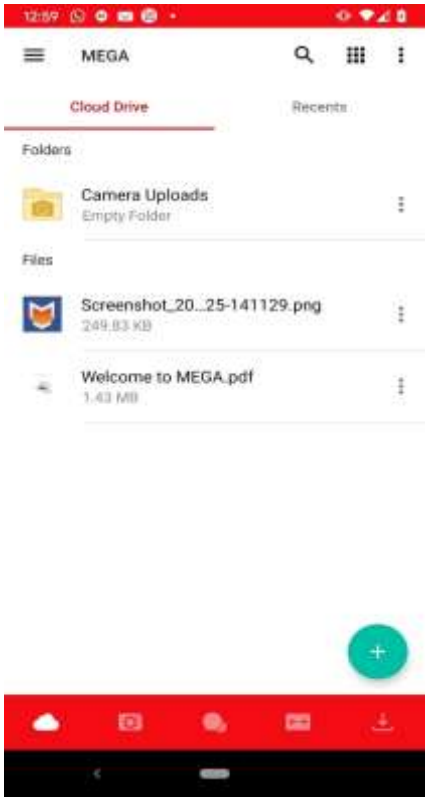


Рисунок 1.3 – Головний екран додатку Mega

Із ключових можливостей додатку варто відзначити наступні:

– автоматичне резервне копіювання (в тому числі для медіа-файлів);

– можливість створювати чати з іншими користувачами;

– наскрізне шифрування усіх файлів та чатів;

– можливість зберігання попередніх версій файлів.

Щоправда, опція автоматичного резервного копіювання фото та відео розповсюджується відразу на всі медіа-файли і налаштувати її таким чином, щоб, наприклад, не зберігати у хмару фото із альбому «Знімки екрану» (чи будь-якого іншого) неможливо. Оскільки будь-які передачі даних використовують квоту, яка у будь-який момент може закінчитись, то такий підхід (при якому автоматичне резервне копіювання виконується і для файлів, які

копіювати неважливо або не потрібно в принципі) не підійде для більшості користувачів. Єдиним виходом із такої ситуації слугує вимкнення опції автоматичного зберігання медіа-файлів у хмару та перехід на ручний режим, де користувач сам обиратиме, які файли потрібно вивантажувати на сервери Mega.

Безпека Mega також виявилась «палкою із двох кінців». Оскільки на Mega всі дані шифруються на основі ключа, сформованого із паролю користувача, то змінити цей пароль, без втрати доступу до всіх попередньо вивантажених файлів, просто неможливо. Це також означає, що зі втратою пароля буде втрачений доступ і до даних. Єдиним виходом в цій ситуації є використання ключа відновлення (якщо користувач, звісно, його десь попередньо зберіг).

На сьогодні існує також багато інших сервісів, які, хоча й відстають за популярністю від вищезгаданих, але надають значно приємніші умови (принаймні так може здаватися на перший погляд). Хорошим прикладом може слугувати додаток Dubox, який надає своїм користувачам цілий терабайт хмарного сховища абсолютно безкоштовно. «Підступ» полягає у ліцензійній угоді, з якою погоджується користувач на етапі реєстрації. Згідно з нею компанія залишає за собою право в будь-який момент та на свій розсуд змінити умови надання послуг як для окремо взятого користувача, так і для всіх та відразу. В угоді також сказано, що деякі (або навіть всі файли) можуть бути видалені без попередження та можливості відновлення. Зрозуміло, що такий сервіс не може викликати повної довіри, а використовувати його для збереження даних, які становлять для користувача хоча б якусь цінність, недоречно.

Не дивлячись на згадані вище особливості, Dubox все ще працює та з кожним днем розширює свою аудиторію, на відміну від сервісів Shoebox та Ever, які проіснували всього близько двох років, після чого були закриті. Серед інших додатків ці два сервіси також відрізнялися доволі вигідними умовами і вони також стали популярними в надзвичайно короткі терміни.

Отже, не варто забувати, що для хмарного сховища важлива не лише зручність, швидкість, об’єм сховища та спосіб шифрування файлів користувача, але й загальна надійність та репутація самого сервісу. Врешті-решт, користувачі

виконують резервне копіювання, щоб вберегти свої файли, і вони не повинні боятися втратити їх у будь-який момент. Тому ступінь довіри до сервісу є ключовим фактором у виборі хмарного сховища.

Для більш наочного представлення та порівняння найпопулярніших хмарних сховищ між собою (з точки зору їх використання як додатку для перегляду та резервного копіювання медіа-файлів) вони були проаналізовані за деякими ключовими критеріями. На основі отриманих даних було складено порівняльну таблицю наявного програмного забезпечення (таблиця 1.1).

Таблиця 1.1 – Порівняльна характеристика наявного ПЗ

Назва	Об'єм безкоштовного дискового простору	Можна використовувати замість стандартної галереї	Можливості для пошуку	Автоматичне резервне копіювання медіа-файлів
Google Photos	15 ГБ	Так	По назві, альбому, вмісту (визначається автоматично із використанням засобів штучного інтелекту)	Є, за замовчуванням резервне копіювання виконується лише із альбому камери. У разі потреби можна увімкнути автоматичну синхронізацію і для інших альбомів.
Назва	Об'єм безкоштовного дискового простору	Можна використовувати замість стандартної галереї	Можливості для пошуку	Автоматичне резервне копіювання медіа-файлів
Dropbox	2 ГБ	Ні	По назві та вмісту (лише у платному тарифі)	Автоматична синхронізація копіює у хмару всі медіа-файли із всіх альбомів. Для більш гнучкого керування процесом, її необхідно вимкнути і налаштувати автоматичне резервне копіювання вручну таким чином, як це робиться і для звичайних папок.
Mega	50 ГБ (з обмеженнями у вигляді квот, які розраховуються динамічно)	Ні	Лише по назві	Так само, як і в Dropbox

Проведений аналіз подібного програмного забезпечення показав, що розроблюваний додаток повинен:

– мати щедрий об’єм безкоштовного дискового простору (Telegram не накладає жодних обмежень на кількість файлів, які користувач може вивантажити у їхнє хмарне сховище);

– бути взаємозамінним із стандартним додатком галереї;

– мати можливість організовувати медіа-файли для того, щоб серед них можна було простіше орієнтуватися та швидше здійснювати пошук;

– мати функцію автоматичного резервного копіювання із можливістю ручного налаштовувати того, з яких саме альбомів слід синхронізувати медіа-файли;

– мати простий та інтуїтивно зрозумілий дизайн інтерфейсу;

– бути максимально надійним.

1.3 Визначення вимог до мобільного додатку

Опишемо вимоги до розроблюваного додатку та покажемо, що саме зможе зробити користувач за допомогою діаграми варіантів використання (ВВ).

У таблиці 1.2 подано опис акторів (користувачів додатку), а в таблиці 1.3 – самих варіантів використання.

Таблиця 1.2 – Опис акторів

Актор	Короткий опис
Неавторизований користувач	Може конфігурувати загальні налаштування клієнта, в тому числі використання проксі-сервера для обходу блокувань у деяких регіонах. Для доступу до основного функціоналу додатку користувачу необхідно авторизуватися через Telegram.
Авторизований користувач	Має можливість переглядати свою локальну галерею, а також медіа-файли, які були збережені у хмару раніше. Може додавати до медіа-файлів теги, шукати серед доданих тегів та альбомів, налаштовувати автоматичне резервне копіювання, має можливість швидко поділитися знайденим медіа-файлом.

Таблиця 1.3 – Опис варіантів використання

Актор	Назва ВВ	Опис ВВ
1	2	3
Неавторизований користувач	Зміна стандартного API ID та API Hash	Користувач може отримати API ID та API Hash на сторінці https://my.telegram.org/apps та вказати їх у додатку. Буде корисним для забезпечення більшого рівня надійності та дозволить продовжити використовувати додаток у випадку, якщо оригінальний API ID буде заморожений або заблокований.
	Зміна інформації про клієнта	Дозволяє кастомізувати ім’я пристрою, версію системи, версію додатку та мову, прийнятну для даного клієнту. Таким чином, дозволяє легше відрізняти сесію даного додатку від інших активних сесій для даного акаунту.
	Налаштування проксі-сервера	Дозволяє використовувати проксі-сервер у разі виникнення необхідності (наприклад, для обходу блокування в деяких регіонах)
	Зміна стандартного дата-центру	Просунуті користувачі можуть самостійно вказати дата-центр із яким буде ініціалізуватися перше з’єднання.
	Авторизація через Telegram	Користувач може авторизуватися, використовуючи свій Telegram-акаунт.
Авторизований користувач	Вибір чату, в який буде виконуватись копіювання медіа-файлів	Користувач може створити нову приватну групу в яку будуть зберігатися медіа-файли або вибрати існуючу.
	Вибір чату в який буде виконуватись копіювання бази даних	Користувач може налаштувати автоматичне збереження зашифрованої копії локальної бази даних у хмару після завершення кожної операції резервного копіювання.
	Вибір альбомів для автоматичного копіювання	Користувач може вказати з яких саме альбомів слід виконувати автоматичне резервне копіювання медіа-файлів.
	Встановлення графіку автоматичного резервного копіювання	Користувач може обрати бажаний час для початку процесу автоматичного резервного копіювання.
	Перегляд фото та альбомів	Користувач має можливість переглядати альбоми та їх вміст прямо з додатку.
	Редагування тегів	Користувач може додавати та видаляти теги із медіа-файлів.
	Пошук по тегах	Користувач може виконувати пошук серед усіх медіа-файлів за раніше прикріпленими тегами.

Кінець таблиці 1.3

1	2	3
Авторизований користувач	Ручне резервне копіювання у звичайному режимі	Користувач може виконувати резервне копіювання окремо взятих медіа-файлів.
	Ручне резервне копіювання із попереднім шифруванням	Аналогічно до попереднього випадку, тільки перед резервним копіюванням файл буде зашифровано.
	Поділитися медіа-файлом	Користувач може поділитися медіа-файлом через інші встановлені на його пристрої додатки.
	Експорт локальної бази даних	Користувач може експортувати БД у файл та зберегти його на будь-якому зручному для нього носії або відправити через будь-який додаток.
	Очищення даних сесії	Дозволяє користувачу не лише вийти з Telegram акаунту, але й видалити всі дані пов’язані із поточною сесією.
	Імпорт бази даних	Локальна БД може бути замінена експортованою раніше БД.
	Перевірка актуальності даних	Додаток автоматично виявлятиме та оновлюватиме застарілі дані по мірі роботи з ними, але користувач може запустити повну процедуру перевірки актуальності даних вручну. Це дозволяє прибрати із бази даних всю неактуальну інформацію.
Авторизований користувач	Видалити скопійовані у хмару медіа-файли	Користувач може звільнити місце на пристрої видаливши із нього медіа-файли, які вже були до цього вивантажені у Telegram.

Відповідно до описаних акторів та варіантів використання побудовано діаграму, яка подана на рисунку 1.4.

Технічне завдання на розробку мобільного додатку подано у додатку А.

Таким чином, у розділі було проаналізовано предметну область, з’ясовані причини частого виникнення проблеми нестачі пам’яті у користувачів смартфонів, а також досліджені наявні програмні засоби для резервного копіювання. В результаті було визначено їх переваги та недоліки, а також доведено потребу у розробці спеціалізованого програмного продукту, який призначатиметься саме для роботи з медіа-файлами. Сформовані вимоги до

мобільного додатку були описані за допомогою діаграми варіантів використання та формувалися, опираючись на інші успішні додатки. Окрім того, майбутня зміна умов надання послуг у найпопулярнішого в світі додатку для резервного копіювання фото змусить частину його користувачів шукати альтернативи, що робить кінець весни 2021 року ідеальним часом для запуску нового програмного продукту.

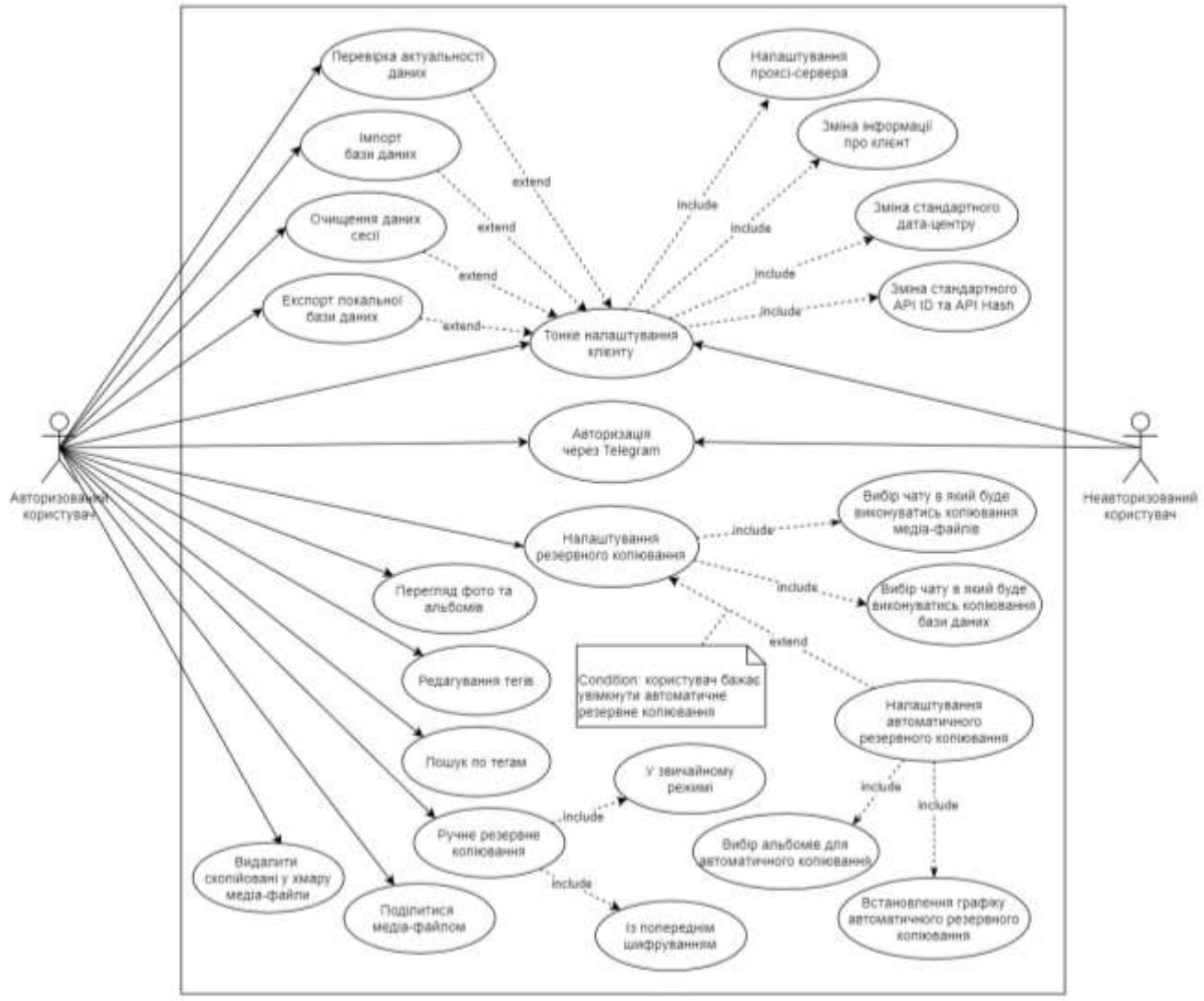


Рисунок 1.4 – Діаграма варіантів використання

2 ПРОЕКТУВАННЯ МОБІЛЬНОГО ДОДАТКА

2.1 Архітектура та функціональна структура додатка

При розробці мобільних додатків використовується декілька основних архітектурних підходів. Вибір найбільш підходящого із них дуже сильно залежить від того, які проблеми повинен вирішувати додаток, та на якій платформі і яким чином він це робитиме. Розглянемо детальніше найпопулярніші архітектурні шаблони проектування мобільних додатків в контексті їх використання разом із фреймворком Xamarin. До таких шаблонів належать MVC, MVP та MVVM.

Архітектура MVC (Model-View-Controller) виступає загальноприйнятим стандартом в розробці мобільних додатків. Даний шаблон чудово підходить для невеликих проектів та дозволяє досить швидко розробити концептуальну версію додатку, за рахунок того, що для цього шаблону існує нативна підтримка як на платформі Android так і на iOS. Даний шаблон дозволяє відділити логіку представлення даних від їх обробки, розділивши додаток на 3 рівня відповідальності: Model, View та Controller.

Модель (Model) – це компонент, який відповідає за дані. Сюди відносяться бізнес-об’єкти, які містять в собі логіку збереження й отримання інформації, із якою працює додаток. Іноді його ототожнюють із шаром доступу до даних (DAL).

Представлення (View) – це компонент, який відповідає за відображення користувацького інтерфейсу. Представлення можуть відображати дані отримані від моделі.

Контролер (Controller) – це компонент, який містить всю логіку конкретного екрану (сторінки додатку). Контролер обробляє дії користувача і потім оновлює модель і/або представлення.

Даний шаблон ще називають пасивним представленням MVC. Існує також варіація цього шаблону, яка називається Massive View Controller. Основна

відмінність полягає у тому, що діяльність (екран додатку) разом із її логікою потрібно розглядати як шар відображення, а для контролера необхідно створювати окремий клас.

Загальна схема шаблону MVC показана на рисунку 2.1.

Рисунок 2.1 – Схема шаблону MVC

Тим не менше, використання MVC в додатках із складними представленнями спричиняє проблему перевантажених контролерів, які стають доволі об’ємними та можуть роздуватися до кількох тисяч стрічок коду. В таких об’ємах коду дуже складно орієнтуватися, не говорячи вже про його підтримку та розширення. Окрім цього, подібний код також складно тестувати. Шаблон MVC погано підходить для використання у кросплатформних сценаріях. Натомість Microsoft рекомендує притримуватись шаблону MVP або MVVM при розробці проектів з використанням фреймворку Xamarin [6].

Шаблон MVP (Model-View-Presenter) це варіація шаблону MVC, яка чудово підходить для розробки кросплатформних додатків та дозволяє в повній мірі скористатися нативними можливостями платформи. Окрім цього, правильне використання MVP дає більше контролю над користувацьким інтерфейсом додатку. Представник (Presenter) містить в собі всю логіку користувацького інтерфейсу і відповідає за синхронізацію моделі та представлення. Вся інформація про дії користувача передається із представлення для представника. Після цього представник приймає рішення про оновлення моделі та синхронізує всі зміни між моделлю та представленням. Основна відмінність від MVC полягає у тому, що представник взаємодіє із представленням не напряму, а через інтерфейс, який робить їх слабко

пов’язаними один з одним. Такий підхід також сильно спрощує тестування, адже представник та представлення можуть бути протестовані окремо. Загальну схему шаблону MVP показано на рисунку 2.2.

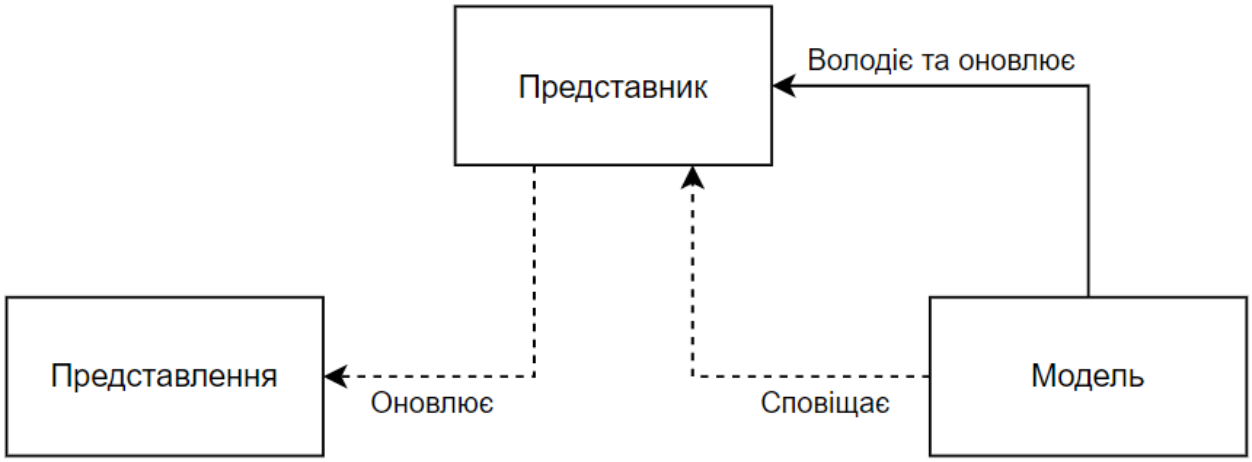


Рисунок 2.2 – Схема шаблону MVP

Представник придатний для повторного використання та може бути замінений в залежності від того, на якій платформі запущено додаток, а представлення можуть реалізовувати відразу декілька інтерфейсів. Це робить даний шаблон значно гнучкішим за MVC, а також дещо спрощує підтримку та розширення існуючого коду. Тим не менше, при такому підході доведеться писати доволі багато коду як в представленні, так і в представнику, через необхідність створення та реалізації всіх інтерфейсів. Оскільки кожен інтерфейс повинен охоплювати лише невелику частину пов’язаних між собою взаємодій, то таких інтерфейсів врешті-решт може виявитись дуже багато.

Шаблон MVVM (Model-View-ViewModel) дозволяє прив’язувати елементи представлення до моделі представлення (ViewModel) замість того, щоб прив’язувати представлення до інтерфейсу (як це відбувається в MVP). Відповідно, контролер представлення звільняється від коду взаємодії з моделлю, бо тепер відповідальність за це бере на себе модель представлення. Саме представлення повинне зберігати посилання на модель представлення, однак модель представлення не знає про представлення в якому воно

відображається. Це дозволяє використовувати одну й ту ж саму модель представлення із різними представленнями.

Модель представлення по суті є обгорткою для тих даних моделі, які підлягають зв’язуванню (Data Binding). Тому модель представлення може спостерігати за змінами в звичайній моделі, а оскільки представлення та модель представлення пов’язані між собою, то всі зміни в моделі автоматично відображатимуться в графічному інтерфейсі користувача (на відміну від попередніх шаблонів, де UI потрібно було оновлювати вручну). Окрім того, зв’язування можна налаштувати таким чином, щоб воно працювало в обидві сторони, що означає що зміни даних в інтерфейсі користувача відразу відображатимуться і на моделі. Це дозволяє по максимуму використовувати можливості додатка за рахунок гнучкої взаємодії з його користувачами та звільняє від необхідності написання зайвого шаблонного коду. Представлення взаємодіє із своєю моделлю через команди, які також зв’язуються. Окрім цього, модель представлення добре піддається юніт тестуванню. Загальну схему архітектурного шаблону MVVM показано на рисунку 2.3.

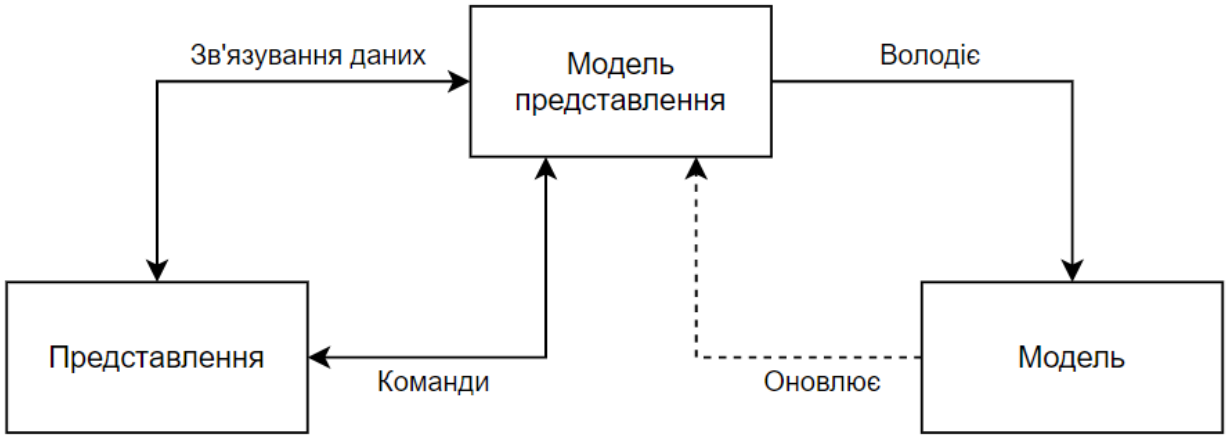


Рисунок 2.3 – Схема шаблону MVVM

Стандартна практика рекомендує використовувати MVVM у разі, якщо існує велика імовірність виникнення «роздутих» контролерів представлень. Навіть якщо при розробці додатку із самого початку використовувався шаблон

MVC, міграція на MVVM буде відносно легкою та швидкою.

До причин використовувати MVVM також можна віднести загальну продуктивність додатків, в яких використовується цей шаблон. Синтетичні тести показують, що додатки, побудовані на архітектурі MVVM, в середньому працюють дещо швидше за аналогічні, які створювались слідуючи шаблону MVC або MVP. І якщо різниця з архітектурою MVP практично непомітна, то додатки на MVC зазвичай використовують на 24% більше оперативної пам'яті (необхідної для відображення екрану додатка) та створюють на 17% більше навантаження на центральний процесор, ніж додатки побудовані з використанням архітектури MVVM [7].

З огляду на описані вище фактори, мною було прийнято рішення про використання архітектури MVVM, яка, окрім вищезгаданого, також має чудову підтримку з боку Xamarin. Даний фреймворк надає все необхідне для швидкого налаштування прив'язок даних між представленнями та їх моделями, як за допомогою мови C# (в контролері представлення) так і мови XAML (в самому представленні), яка тут використовується для розмітки інтерфейсу користувача.

2.2 Аналіз Telegram Client API

Telegram надає доступ до двох різних видів API [8]: Bot API (для створення ботів) та Client API (для розробки своїх власних клієнтів). І якщо логіка бота повинна виконуватись на сервері його розробника (Telegram в такому випадку слугуватиме лише інтерфейсом для прийому та відправки повідомлень), то розробникам клієнтів немає необхідності розробляти свої власні сервери, адже вони можуть взаємодіяти із серверами Telegram напряму, використовуючи клієнтське API, яке доступне абсолютно безкоштовно для всіх розробників. Ще однією ключовою відмінністю виступає те, що в Bot API всі запити виконуються з використанням стандартного HTTP протоколу і ніяких знань про низькорівневий протокол Telegram від розробників ботів не

вимагається в принципі. Зрозуміле та багатофункціональне API, разом із відсутністю необхідності вникати у внутрішню будову месенджера, стали причиною такого широкого розповсюдження в Telegram найрізноманітніших ботів. Щодо клієнтських додатків, то тут все дещо складніше.

Розберемо процес розробки власного Telegram клієнту і визначимо вхідні та вихідні дані основних функцій Telegram Client API, які будуть використовуватись для досягнення цілей, встановлених завданнями дипломного проектування. Отримана інформація знадобиться в процесі детального проектування та допоможе в подальшому виділити основні сутності з якими працюватиме розроблюваний додаток.

Перш за все, необхідно створити новий клієнтський додаток. Для цього потрібно увійти в свій Telegram акаунт та перейти на сторінку https://my.telegram.org/apps. У кожного додатка є свої власні унікальні API Id та API Hash. Варто відзначити, що з одного акаунту не можна створити більше ніж один додаток, а публікація або передача свого API Id та API Hash третім особам заборонена. Окрім того, якщо додаток буде використовуватись для розсилки спаму або флуду, то його буде заблоковано назавжди [9]. Оскільки планується що розроблюваний додаток буде виконувати автоматичне резервне копіювання медіа-файлів (яких може бути досить багато), то користувачам буде заборонено виконувати резервне копіювання у діалоги із іншими користувачами або у чати, які користувач не створював власноруч. Тобто резервне копіювання можна буде виконувати лише у свої збережені повідомлення або у приватний канал (при умові що користувач є його власником). Це дозволить уникнути звинувачень у флуді чи спамі (або, принаймні, звести такі випадки до мінімуму).

При створенні нової сесії та ініціалізації з’єднання із серверами Telegram у першу чергу необхідно створити ключ авторизації. Він генерується згідно з алгоритмом Діффі-Хелмана [10], при якому одна з двох створених частин ключа ніколи не передається через канали зв’язку та не покидає пристрій користувача. Після цього всі наступні повідомлення шифруються з врахуванням згенерованого в результаті виконаних дій ключа авторизації.

Для забезпечення безпечного обміну повідомленнями між клієнтом та сервером у Telegram використовується протокол їх власної розробки – MTProto. Даний протокол складається із трьох основних частин [11]:

– високорівнева (або мова запитів API), яка визначає яким чином запити та відповіді будуть перетворюватись в двійкові повідомлення і назад (серіалізуватися та десеріалізуватися);

– криптографічна, яка визначає методи шифрування повідомлень перед їх передаванням через транспортний протокол;

– транспортна, яка визначає за допомогою якого іншого протоколу передачі даних (TCP, UDP) клієнт та сервер будуть обмінюватися повідомленнями між собою.

Для опису типів, а також правил їх серіалізації та десеріалізації у двійкові повідомлення використовується мова TL (Type Language) [12]. Об’єктами в TL виступають не лише типи, але й функції. Окрім того, дана мова має підтримку поліморфізму [13], а абсолютна більшість API методів як результат свого виконання повертають саме поліморфні типи. Якщо спроектувати схему TL на мову C#, то такі типи у ній виражатимуться за допомогою абстрактних класів. Абсолютно всі об’єкти в мові TL мають конструктор (його не слід плутати із конструкторами в інших мовах програмування). TL конструктор це число, яке однозначно ідентифікує тип. Саме він використовується для виведення конкретного типу із поліморфного.

Для того, щоб краще зрозуміти поняття схеми TL, проаналізуємо її на простому прикладі. Наприклад, ось так виглядає опис функції для оновлення імені користувача:

```
account.updateUsername#3e0bdd7c username:string = User;
```

А тепер по порядку:

– ім’я типу (в даному випадку ім’я функції): account.updateUsername (приставка account позначає простір імен);

– конструктор (визначається за алгоритмом CRC32): 3e0bdd7c;

– ім’я параметру та його тип (через двокрапку): username:string;

– тип результату (після знаку дорівнює): User.

Тип User є поліморфним (взагалі, всі типи в схемі TL, що починаються з великої літери є такими), а тому все що про нього відомо це його конструктор (оскільки всі типи в мові TL гарантовано мають це поле). В документації вказано, що даний тип має лише два підтипи: userEmpty та user. Порівнявши конструктор відповіді із конструкторами можливих підтипів, можна визначити її реальний тип та десеріалізувати результат у правильний об’єкт.

Тепер, коли ми вже володіємо базовими знаннями про мову TL, виділимо та проаналізуємо вхідні та вихідні дані для функцій, які ми використовуватимемо в додатку найчастіше. Перш за все, було б дуже зручно якби користувач міг створювати новий приватний канал для резервного копіювання прямо з додатка. Ось TL схема необхідної функції:

```
channels.createChannel#3d5fb10f flags:# broadcast:flags.0?true
megagroup:flags.1?true for_import:flags.3?true title:string
about:string geo_point:flags.2?InputGeoPoint address:flags.2?string
= Updates;
```

Із незвичайного варто відмітити складене поле flags, яке має тип #. Згідно з схемою TL, тип # є псевдонімом типу nat, яке використовується для позначення натуральних чисел [14]. Те, чи встановлений конкретний біт цього числа визначатиме чи створюється канал і чи є він мегагрупою. Результатом виконання запиту є поліморфний тип Updates, який необхідно буде десеріалізувати в конкретний тип updates та перевірити, чи існує серед отриманих оновлень оновлення, яке стосується створення каналу. Якщо таке є, значить канал було створено успішно і це оновлення (яке матиме поліморфний тип Chat) потрібно буде десеріалізувати в тип channel. Даний тип має доволі велику кількість полів, які описують всю інформацію про канал, а тому він не буде розглядатися повністю. Весь цей шлях був необхідний, оскільки згідно з

TL схемою нам достатньо мати лише два параметри щоб відправити повідомлення в канал: його ідентифікатор (параметр id, який має тип int) та хеш доступу (access_hash, який має тип long). Ці параметри наявні в отриманому об’єкті channel. Даний етап можна було б підсумувати тим, що в додатку необхідно буде зберігати ці два значення для кожного каналу, який використовуватиметься для резервного копіювання медіа-файлів чи бази даних, однак зберігати в базі даних (чи налаштуваннях додатку) хеш доступу до каналу недоцільно. Причиною цього виступає те, що хеш доступу не постійний та з часом перестає бути актуальним. Документація не дає відповіді на запитання скільки саме часу ним можна користуватися. Хеш доступу може змінитись навіть після того, як клієнт підключиться до сервера заново. Тому найоптимальнішим варіантом буде зберігати лише числові ідентифікатори таких каналів. У разі виникнення необхідності відправки повідомлення в один із таких чатів, спершу потрібно буде отримати його хеш доступу, скориставшись функцією із Telegram API для отримання інформації про чат за його id.

Наступним ключовим моментом є вивантаження та завантаження файлів із Telegram. Розберемо лише основні моменти без наведення прикладів TL схеми. Розпочнемо із вивантаження.

Якщо мова йде про медіа-файли, то в Telegram їх можна вивантажити двома способами: як медіа-файл (за замовчуванням; медіа-файли при цьому буде стиснено) та як документ (без стиснення). Нас же цікавить лише другий варіант. Згідно з документацією Telegram, документи бувають двох типів: звичайні та великі [15]. Великими вважаються файли, розмір яких перевищує 10 МБ. Різниця полягає у тому, що для великих файлів Telegram не генеруватиме іконок (зображень в низькому розширенні, які використовуватимуться для відображення попереднього перегляду). Файли в Telegram вивантажуються частинами. Відповідно, в Telegram API існує дві функції: saveFilePart та saveBigFilePart. Обидві функції як параметри приймають file_id (генерується випадковим чином на стороні клієнта), номер частини та масив байт (частина файлу). Для великих файлів додатково потрібно вказувати загальну кількість

частин, на яку було розділено файл. Варто зазначити, що всі частини повинні бути однакового розміру та без остачі ділитися на 1024 (1 КБ). Окрім того, число 524288 (512 КБ) повинне без остачі ділитися на вибраний розмір частини, а максимальна кількість частин – 3000 [15]. Це означає, що розмір вивантажуваного файлу не повинен перевищувати 1.5 ГБ. Однак, в одному з останніх оновлень було вказано, що максимальний розмір вивантажуваних файлів збільшено до 2 ГБ [16]. Це вказує на те, що офіційна документація, яка описує роботу з файлами, дещо застаріла. Але оскільки навіть 2 ГБ на один файл не завжди достатньо, в додатку необхідно передбачити можливість розбиття вихідного файлу на шматки (наприклад, по 1.5 ГБ), кожен з яких буде вивантажуватись окремо. Це дозволить вивантажувати у Telegram файли будь-якого розміру. Після завершення вивантаження файлу його можна відправити в чат. Як результат цієї дії ми отримаємо поліморфний тип Updates, з якого ми можемо дізнатися всю інформацію про щойно відправлене повідомлення.

Для завантаження документів з Telegram використовується функція getFile, якій необхідно передати ідентифікатор документа (id, має тип long), хеш доступу (access_hash, має тип long) та посилання на файл (file_reference, масив байт). Всі ці дані можна отримати із щойно відправленого повідомлення та відразу зберегти у базу даних додатка, але, як і у випадку з каналами, хеш доступу та посилання на файл не постійні. Кожні два-три дні (іноді частіше) посилання стає недійсним, і його потрібно оновлювати. Для того, щоб оновити посилання потрібно знайти повідомлення в якому було відправлено документ. Зробити це можна за допомогою функції channels.getMessages, яка приймає на вхід інформацію про канал, в якому потрібно шукати повідомлення (ідентифікатор та хеш доступу) та вектор ідентифікаторів повідомлень, які необхідно знайти. Отже, в базі даних доцільно зберігати лише контекст, в якому було відправлено повідомлення (ідентифікатор чату та ідентифікатор повідомлення), а не посилання на документ. Більше того, використання посилання на документ може спричиняти деякі непорозуміння. Наприклад, якщо видалити файл із Telegram, то він ще деякий час буде доступний за

кешованим посиланням (якщо, звісно, воно було актуальним на момент видалення). Це може бути пов’язано із тим, що насправді Telegram видаляє файли не відразу, а лише через певний проміжок часу. Тим не менш, така поведінка додатку могла б здатися користувачам дивною та навіть викликати недовіру, оскільки у них могло б скластися враження, що їхні персональні файли було завантажено додатком ще й у якесь інше місце, про яке вони не знають і завантаження видаленого файлу відбувається саме звідти (що, звісно, не так). В той же час, отримати посилання на файл із вже видаленого повідомлення неможливо, про що користувача можна буде повідомити відразу.

2.3 Визначення основних модулів додатка

В результаті попереднього аналізу функціональної структури додатка, мною було визначено 5 основних рівнів (модулів), з яких складатиметься додаток Cloudgram:

– рівень даних;

– рівень доступу до даних;

– рівень мережевої взаємодії з Telegram;

– рівень додатка;

– рівень користувацького інтерфейсу.

Розглянемо кожен з цих рівнів детальніше.

Рівень даних використовуватиметься для довготривалого зберігання даних (сюди відносяться бази даних та параметри додатка) а також модель ПО.

Рівень доступу до даних по суті буде оболонкою для попереднього рівня, який приховуватиме свою внутрішню реалізацію від викликаючого коду, та надаватиме простий інтерфейс для створення, читання, оновлення та видалення даних додатка.

Рівень мережевої взаємодії з Telegram складатиметься із двох модулів: бібліотеки для взаємодії з Telegram Client API та оболонки над нею, яка

виступатиме адаптером і у своїй внутрішній реалізації автоматично трансформуватиме об’єкти-моделі із рівня доступу до даних в об’єкти схеми TL. Окрім цього, адаптер також візьме на себе всю роботу по синхронізації роботи внутрішньої бібліотеки в багатопоточних сценаріях шляхом організації та підтримки розумної пріоритетної черги із запитів, що включає наступне:

– автоматичне повторне підключення до Telegram у разі втрати з’єднання;

– зупинка виконання запитів з черги до моменту поки користувач не здійснить повторне підключення самостійно (у разі якщо спроба автоматичного повторного підключення була невдалою, наприклад, якщо це неможливо оскільки зникло з’єднання з Інтернетом);

– дотримання виконання запитів з черги в порядку їх пріоритету (від запитів з найвищим пріоритетом до запитів із найнижчим);

– автоматичне ігнорування запитів, які повернули код помилки 420 (флуд) до моменту поки їх не можна буде відправляти знову (коли за короткий проміжок часу здійснюється забагато однакових запитів, Telegram може повернути помилку з кодом 420 та вказати, що наступний запит такого типу можна буде відправити лише через 5-30 секунд; якщо відправити такий же запит до того, як цей час вийде – час очікування може збільшитись геометрично; після виходу часу очікування запит повернеться в чергу);

– повернення інформації про непередбачені виключні ситуації пов’язані із виконанням запиту викликаючому коду без переривання виконання наступних запитів з черги.

Окрім вищевказаного даний модуль також міститиме логіку шифрування та дешифрування файлів паролем (у випадку, якщо користувач захоче захистити свої файли додатково). Шифрування виконуватиметься за алгоритмом AES, реалізація якого в .NET представлена класом Rijndael, а для генерації ключа на основі паролю буде використано вбудований клас Rfc2898DeriveBytes. Даний компонент дозволятиме також отримувати хеш із паролю (який буде зберігатись у базі даних), та верифікувати його.

Окрім вищевказаного, модуль взаємодії з Telegram також надаватиме

простий інтерфейс для створення, керування та відслідковування прогресу виконання спеціалізованих для даного додатку завдань, таких як завдання завантаження файлу, завдання вивантаження файлу та завдання резервного копіювання відразу групи файлів, позбавивши клієнтський код від необхідності будь-якої прямої взаємодії із Telegram Client API та його абстракціями (мовою TL). Для реалізації подібної функціональності буде використано шаблони Facade та Mediator, модифіковані таким чином, щоб завдання на завантаження та вивантаження файлів мали схожий до .NET Task Parallel Library (TPL) інтерфейс для взаємодії. Кожне завдання надаватиме набір подій, відслідковуючи які клієнтський код зможе дізнаватися про початок завантаження або вивантаження файлів, його прогрес та результати. Діаграму класів даного компоненту показано на рисунку 2.4.

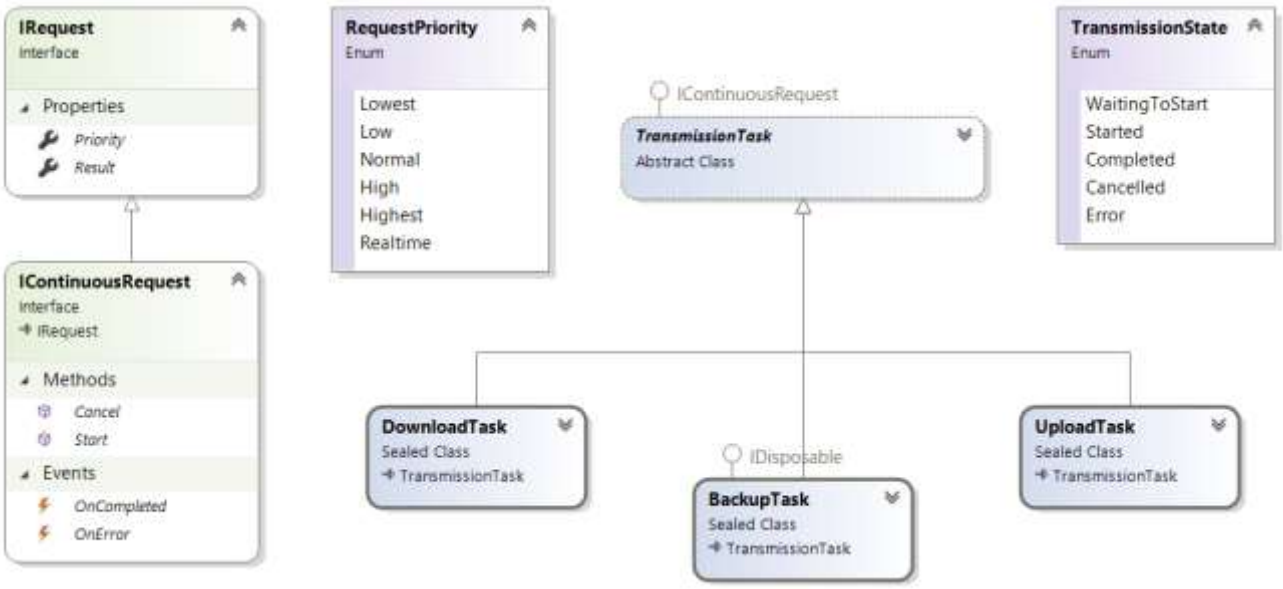


Рисунок 2.4 – Діаграма класів завдань завантаження та вивантаження файлів у Telegram

Абстрактний клас TransmissionTask окрім унаслідованих від IContinuousRequest подій надаватиме також події OnTransmissionStateChanged (можливі стани описано перечисленням TransmissionState на діаграмі), OnStarted та OnProgressChanged. BackupTask у своїй внутрішній реалізації буде

використовувати UploadTask і, окрім всіх вищевказаних подій, визначатиме також подію OnFileUploaded.

Оскільки модуль мережевої взаємодії з Telegram буде досить об’ємним, то щоб не роздувати код основного проекту, було прийнято рішення винести його в окрему .NET Standart бібліотеку під назвою Cloudgram.Base та розробляти незалежно від основного додатку. На даний момент на платформі .NET немає жодної бібліотеки, яка дозволяла б так легко працювати із завантаженням та вивантаженням файлів у Telegram, а використання .NET Standart 2.0 в якості цільового фреймворку дозволить підключати її до будь-яких проектів: від мобільних додатків до програм, які працюватимуть на базі операційних систем сімейства Linux.

До рівня додатка у фреймворці Xamarin зазвичай відносять залежний від платформи код (тобто той код, який не використовується спільно із іншими платформами) або код, який характерний лише для додатка та який не підходить для повторного використання [17]. Сюди будуть відноситись всі сервіси, що використовуватимуться в даному додатку та для яких буде здійснена ін’єкція залежностей, а саме:

– сервіс для роботи з локальними медіа-файлами (отримання інформації про файл, завантаження списку локальних файлів, видалення локального файлу, збереження медіа-файлу в галерею);

– сервіс для читання та збереження параметрів додатку;

– сервіс для взаємодії з Telegram (поставляє класи із переносної .NET Standart бібліотеки Cloudgram.Base);

– сервіс для автоматичного резервного копіювання медіа-файлів у фоновому режимі (використовує попередній сервіс);

– сервіс для роботи із вивантаженими медіа (поставляє класи із модуля доступу до даних);

– сервіс для зміни кольору панелі статусу.

Рівень користувацького інтерфейсу відповідатиме за відображення графічного інтерфейсу та взаємодію з користувачем і включатиме в себе

представлення, контролери представлень та моделі представлення, які будуть описані пізніше.

Діаграму описаних модулів додатка та їх відношень між собою показано на рисунку 2.5.

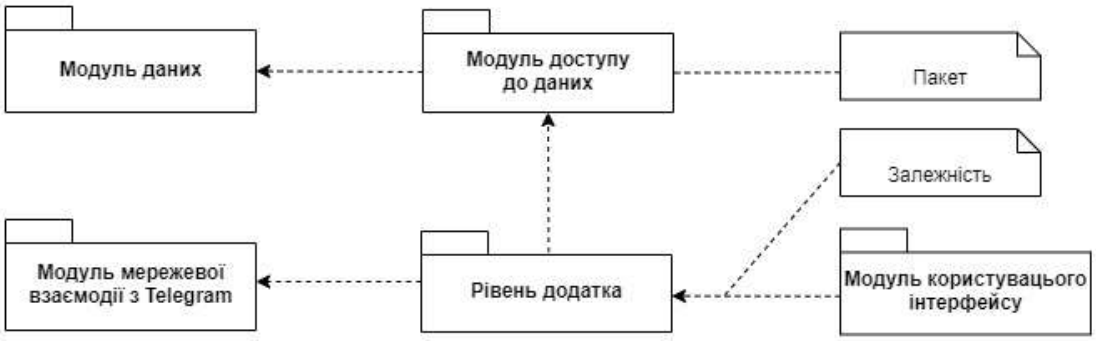


Рисунок 2.5 – Діаграма модулів додатка

2.4 Проектування логічної моделі бази даних

Додаток використовуватиме базу даних для збереження інформації про вивантажені у Telegram медіа-файли. Оскільки додаток у будь-якому випадку генеруватиме мініатюрні іконки для локальних медіа-файлів (для збільшення продуктивності відображення в галереї великої кількості медіа-файлів одночасно), а Telegram створюватиме свої власні лише для невеликих файлів (до 10 МБ), було прийнято рішення вивантажувати в хмару не лише оригінал, але й згенеровану іконку (користувач зможе створити для іконок окремий чат). Окрім того, у користувача буде можливість додавати до кожного такого медіа-файлу теги, щоби потім можна було виконувати по ним пошук. Отже, БД матиме наступні сутності:

– BackupedMedia (містить загальну інформацію про вивантажений файл);

– FileLocation (зберігає інформацію про контекст повідомлення, яким було відправлено медіа-файл);

– ThumbnailLocation (зберігає інформацію про контекст повідомлення, яким було вивантажено іконку);

– Tag (зберігає додані користувачем теги).

Розглянемо кожну з сутностей детальніше.

Сутність BackupedMedia містить наступні атрибути:

– BackupedMediaId (первинний ключ);

– FilePath (шлях до файлу);

– ThumbnailPath (шлях до згенерованої додатком іконки);

– MimeType (визначає тип файлу);

– IsVideo (визначає чи є файл відео);

– Album (локальний альбом в якому було збережено файл);

– CreatedAt (дата створення файлу);

– Size (розмір файлу);

– IsEncrypted (визначає чи файл зашифровано);

– PasswordHash (хеш паролю за допомогою якого було виконано шифрування даного файлу; може бути null у випадку якщо файл не було зашифровано).

Сутність FileLocation містить наступні атрибути:

– FileLocationId (первинний ключ);

– ChatId (ідентифікатор Telegram чату в який було відправлено файл);

– MessageId (ідентифікатор повідомлення, яким було відправлено файл);

– Size (розмір документу; не завжди дорівнює розміру оригінального файлу, оскільки він міг бути розділений на декілька частин);

– BackupedMediaId (зовнішній ключ який позначає, що один файл може бути вивантажений в Telegram декількома повідомленнями);

– ThumbnailLocationId (зовнішній ключ, який позначає, що всі частини вихідного файлу посилаються на одну єдину іконку).

Сутність ThumbnailLocation має майже такі ж самі поля що й FileLocation, за виключенням останніх двох атрибутів. Не дивлячись на ідентичність решти полів, варто відзначити, що це все ще дві різні сутності, які позначають різні

поняття предметної області, а тому об’єднувати їх в одну таблицю було б нелогічно. Більше того, якщо в майбутніх оновленнях доведеться змінити логіку збереження та вивантаження іконок (наприклад, щоби використовувати іконки, які автоматично генерує Telegram), то до таблиці потрібно буде додати ще 3 атрибути, які просто будуть пустими у тих об’єктів, які не є іконками, що тільки ще більше ускладнить програму.

Сутність Tag містить наступні атрибути:

– TagId (первинний ключ);

– TagName (ім’я тега, унікальне значення).

Оскільки один і той самий тег може бути прив’язаний до багатьох файлів, а один файл може мати багато тегів, то між цими сутностями існує відношення багато до багатьох, виражене через проміжну сутність BackupedMediaTag. Діаграму «сутність-зв’язок» показано на рисунку 2.6.

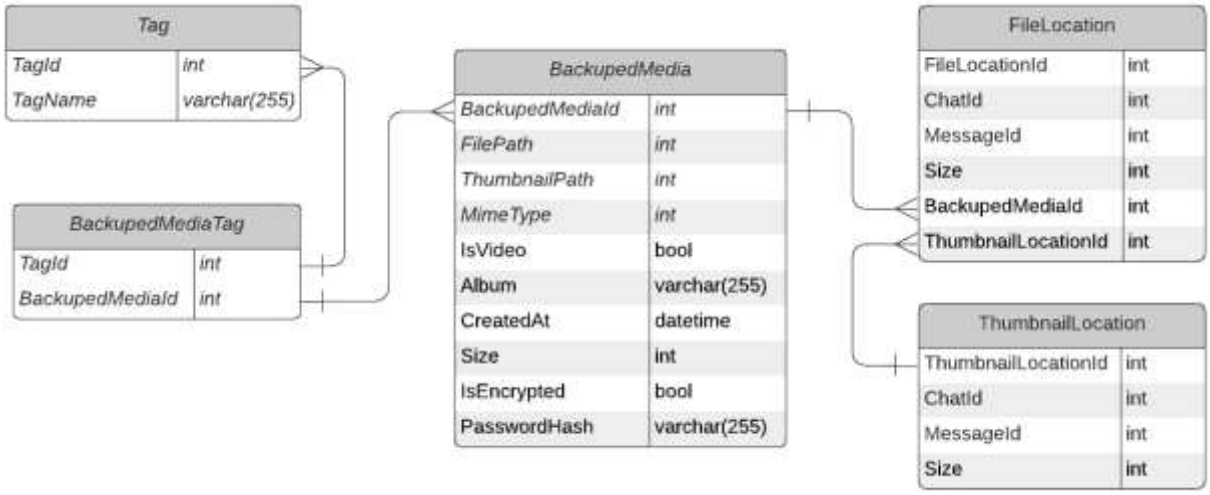


Рисунок 2.6 – Логічна модель БД

2.5 Проектування інтерфейсу користувача

Для оформлення дизайну додатку було обрано три основних кольори: блакитний, білий і чорний. Сам додаток повинен мати максимального легкий та

не перевантажений деталями матеріальний дизайн інтерфейсу. Перш за все, розробимо дизайн для екрану завантаження додатка. Як фон було обрано звичайний білий колір, а по середині екрану розміщено простий логотип блакитного кольору. Під ним знаходиться назва додатку написана великим шрифтом чорного кольору. Оскільки завантаження може зайняти певний час, було передбачено виведення текстової підказки про поточний прогрес завантаження. У разі, якщо з якихось причин додатку не вдасться підключитися до Telegram, колір панелі статусу буде змінено на червоний, а користувачу буде запропоновано спробувати підключитися ще раз або перейти до тонких налаштувань клієнта. Початковий екран додатку Cloudgram показано на рисунку 2.7.

Рисунок 2.7 – Початковий екран додатку під час завантаження (зліва) та у разі виникнення помилки підключення (справа)

Якщо підключення було виконано успішно, а користувач ще не виконував вхід у Telegram із цього додатка, його буде переведено на екран

авторизації. Загальні принципи дизайну не піддалися особливим змінам. Даний екран виконано з дотриманням правил матеріального дизайну, а тому він відчувається легким та не перевантаженим інформацією. На сторінці присутня кнопка виклику короткої довідки, в якій буде пояснено навіщо для роботи з додатком потрібно виконувати вхід в Telegram. Також із даного екрану можна потрапити на сторінку тонких налаштувань клієнта (див. рисунок 2.8).

В загальному, дизайн кожної з опцій, які представлені на сторінці тонких налаштувань, особливо один від одного не відрізняється (і це логічно, оскільки вони розташовані в одному модулі). Кожну таку сторінку можна розділити на декілька частин: заголовок, короткий опис, форма для конфігурації якогось із компонентів клієнту, кнопки які дозволяють зберегти зміни або повернутися назад без їх збереження. Дизайн сторінок налаштування App ID та проксі-сервера показано на рисунку 2.9.

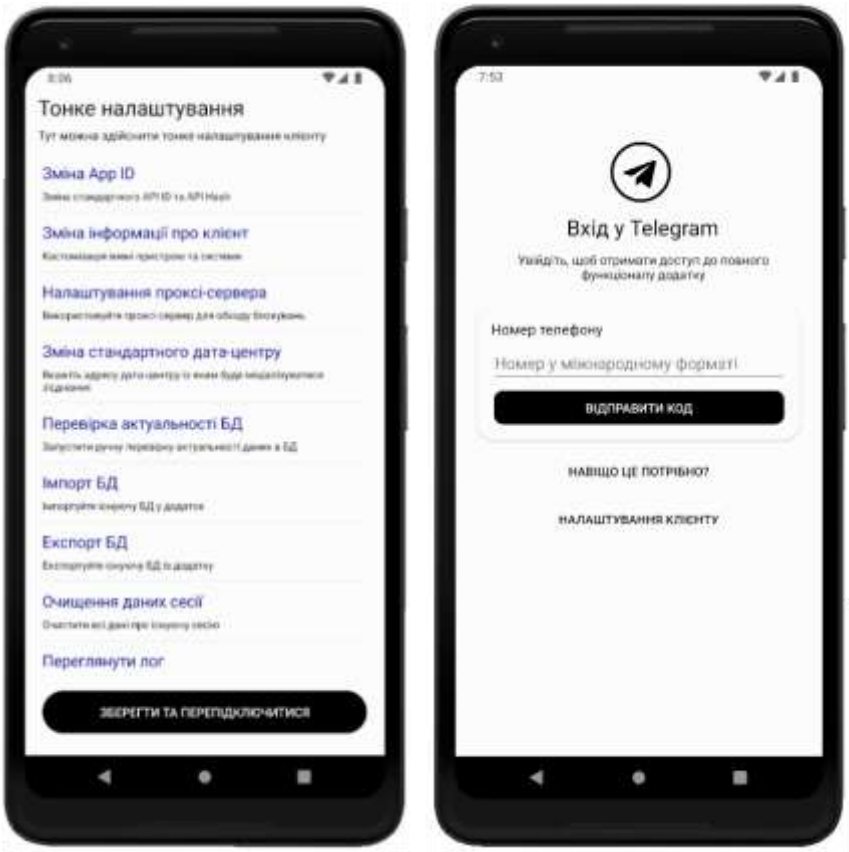


Рисунок 2.8 – Екран тонкого налаштування додатку та входу у Telegram

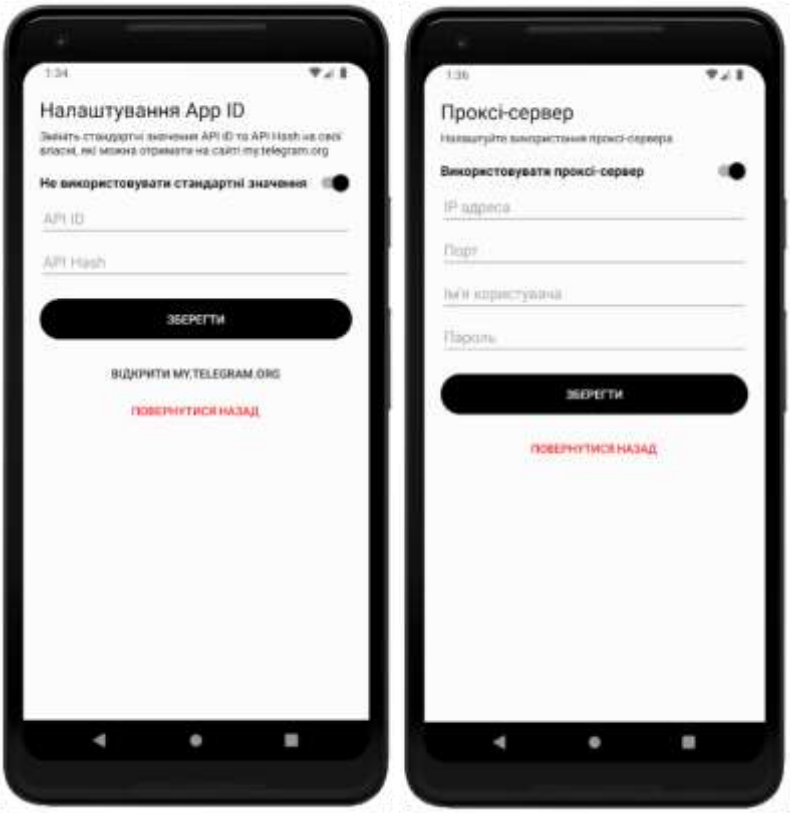


Рисунок 2.9 – Налаштування App ID (зліва) та проксі-сервера (справа)

Після входу для користувача відобразиться головний екран додатка, на якому представлені усі локальні медіа-файли. За замовчуванням вони групуватимуться по даті, однак їх також можна буде сортувати за альбомами та тегами, які були до них прикріплені. Нижнє меню, реалізоване за допомогою вкладок, дозволятиме перемикатися між локальними та вже вивантаженими у хмару медіа-файлами. Окрім того, в нижньому меню також присутня кнопка переходу до екрану пошуку за тегами. Вона знаходиться по середині, відрізняється за дизайном, а також більша за решту кнопок в додатку. Це зроблено для того, щоб користувач за необхідності міг максимально швидко знайти цю кнопку та розпочати пошук. У заголовку сторінки, поруч із кнопкою сортування медіа-файлів, знаходиться кнопка переходу до екрану налаштувань, пов’язаних із акаунтом. Дана кнопка має форму кола в якому знаходиться зображення профілю користувача. Дизайн головного екрану додатка показано на рисунку 2.10.

Рисунок 2.10 – Головний екран додатку

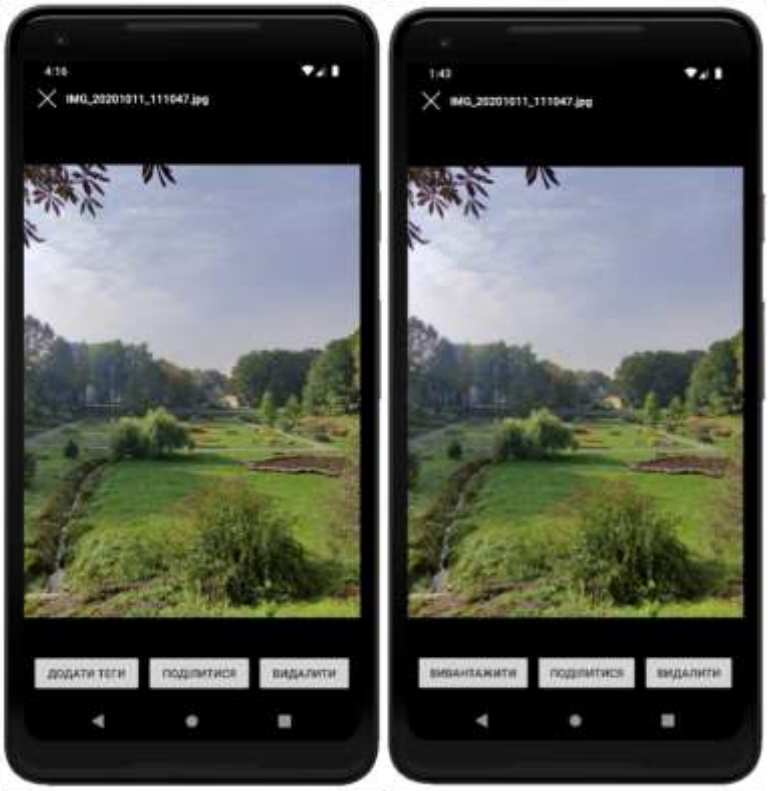


Рисунок 2.11 – Екран перегляду локального (зліва) та вивантаженого раніше (справа) медіа-файлу

При натисканні на медіа-файл відкриватиметься екран на якому можна буде його переглянути у повному розмірі. Нижнє меню буде відрізнятися в залежності від того, чи вивантажений цей додаток в Telegram і чи доступний він на даний момент локально. Якщо файл доступний лише локально, нижнє меню складатиметься із кнопок «Вивантажити», «Поділитися» та «Видалити». Якщо файл вже вивантажений у хмару, то замість кнопки «Вивантажити» буде кнопка «Додати теги». На рисунку 2.11 показано обидва варіанти представлення.

Екран додавання тегів містить коротку інформацію про поточний медіа-файл (назва та альбом) а також його мініатюру. Нижче буде представлено поле для введення нових тегів. Видаляти існуючі можна буде просто торкнувшись до них. Зовнішній вигляд екрану редагування тегів показано на рисунку 2.12.

Рисунок 2.12 – Екран додавання тегів

По доданим тегам користувач може виконувати пошук. Сторінка пошуку має максимально спрощений вигляд та розділена на дві частини: заголовок (у якому є поле для введення тегу, за яким необхідно буде виконувати пошук, та кнопка пошуку) і основна частина, в якій будуть відображатися знайдені медіа-файли, згруповані за тегами. Зовнішній вигляд екрану пошуку представлено на рисунку 2.13.

Рисунок 2.13 – Екран пошуку за тегами

Натиснувши на головній сторінці на кнопку із зображенням свого профілю, користувачу відкриється екран із додатковими налаштуваннями (див. рисунок 2.14). Там можна буде побачити статистику про кількість вивантажених файлів та їх розмір. Окрім того, звідти можна буде потрапити до налаштувань автоматичного резервного копіювання (див. рисунок 2.15) та сторінку з просунутими налаштуваннями клієнта, яка була розглянута раніше.

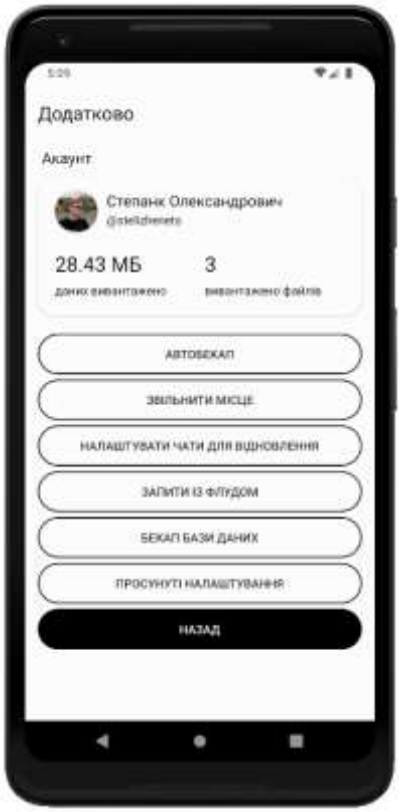


Рисунок 2.14 – Екран із додатковими налаштуваннями

Рисунок 2.15 – Екран налаштування автобекапу

Всього в додатку буде 22 екрани, а їхні назви та схему можливих переходів між ними показано на рисунку 2.16.

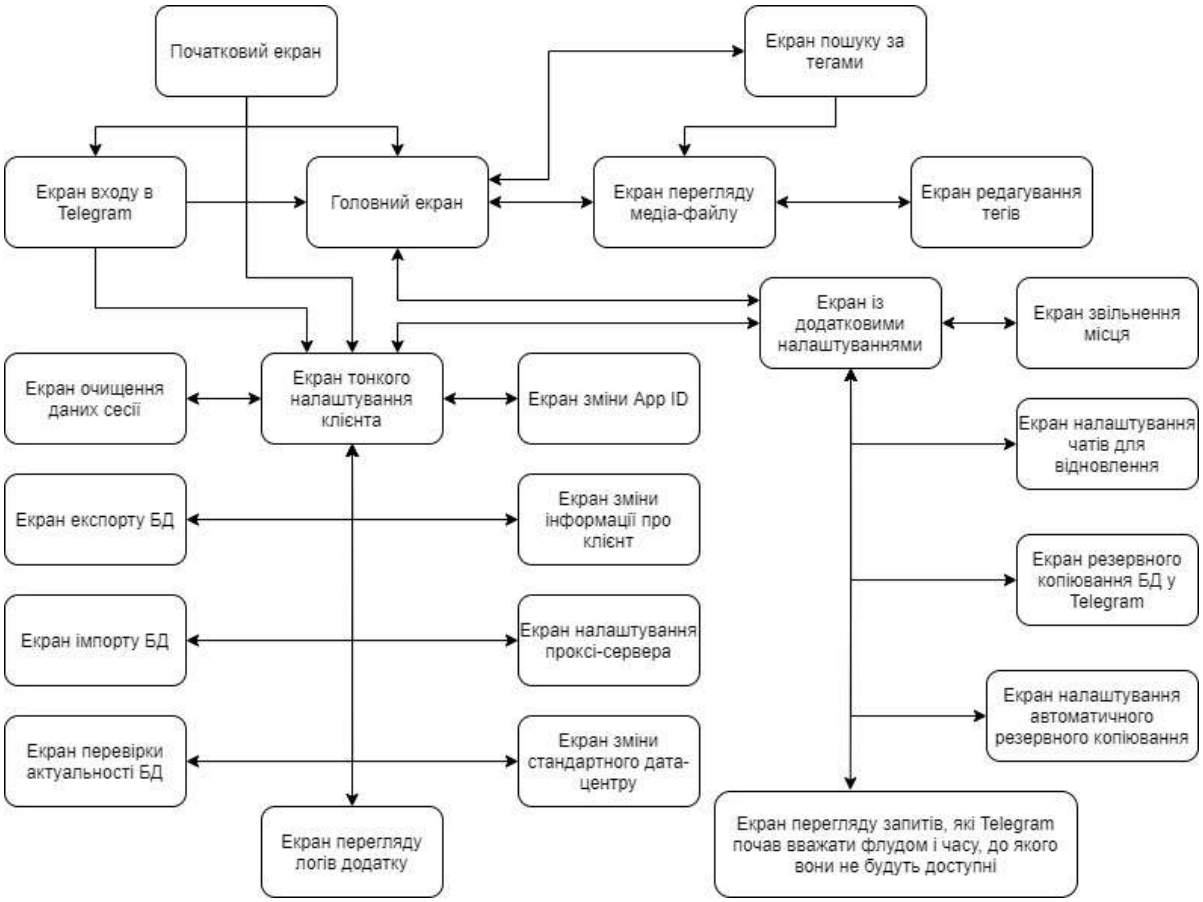


Рисунок 2.16 – Схема переходів між екранами додатку

2.6 Аналіз та вибір технологій і методів реалізації додатка

Як вже зазначалось раніше, додаток розроблятиметься на платформі Xamarin.Forms 5. Це найновіша на даний момент версія фреймворку, який дозволяє розробляти кросплатформні додатки на такі платформи як Android, iOS та Windows з використанням .NET. Це означає що розробник може створити єдиний дизайн інтерфейсу та кодову базу (бізнес-логіку додатку) для проектів на всіх підтримуваних платформах використовуючи при цьому мову C# (спільний для всіх платформ код пишуть в окремому проекті, який називають Shared Project). У разі виникнення необхідності скористатися

специфічної для якоїсь із платформ функціональності, даний фреймворк дає можливість реалізовувати залежний від платформи код окремо та зв'язати його із Shared Project за допомогою механізму ін'єкції залежностей (в Shared Project створюється інтерфейс, який потім реалізовується на конкретній платформі). Для додавання підтримки нової платформи достатньо лише реалізувати інтерфейси, які використовують специфічну для платформи функціональність.

Для збереження даних використовуватиметься база даних SQLite, оскільки вона має чудову продуктивність і є рекомендованою для використання на мобільних пристроях.

Для доступу та взаємодії із БД буде використано Entity Framework Core, який дозволяє абстрагуватися від типу сховища, яке використовується в додатку. Окрім цього, завдяки використанню даного фреймворку, відпадає необхідність написання SQL запитів вручну. Замість цього пропонується використовувати надзвичайно просту та зрозумілу мову LINQ.

Для роботи з Telegram API буде використано дещо модифіковану бібліотеку з відкритим кодом під назвою TgSharp. Ця бібліотека вже містить в собі спроектовану на мову C# схему TL, а також логіку серіалізації та десеріалізації цих об'єктів, що звільняє мене від необхідності виконання цієї рутинної роботи самостійно. TgSharp розповсюджується під ліцензією MIT, розвивається спільнотою, та є наступним етапом розвитку ще однієї клієнтської бібліотеки з відкритим кодом – TLSharp, яка на початковому етапі розроблялась Іллею Пироженко (sochix).

Для відображення списку медіа-файлів використовуватиметься бібліотека FlowListView, яка дозволяє відображати елементи у вигляді сітки, а для створення дизайну вкладок та сторінки редагування тегів буде використано бібліотеки Sharpnado.Tabs та TagEntryView відповідно.

Для того, щоб оптимізувати процес завантаження, кешування та відображення медіа-файлів буде використано бібліотеку glidex.forms.

Отже, в процесі написання розділу було проведено аналіз переваг та недоліків найпопулярніших архітектур мобільних додатків, і встановлено, що

для розроблюваної системи найбільше підходить шаблон MVVM.

Окрім того, було проведено аналіз Telegram API, що дало можливість визначити які саме дані необхідно буде зберігати у базі даних для досягнення цілей, поставлених у завданні до дипломного проектування. Зібрана інформація допомогла спроектувати діаграму «сутність-зв’язок» на якій було відображено схему БД розроблюваного додатка.

У результаті детального аналізу було визначено та описано основні модулі додатка і те, як вони взаємодіють між собою. Отриману інформацію було продемонстровано за допомогою діаграми пакетів.

Окрім вищезгаданого, було також виконано проектування основних елементів користувацького інтерфейсу та проведено їх юзабіліті дослідження. Список всіх екранів, а також способи переходів між ними, було продемонстровано за допомогою схеми переходів.

Примітка. Проектування та дизайн інтерфейсу користувача було виконано методом еволюційного прототипування.

Насамкінець проведений аналіз технологій і засобів, які будуть використовуватись в процесі розробки програмного додатку.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1 Реалізація логіки взаємодії з Telegram API

У ході детального проектування було вирішено винести модуль мережевої взаємодії з Telegram API в окрему бібліотеку. Окрім того, було прийнято рішення використовувати в ній бібліотеку TgSharp, яка вже має все необхідне для роботи із об’єктами схеми TL, їх серіалізації і десеріалізації, а також передачі на сервери Telegram. Основним завданням розроблюваної мною бібліотеки є створення такого інтерфейсу для взаємодії із нею, який дозволив би її користувачам максимально швидко та просто (без вникання в принципи роботи MTProto) керувати виконанням покладених на додаток завдань і відслідковувати результати їх виконання.

Для створення бібліотеки Cloudgram.Base буде використано IDE Visual Studio, а в якості цільової платформи було обрано .NET Standart 2.0. Ця платформа виступає своєрідним спільним знаменником між іншими реалізаціями .NET (.NET Framework, .NET Core, .NET 5+, Mono) та надає доступ до стандартних наборів API-інтерфейсів, які гарантовано будуть підтримуватись у всіх реалізаціях [18]. API для взаємодії з самою бібліотекою надаватиметься через два класи: TgClient та Backuper. Розберемо кожен із них детальніше.

Клас TgClient виступає обгорткою над класом TelegramClient, який поставляється через бібліотеку TgSharp. До проблем, які вирішуються в даному класі належить керування чергою запитів та приховування від клієнтського коду використання мови TL. Ми повернемось до розгляду вирішення даних проблем трохи пізніше.

Клас TelegramClient перебуває у відношенні композиції із класом TgClient, тобто його життєвий цикл як дочірнього об’єкту повністю співпадає із життєвим циклом батьківського (TgClient), а тому цей об’єкт не може використовуватись окремо. Батьківський клас реалізовує шаблон відомий в .NET під іменем Dispose Pattern шляхом імплементації інтерфейсу IDisposable,

що дає можливість вивільняти всі ресурси як тільки об’єкт перестає бути потрібним (в даному випадку він використовується для розриву з’єднання з Telegram та зупинки виконання черги запитів).

Створення екземпляру класу TgClient через конструктор неможливе поза межами бібліотеки (він має модифікатор доступу internal). Оскільки це досить складний об’єкт, для якого можна задати велику кількість налаштувань (App ID, інформацію про клієнт, проксі, стандартний дата-центр, місце збереження файлу сесії і т.д.), було розроблено клас TgClientBuilder, який дозволяє виконувати поетапне конструювання клієнту шляхом використання шаблону будівельник. Далі подано фрагмент його реалізації:

```
public TgClientBuilder SetAppId(AppId appId)
{
	this.appId = appId;
	return this;
}

...

public TgClient Build()
{
	TcpClientConnectionHandler connectionHandler = null;
	if(proxy != null)
	{
		connectionHandler = (address, port) =>
		{
		    ProxyTcpClient proxyClient = new ProxyTcpClient();
		    return proxyClient.GetClient(address, port, proxy.Address,
		        proxy.Port, proxy.Username, proxy.Password);
		};
	}
	TgClient client = new TgClient(appId.ApiId, appId.ApiHash, sessionStore,
		sessionUserId, connectionHandler,
		dcData?.DataCenterIPVersion ?? DataCenterIPVersion.Default,
		clientInfo, dcData?.Address, dcData?.Port ?? 0, logger);
	return client;
}
```

У представленому вище фрагменті коду не показано методи для встановлення інших параметрів клієнта окрім AppId, оскільки вони всі побудовані за однаковим принципом: в метод виду SetX() (де X це назва параметру клієнта) передається аргумент, який присвоюється відповідному полю класу TgClientBuilder. В результаті виконання ці методи повертають

посилання на екземпляр класу, на якому і викликався даний метод. Це дозволить клієнтському коду будувати ланцюги виклику методів для налаштування об’єкту TgClient.

Екземпляр класу-делегата TcpClientConnectionHandler, який створюється в методі Build(), використовується для збереження посилання на функцію, яка повинна встановити TCP з’єднання із дата-центром через проксі-сервер та як результат свого виконання повернути підключений TcpClient.

Діаграму класів, на якій показано клас TgClientBuilder та структури даних з якими він перебуває у відношенні асоціації (через свої поля), продемонстровано на рисунку 3.1.

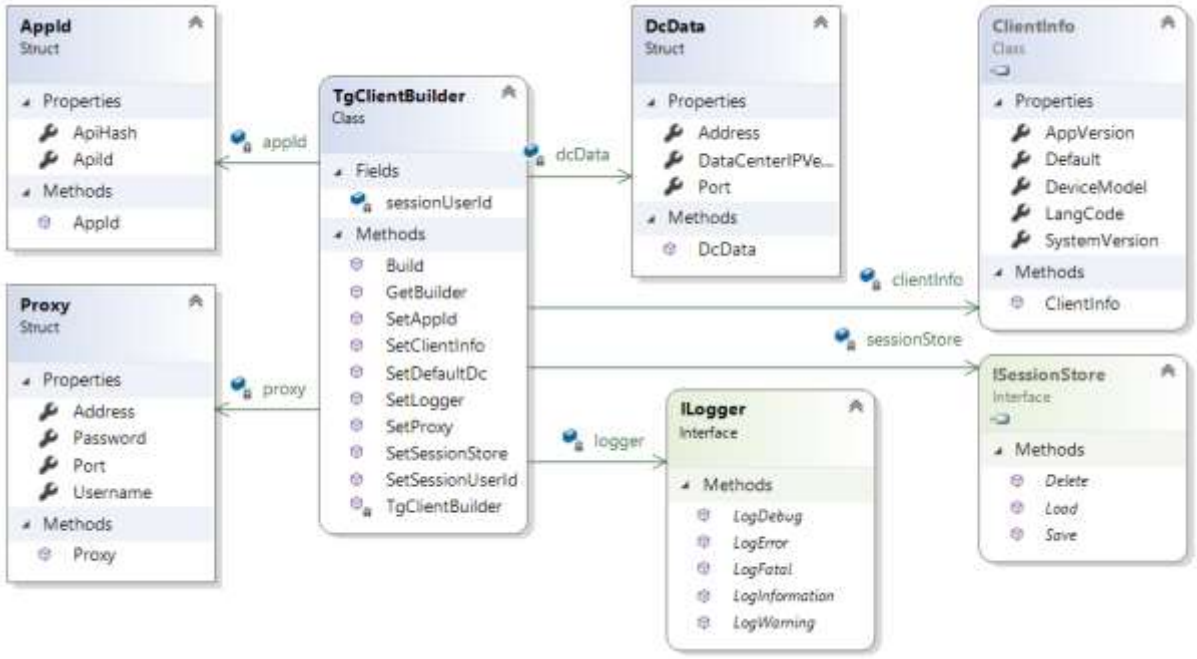


Рисунок 3.1 – Діаграма класів компоненту будівельника клієнта

Імплементації інтерфейсів ILogger та ISessionStore відсутні в бібліотеці Cloudgram.Base, а тому відповідальність за їх реалізацію цілком покладається на клієнтський код (проекти в яких буде використовуватись дана бібліотека). Реалізація ISessionStore дозволяє визначити те, де саме і яким чином будуть зберігатися дані сесії (шлях та спосіб отримання даних із локального чи віддаленого файлу, або, навіть, із непостійної пам’яті). Cloudgram.Base не

прив’язується до якоїсь із бібліотек для логування інформації. У клієнтському проекті може використовуватись будь-яка подібна бібліотека – достатньо лише створити клас-адаптер, в якому слід імплементувати інтерфейс ILogger (відобразити виклики описаних там методів на відповідні методи із використовуваної бібліотеки для логування) та передати його в TgClientBuilder.

Мова C# дозволяє розділяти один клас на декілька файлів, шляхом використання в його оголошенні ключового слова partial. Це допомагає досягти більшої структурованості коду. Скористаємось цією можливістю, та розділимо клас TgClient на наступні файли:

– TgClient.cs;

– TgPriorityRequestExecutor.cs;

– TgUploader.cs;

– TgDownloader.cs.

Для початку розглянемо вміст файлу TgPriorityRequestExecutor.cs. Даний клас відповідає за створення, керування та виконання запитів з черги. Черга запитів представлена класом RequestQueue, а її елементами виступають екземпляри класу PriorityRequest. Клас PriorityRequest імплементує відразу три інтерфейси – IRequest, IComparable та IComparable<PriorityRequest>. Перший приносить в даний клас властивості Priority та Result, а останні два дозволяють порівнювати запити між собою за їх пріоритетом.

Відправка запиту та очікування і обробка відповіді відбувається в окремому потоці із пулу потоків. Бібліотека паралельних завдань (TPL) бере на себе весь процес керування створенням та виконанням потоків для того, щоб це відбувалося максимально швидко та ефективно. Вона надає набір API інтерфейсів, які дозволяють легко створювати та очікувати на результати виконання довготривалих і/або складних операцій, не блокуючи при цьому викликаючий потік [19]. Microsoft рекомендує використовувати асинхронну модель на основі завдань (TAP) для виконання асинхронних операцій [20]. Отже, нам було б достатньо зберігати в класі PriorityRequest лише завдання (екземпляр класу Task із TPL), в якому описано алгоритм дій для створення і

відправки запиту із подальшою обробкою відповіді, та повертати це завдання клієнтському коду, щоби він міг дочекатися результату в асинхронному режимі. Тим не менше, такий підхід має відразу декілька мінусів. По-перше, клієнтський код може безперешкодно викликати метод Start() на отриманому завданні, що призведе до негайного виконання запиту, ігноруючи чергу та створюючи кашу в TCP клієнті, оскільки не буде точно відомо на який саме (із одночасно відправлених через один TCP клієнт) запит прийшла відповідь. Окрім того, коли до такого запиту нарешті дійде черга на виконання, виконавець запитів із черги (який буде розглянуто пізніше) також викличе на цьому завданні метод Start(), що призведе до виключення, оскільки запустити завдання на виконання можна лише один раз. Із останнього випливає ще один мінус описаного підходу, адже це унеможливлює повторну відправку цього ж запиту (після повторного підключення у разі якщо запит не вдалося відправити через відсутність мережі, або після виходу часу очікування на запиті який Telegram порахував за флуд).

Насправді, відправити запит повторно можна просто створивши та запустивши на виконання нове завдання, яке виконуватиме те ж саме, що й попереднє. Головна проблема полягає у тому, що клієнтський код ніколи не дізнається про нове завдання, оскільки він має доступ та може очікувати лише на те завдання, яке не вдалося виконати з першого разу. При спробі отримати результат виконання такого завдання буде згенеровано виключення.

Рішенням для обох проблем виступає можливість створення окремого (фейкового) завдання, яке й буде повертатись клієнту. Таке завдання не міститиме логіки відправки запиту та обробки результатів, запускатиметься на виконання в момент свого створення (унеможливлює безперешкодний виклик Start() із клієнтського коду) і вважатиметься виконаним після завершення виконання основного завдання. У разі виникнення необробленого виключення в основному завданні, це виключення повинне також генеруватися і у фейковому.

На щастя, мені навіть не доведеться створювати свої власні фейкові завдання, оскільки в TPL вже існує клас TaskCompletionSource, який дозволяє досягати точно таких же результатів.

Таким чином, для клієнтського коду все виглядатиме максимально просто (він працюватиме лише з одним завданням, яке поводитиметься саме так, як і було заявлено у вимогах до черги запитів, описаних у попередньому розділі), в той час як «під капотом» реальні завдання у разі потреби можуть безперешкодно і без будь-яких наслідків проходити повторну ініціалізацію.

Додати в чергу новий запит можна через узагальнений метод CreatePriorityRequest<TResponse>(), визначення якого знаходиться у файлі TgPriorityRequestExecutor.cs:

```
internal Task<T> CreatePriorityRequest<TResponse>(Func<object> requestBody, int
priority, string reqType = null)
{
        PriorityRequest priorityRequest = null;
        int requestId = Interlocked.Increment(ref _requestNumber);
        priorityRequest = new PriorityRequest(requestId, requestBody, reqType)
        {
            Priority = priority,
        };

        Task<T> resultingTask = priorityRequest.WaitTask.Task
            .ContinueWith(t => (TResponse)t.Result);
        _requestQueue.Insert(priorityRequest);
        _waitHandler.Set();

        return resultingTask;
}
```

Даний метод має модифікатор доступу internal, що робить його недоступним для прямого звернення із клієнтського коду. Клас Interlocked надає простий інтерфейс для виконання атомарних операцій зі змінними. В даному випадку він використовується для безпечної у відношенні потоків інкрементації лічильника створених запитів. Параметр requestBody має тип класу-делегата Func<object>, що означає що він може посилатися на будь-яку функцію, яка в результаті свого виконання повертатиме деякий об’єкт (очікується, що це буде функція яка міститиме логіку відправки запиту та обробки результату). Поле WaitTask класу PriorityRequest має тип TaskCompletionSource<object> і представляє фейкове завдання. Відповідно, основне та фейкове завдання як результат свого виконання повертатиме тип object. Очікуваний клієнтським кодом тип результату визначається параметром-типом TResponse. TPL дозволяє

створювати ланцюги із завдань, чим ми й скористаємось, встановивши що відразу після завершення фейкового завдання (а воно завершиться як тільки завершиться основне) слід привести результат його виконання до типу TResponse. Для цього скористаємось методом ContinueWith(), що повертає екземпляр Task<TResponse>, який ми і передамо клієнтському коду.

Використання загального типу object як результуючого для внутрішніх завдань (тобто усіх, окрім того, що повертається клієнтському коду) замість конкретного типу TResponse жодним чином не ускладнює код та не уповільнює його роботу, а навіть навпаки. По-перше, усі об'єкти схеми TL представлені класами, а приведення типів посилання відбувається практично миттєво (хоча й в коді ми працюємо із типом object, середовище виконання CLR знає справжній тип даного об'єкту та може легко визначити чи можливе таке приведення). При цьому після приведення типу в керованій купі навіть не виділяється пам'ять під новий об'єкт – в отриманій змінній буде зберігатися посилання на ту ж саму адресу в пам'яті, де й зберігається екземпляр object. По-друге, в .NET існують певні обмеження на сценарії, в яких можуть використовуватись узагальнені типи. Я міг би уникнути використання типу object, визначивши клас PriorityRequest як узагальнений (PriorityRequest<TResponse>), але оскільки в більшості запитів тип TResponse буде відрізнятися, створити одну чергу для всіх запитів було б неможливо. Більше того, для списків не працює коваріантність, що означає що створити список типу List<PriorityRequest<object>> щоб додавати в нього запити з різними типами результату неможливо.

Такий підхід спрацює лише у випадку якщо ми зберігатимемо список запитів використовуючи тип List<object>, але це значно ускладнить виявлення реального типу кожного конкретного запиту. Рішенням виступало б створення базового абстрактного класу для ProrityRequest<TResponse> який надавав би API для встановлення результату виконання основного завдання фейковому (у випадку успіху) та генерації виключення (у випадку невдачі).

Підхід, який використовується на даний момент швидший в реалізації (не

вимагає написання додаткового коду) та нічим не поступається описаному вище в плані продуктивності. Сам же PriorityRequest як компонент не планується для розширення. Однак, якщо коли-небудь виникне така необхідність, архітектура системи дозволяє швидко застосувати описані вище зміни.

Після вставки запиту в чергу, ми сповіщаємо виконувача запитів про це шляхом встановлення об'єкта-події AutoResetEvent в сигнальний стан (викликавши _waitHandler.Set()). Це змушує потік виконувача запитів вийти з режиму очікування та почати відправку запитів із черги (якщо він цим ще не займається на даний момент).

Виконувач запитів працює в окремому потоці та представлений методом RequestExecutor(). Даний метод працює до тих пір, поки його виконання не буде скасовано. Ми можемо перевіряти чи були спроби відмінити дану операцію через об'єкт CancellationToken (використовуючи його властивість IsCancellationRequested), який даний метод отримує як свій параметр. Оскільки ми реалізували в даному класі Dispose Pattern, зупинка виконання запитів з черги відбуватиметься автоматично під час фіналізації екземпляру даного класу, або у випадку коли на його екземплярі буде викликано метод Dispose().

```
private void RequestExecutor(CancellationToken token)
{
    while (!token.IsCancellationRequested)
    {
        _waitHandler.WaitOne();
        while (_requestQueue.Count > 0 && !token.IsCancellationRequested)
        {
            PriorityRequest req = _requestQueue.Pull();

            if(req != null)
            {
                try
                {
                    SendRequest(req);
                }
                catch (Exception ex)
                {
                    if (!req.WaitTask.TrySetException(ex))
                    {
                        logger?.LogFatal($"Error while trying to set exception
for {req}\n\tException: {ex.Message}\n\tStack trace: {ex.StackTrace}");
                    }
                }
            }
```

```
            else if(_requestQueue.Count > 0)
            {
                break;
            }
        }
    }
}
```

Спершу ми очікуємо на об’єкт-подію _waitHandler (як вже було показано раніше, вона з’являється під час вставки нового запиту у чергу). Як тільки вона стається, ми намагаємося опрацювати всі запити, які на даний момент перебувають у черзі. Приватне поле _requestQueue є екземпляром класу RequestQueue та представляє чергу запитів. Даний клас надає API для вставки та отримання запитів з черги, відповідно до їх пріоритету. Окрім того, він дозволяє отримати інформацію про ті запити, які Telegram почав вважати флудом і надає інформацію про те, до якого моменту часу вони будуть недоступними для повторної відправки.

Ми отримуємо наступний запит з пріоритетної черги через виклик _requestQueue.Pull(), який як результат свого виконання повертає екземпляр класу PriorityRequest (з найвищим на даний момент пріоритетом) або null (у випадку якщо у черзі на даний момент немає жодного запиту чи якщо там перебувають лише ті запити, які Telegram на даний момент вважає за флуд). Якщо отриманий запит не рівний null, то він передається у метод SendRequest(), код якого представлено нижче.

```
private void SendRequest(PriorityRequest req)
{
    if (req == null) return;
    bool completed = false;

    while (!completed)
    {
        try
        {
            logger?.LogInformation($"Starting to execute {req}");

            CheckConnection();

            req.RequestExecutor.Start();
            object result = req.RequestExecutor.GetAwaiter().GetResult();
            if (req.Exception != null)
            {
```

```
                throw req.Exception is AggregateException ae ?
                    ae.InnerException : req.Exception;
            }
            req.WaitTask.SetResult(result);

            completed = true;
            lastSuccessfulRequest = DateTime.Now;
            logger?.LogInformation($"Completed execution of {req}");
        }
        catch (Exception ex)
        {
            ex = ex is AggregateException ae ? ae.InnerException : ex;
            if (ex is FloodException fe)
            {
                if (req.RetriesCount > 0)
                {
                    logger?.LogError($"Flood performing RETRY of {req}\
                        $"\tThis request WILL NOT sent!\n" +
                        $"\tClougram.Base will automatically wait
{fe.Ti.TotalSeconds} seconds " +
                        $"before sending next request of the same type");

                    req.WaitTask.SetException(fe);
                    AddFloodWaitedRequest(req, fe);
                    break;
                }

                logger?.LogError($"Flood performing {req}\n" +
                        $"\tWaiting {fe.TimeToWait.TotalSeconds} secondry");

                AddFloodWaitedRequest(req, fe);
                req.ReinitExecutorTask();
                _requestQueue.Insert(req);
                completed = true;
            }
            else if (ex is SocketException || ex is ClientNotConnectedException)
            {
                OnConnectionError(req, ex);
            }
            else
            {
                completed = true;
                req.WaitTask.SetException(ex);
                logger?.LogError($"Error performing {req}\n" +
                        $"\tException: {ex.Message}\n" +
                        $"\tStack trace: {ex.StackTrace}");
            }
        }
    }
}
```

Спершу виконується перевірка наявності підключення до Telegram. Якщо воно існує, то запит ставиться на виконання. Отриманий результат встановлюється для фейкового завдання шляхом виклику req.WaitTask.SetResult(). Якщо ж в процесі виконання запиту виявиться що з’єднання було втрачено, керування буде передано методу OnConnectionError().

```
private void OnConnectionError(PriorityRequest req, Exception ex)
{
    ConnectionStatus = ConnectionStatus.NotConnected;
    logger?.LogError($"Reconnecting: connection error\n" +
        $"\tException: {ex.Message}\n" +
        $"\tStack trace: {ex.StackTrace}");

    try
    {
        CheckConnection(beSureToTryReconnect: true);
    }
    catch (ClientNotConnectedException e)
    {
        ConnectionStatus = ConnectionStatus.WaitingToReconnect;

        logger?.LogError($"Connection error while reconnecting\n" +
            $"\tException: {e.Message}\n" +
            $"\tStack trace: {e.StackTrace}");

        logger?.LogWarning("Waiting for manual reconnection");
        _waitingToReconnect = true;
        _reconnectionEvent.WaitOne();
        _waitingToReconnect = false;
        logger?.LogInformation("Client was reconnected manually. Retry…");
    }

    req.ReinitExecutorTask();
    logger?.LogInformation($"Resending {req}");
}
```

Спершу буде зроблено спробу автоматичного повторного підключення. Якщо вона буде невдалою, відкритому полю ConnectionStatus, яке представляє однойменний тип перечислення, буде присвоєно значення ConnectionStatus.WaitingToReconnect. Клієнтський код може слідкувати за змінами цього поля підписавшись на подію OnConnectionStatusChanged. Очікується, що у випадку втрати з’єднання користувачу буде запропоновано встановити підключення вручну. Коли це станеться об’єкт-подія _reconnectionEvent перейде в сигнальний стан і виконувач запитів продовжить свою роботу. Він повторно ініціалізує основне завдання через виклик req.ReinitExecutorTask() після чого керування буде передано назад – методу SendRequest(). Оскільки локальна булева змінна completed все ще матиме значення false (що означає що запит ще не виконався), то буде виконано повторну спробу відправки даного запиту.

Якщо ж Telegram почне вважати відправку даного запиту за флуд, то його все одно буде ініціалізовано повторно і він повернеться назад у чергу з

відповідною приміткою (із зазначенням часу до якого даний запит не можна відправляти повторно). У разі виникнення будь-яких інших виключних ситуацій, виконувач запитів не намагатиметься надіслати проблемний запит повторно, а повідомить про це клієнтський код шляхом встановлення отриманого виключення результатом виконання фейкового завдання. Зробити це можна через виклик методу req.WaitTask.SetException().

Тепер, коли нами було розібрано процес роботи черги запитів, розглянемо процес створення конкретних запитів. Для цього перейдемо до огляду файлу TgClient.cs. В ньому містяться описи методів, які дозволяють уникнути роботи з TL схемою запитів із клієнтського коду. Код методу, який дозволяє отримати інформацію про поточного користувача показано нижче.

```
public Task<TLUserFull> GetMeAsync(RequestPriority priority =
    RequestPriority.Normal)
{
    return CreatePriorityRequest<TLUserFull>(async () =>
    {
        var req = new TLRequestGetFullUser { Id = new TLInputUserSelf() };
        var tlUserFull = await SendAuthenticatedRequestAsync<TLUserFull>(req,
token);
        var tlUser = (TLUser)tlUserFull.User;
        return tlUserFull;
    }, priority);
}
```

У цьому файлі також містяться визначення для таких методів як SendCodeRequestAsync(), SignInViaCodeAsync(), GetUserPhotoAsync(), GetUserDialogsAsync(), LogoutAsync(), ClearSession(). Повний код даного файлу буде наведено у додатку Б.

У файлі TgDownloader.cs описано методи які спрощують процес завантаження файлів з Telegram. Даний файл містить лише один відкритий метод, код якого представлено нижче.

```
public async Task<byte[]> DownloadAsync(IEnumerable<FileContext> fileContexts,
    Action<ProgressChangedEventArgs> progressReporter = null,
    RequestPriority priority = RequestPriority.Normal,
    CancellationToken token = default)
{
    token.ThrowIfCancellationRequested();
```

```
    int chunkSize;
    int totalSize = fileContexts.Sum(fc => fc.Size);
    progressReporter?.Invoke(new ProgressChangedEventArgs(null, 1, 1, 0, e));

    using (MemoryStream ms = new MemoryStream())
    using (BinaryWriter bw = new BinaryWriter(ms))
    {
        foreach (var fileContext in fileContexts)
        {
            chunkSize = KbPerChunk * 1024;
            await DownloadAsync(fileContext, bw, chunkSize, Length, totalSize,
                progressReporter, priority, token);
        }

        return ms.ToArray();
    }
}
```

У цього перевантаження методу DownloadAsync() є тільки один обов'язковий параметр – перечислення контекстів файлів, які необхідно завантажити. Контекстів може бути багато, оскільки файл при вивантаженні у Telegram міг бути розбитим на декілька частин. Клас FileContext містить ідентифікатор файлу, хеш доступу та посилання на завантажуваний файл (звідки беруться ці поля та навіщо вони потрібні було описано у попередньому розділі). Параметр progressReporter має тип класу-делегата Action та може використовуватись для відслідковування прогресу виконання завантаження на основі механізму зворотнього виклику. Логіка роботи методу доволі проста – для кожного контексту ми викликаємо приватне перевантаження методу DownloadAsync(), який і здійснює поетапне завантаження файлу. Код даного приватного методу представлено нижче.

```
private async Task DownloadAsync(FileContext fileContext, BinaryWriter bw,
    int partSize, int downloadedSize, int totalSize,
    Action<ProgressChangedEventArgs>   progressReporter,
    RequestPriority priority, CancellationToken token)
{
    token.ThrowIfCancellationRequested();

    int fileBytesLength;
    int offset = 0;
    var idfl = new TLInputDocumentFileLocation
    {
        Id = fileContext.FileId,
        AccessHash = fileContext.AccessHash,
        FileReference = fileContext.FileReference,
        ThumbSize = string.Empty,
```

```
    };

    do
    {
        var file = await GetFile(idfl, partSize, offset, priority, token)
        if (file == null) break;

        fileBytesLength = file.Bytes.Length;
        offset += fileBytesLength;
        bw.Write(file.Bytes);
        progressReporter?.Invoke(new ProgressChangedEventArgs(null, 1, 1,
            downloadedSize + offset, totalSize));
    } while (fileBytesLength >= partSize);
}
```

У файлі TgUploader.cs описано допоміжні методи для вивантаження файлів у Telegram. У ньому міститься лише один відкритий метод, код якого представлено нижче.

```
public async Task<IEnumerable<FileContext>> UploadAsync(FileData inputFileData,
            Action<ProgressChangedEventArgs> progressReporter = null,
            RequestPriority priority = RequestPriority.Normal,
            CancellationToken token = default)
{
    token.ThrowIfCancellationRequested();

    byte[] fileBytes;
    List<FileContext> fileContexts;

    if (inputFileData.Data != null)
    {
        fileBytes = inputFileData.Data;
    }
    else
    {
        fileBytes = File.ReadAllBytes(inputFileData.FilePath);
    }

    if(inputFileData.Thumbnail == null)
    {
        if(inputFileData.ThumbnailPath != null)
        {
            inputFileData.Thumbnail =
File.ReadAllBytes(inputFileData.ThumbnailPath);
        }
    }

    List<Queue<byte[]>> parts = GetFileParts(fileBytes);
    List<TLAbsInputFile> inputFiles = await UploadFileParts(parts,
inputFileData, progressReporter, priority, token);
    fileContexts = await SendFiles(inputFileData, inputFiles, priority,token);

    return fileContexts;
}
```

Єдиний обов’язковий параметр inputFileData надає інформацію про вивантажуваний файл. Допоміжний приватний метод GetFileParts() розбиває файл на частини, які потім будуть вивантажені на сервери Telegram через метод UploadFileParts(). Після завершення процедури вивантаження ці файли можуть бути відправлені у резервний чат шляхом виклику методу SendFiles(). Повний код файлу TgUploader.cs буде представлено у додатку Б.

Клас Backuper в свою чергу є оболонкою для класу TgClient (вони перебувають у відношенні агрегації). Він надає API для створення й налаштування завдань передачі даних (Transmission Tasks), загальні принципи роботи яких розкривалися у попередньому розділі. Якщо методи описані у файлах TgUploader.cs та TgDownloader.cs повертали завдання типу Task (із бібліотеки TPL), завершення якого клієнтський код може дочекатися використавши ключове слово await, то методи для завантаження і вивантаження файлів із класу Backuper повертають екземпляри-наслідники класу TransmissionTask. Даний абстрактний клас описує основні методи для керування завданнями завантаження та вивантаження як одиночних, так і відразу декількох файлів. Клас TransmissionTask імплементує інтерфейс IContinuousRequest (який у свою чергу наслідується від IRequest), що позначає всі комплексні операції (тобто такі, для виконання яких може знадобитися відправити декілька запитів, а їх виконання може зайняти певний час). Всі класи, які імплементують даний інтерфейс, надають методи для початку виконання операції та її відміни, а також події OnCompleted та OnError. Окрім вищевказаного, усі завдання передачі даних описують також події OnStarted, OnProgressChanged та OnTransmissionStateChanged. Підхід з використанням подій набагато гнучкіший та зручніший за механізм зворотнього виклику (який використовується методами із файлів TgUploader.cs і TgDownloader.cs) для повідомлення прогресу виконання операції і її статусу, та дозволяє писати простий та добре структурований клієнтський код. Однак, це далеко не єдина причина чому завдання передачі даних можуть виявитись більш зручними для використання. Розпочнемо розгляд абстрактного класу TransmissionTask із його

конструктора, код якого представлено нижче.

```
public TransmissionTask(TgClient client,
            RequestPriority transmissionTaskPriority,
            SynchronizationContext synchronizationContext = null,
            CancellationTokenSource cts = null)
{
    this.client = client;
    _synchronizationContext = synchronizationContext;
    Priority = transmissionTaskPriority;

    if(cts == null)
    {
        this.cts = new CancellationTokenSource();
    }
    else
    {
        this.cts = cts;
    }

    cancellationToken = this.cts.Token;

    SetMainTask();
    SetCompletingTask();
}
```

Перш за все, необхідно відзначити що усі поля даного класу мають модифікатор доступу protected (окрім _synchronizationContext, який оголошено із ключовим словом private), що робить їх доступними для наслідників даного класу. Єдиний створений конструктор приймає екземпляр клієнта (який буде використовуватись для відправки запитів), пріоритет операції, та два не обов'язкових параметри: контекст синхронізації та джерело токенів відміни (яке надає інтерфейс для отримання нового токену та скасування усіх операцій, які використовують згенерований даним джерелом токен). Контекст синхронізації дозволяє запам'ятати контекст із якого створювався даний клас для того, щоб генерувати події у тому ж самому потоці, контекст якого було захоплено. Це надзвичайно корисно, особливо якщо дана бібліотека буде використовуватись при розробці програм із графічним інтерфейсом користувача (GUI). В такому випадку клієнтський код може передати в цей конструктор контекст UI потоку, і хоча всі завдання передачі даних будуть виконуватись у якомусь іншому потоці, сам процес генерації та обробки подій виконуватиметься виключно в контексті UI потоку. В іншому випадку, якби код обробки події намагався оновити якийсь

із елементів GUI, ми б отримали виключення, оскільки оновленням елементів GUI може займатися лише той потік, який їх створював. Після ініціалізації полів класу відбувається послідовний виклик методів SetMainTask() та SetCompletingTask(). Обидва методи оголошені із ключовим словом protected, але перший – абстрактний (що означає що він на даний момент не має реалізації та повинен бути перевизначений дочірніми класами), а другий – віртуальний (має реалізацію, але дочірні класи можуть її перевизначити у разі виникнення такої необхідності). Метод SetCompletingTask() дозволяє визначити шлях, за яким продовжиться виконання програми на основі результатів виконання завдання, яке було створено у методі SetMainTask():

```
protected virtual void SetCompletingTask()
{
    transmissionTask.ContinueWith(tt =>
    {
        if (tt.Status == TaskStatus.RanToCompletion)
        {
            State = TransmissionState.Completed;
            RaiseCompleted();
        }
        else if (tt.Status == TaskStatus.Canceled)
        {
            State = TransmissionState.Cancelled;
            RaiseError(tt.Exception);
        }
        else if (tt.Status == TaskStatus.Faulted)
        {
            State = TransmissionState.Error;
            RaiseError(tt.Exception);
        }
    });
}
```

Методи RaiseCompleted() та RaiseError() генерують відповідні події у захопленому раніше контексті:

```
protected virtual void RaiseCompleted()
{
    if (_synchronizationContext != null)
    {
        _synchronizationContext.Post(_ => OnCompleted?.Invoke(this,
EventArgs.Empty), null);
    }
    else OnCompleted?.Invoke(this, EventArgs.Empty);
}
```

Публічний метод Start() запускає на виконання завдання передачі даних (лише у випадку якщо воно не запускалось раніше):

```
public virtual void Start()
{
    if (State != TransmissionState.WaitingToStart)
    {
        throw new TransmissionTaskAlreadyStartedException(this);
    }
    transmissionTask.Start();
}
```

Клас UploadTask наслідується від класу TransmissionTask та представляє завдання вивантаження одиночного файлу у Telegram. Конструктор даного класу, окрім параметрів необхідних для передачі в конструктор TransmissionTask, приймає екземпляр класу FileData, який вже розглядався раніше. Перевизначення методу SetMainTask() в цьому класі виглядає наступним чином:

```
protected override void SetMainTask()
{
    transmissionTask = new Task<UploadResult>(() =>
    {
        cancellationToken.ThrowIfCancellationRequested();
        State = TransmissionState.Started;
        RaiseStarted();

        var fileContexts = client.UploadAsync(_inputFileData,
RaiseProgressChanged, Priority, cancellationToken).Result;
        var uploadResult = new UploadResult(this, fileContexts);
        Result = uploadResult;
        return uploadResult;
    });
}
```

Метод RaiseProgressChanged передається в UploadAsync в якості методу для зворотнього виклику (який у свою чергу генерує відповідну подію в захопленому контексті потоку). Результатом виконання даного завдання є екземпляр класу UploadResult, який містить перечислення контекстів усіх вивантажених файлів (екземплярів класу FileContext).

Оскільки конструктори всіх завдань для передачі даних оголошені з використанням ключового слова public, клієнтський код може створювати їх

напряму. Тим не менше, це може бути не зручно через кількість параметрів, які вони приймають: всього чотири аргументи, два з яких в більшості випадків будуть однаковими (клієнт та контекст синхронізації). Тому рекомендується створювати завдання передачі даних користуючись API класу Backuper (конструктор якого якраз і приймає два останніх аргументи, які в подальшому будуть використовуватись для створення завдань). Це дозволить створювати нові завдання передаючи у фабричний метод лише один обов'язковий аргумент. Наприклад, код методу для створення задачі на відправку файлу в Telegram виглядає наступним чином:

```
public UploadTask CreateUploadTask(FileData file,
    RequestPriority priority = RequestPriority.Normal)
{
    var uploadTask = new UploadTask(_client, file, priority,
_synchronizationContext);
    return uploadTask;
}
```

Повний код основних класів із бібліотеки Cloudgram.Base можна переглянути у додатку Б.

3.2 Реалізація логіки мобільного додатка

Після створення нового рішення за шаблоном Xamarin.Forms, Visual Studio автоматично створить всі необхідні проекти в залежності від того, які платформи необхідно підтримувати. Оскільки на даний момент необхідна лише підтримка платформи Android, буде згенеровано два проекти: Cloudgram (головна бібліотека в якій буде весь незалежний від платформи код) та Cloudgram.Android (проект, в який можна буде додавати специфічний для даної платформи код). Фреймворк Xamarin дозволяє створювати кросплатформні користувацькі інтерфейси прямо в проекті головної бібліотеки, однак для цього йому необхідний певний час на ініціалізацію. Щоб користувач протягом цього часу бачив не просто пустий екран, а екран завантаження, створимо його у

проекті Android. Покладемо також на цю активність (SplashActivity) відповідальність за перевірку необхідних для роботи додатка дозволів, завантаження користувацьких налаштувань та підключення до Telegram. Ця логіка не буде залежати від платформи, та буде описана в головній бібліотеці. Для цього створимо там окремий клас під назвою InitTasks, а у SplashActivity лише викликатимемо відповідні методи на екземплярі даного класу та відображатимемо результати користувачу. Це дозволить зменшити обсяги коду контролера діяльності та значно спростить процес розробки екранів завантаження для інших платформ (якщо в подальшому виникне така необхідність), оскільки там ми зможемо використовувати все той же клас InitTasks. Фрагмент коду SplashActivity, в якому відбувається початкова ініціалізація додатку:

```
private async Task StartupWork()
{
    await LoadLoggerAsync();
    bool permissionsGranted = await CheckPermissionsAsync();
    if (permissionsGranted)
    {
        await LoadSettingsAsync();
        await ConnectToTelegramAsync();
    }
    else
    {
        OnPermissionsCheckFailed();
    }
}

private async Task LoadLoggerAsync()
{
    logger = await initTasks.LoadLoggerAsync();
}

private async Task<bool> CheckPermissionsAsync()
{
    status.Text = "Перевірка дозволів...";
    retryButton.Visibility = ViewStates.Gone;
    bool result = await initTasks.CheckPermissionsAsync();
    return result;
}

private async Task LoadSettingsAsync()
{
    status.Text = "Завантажую налаштування...";
    settingsProvider = await initTasks.LoadSettingsAsync();
}

private async Task ConnectToTelegramAsync()
```

```
{
    DependencyService.Get<AbsStatusBarColorChanger>()?.SetDefault();
    retryButton.Visibility = ViewStates.Gone;
    settingsButton.Visibility = ViewStates.Gone;
    status.Text = "Підключення до Telegram...";
    try
    {
        TgClient client = await
initTasks.ConnectToTelegramAsync(settingsProvider, logger);
        status.Text = $"Підключено";
        Intent intent = new Intent(this, typeof(MainActivity));
        if (client.IsUserAuthorized)
        {
            intent.PutExtra(NEXT_PAGE_KEY, (int)PageToLoadNext.MainPage);
        }
        else
        {
            intent.PutExtra(NEXT_PAGE_KEY, (int)PageToLoadNext.LoginPage);
        }
        StartActivity(intent);
    }
    catch(ClientNotConnectedException ex)
    {
        DependencyService.Get<AbsStatusBarColorChanger>()?.SetErrorView();
        userShouldRetry = UserShouldRetry.ClientConnection;
        status.Text = $"Помилка під час підключення до Telegram";
        retryButton.Visibility = ViewStates.Visible;
        settingsButton.Visibility = ViewStates.Visible;
        logger.LogError($"[{DEBUG_TAG}] Error while connecting to Telegram\n" +
            $"\tError message: {ex.Message}\n" +
            $"\tStack trace: {ex.StackTrace}");
    }
}
```

Метод StartupWork() здійснює послідовний виклик методів для початкової ініціалізації додатка, які у своїй внутрішній реалізації викликають методи з класу InitTasks та сповіщають користувача про прогрес виконання операцій. Для підключення до Telegram у відповідний метод потрібно передати екземпляр класу що імплементує ISettingsProvider (надає API для збереження та отримання користувацьких налаштувань додатка) та клас-адаптер, який імплементує вже описаний раніше інтерфейс ILogger. Після підключення відбувається перевірка чи користувач входив у Telegram через цей додаток раніше та чи ця сесія досі активна. Залежно від результату цієї перевірки, користувача буде перенаправлено або на головну сторінку додатку, або на сторінку входу в Telegram. Розглянемо детальніше фрагмент коду класу InitTasks, який відповідає за встановлення з’єднання з серверами Telegram.

```
public async Task<TgClient> ConnectToTelegramAsync(ISettingsProvider
    settingsProvider, ILogger logger)
{
    AppId appId = default;
    if (settingsProvider?.UseCustomAppId == true)
    {
        appId = settingsProvider?.GetAppId();
    }
    ClientInfo clientInfo = null;
    if (settingsProvider?.UseCustomClientInfo == true)
    {
        clientInfo = settingsProvider?.GetClientInfo();
    }
    string sessionUserId;
    if (invokedFromService)
    {
        sessionUserId = "backuper_session";
    }
    else
    {
        sessionUserId = settingsProvider?.GetSessionUserId() ?? "session";
    }
    Proxy proxy = null;
    if (settingsProvider?.UseProxy == true)
    {
        proxy = settingsProvider?.GetProxy();
    }
    DcData dcData = null;
    if (settingsProvider?.UseCustomDc == true)
    {
        dcData = settingsProvider?.GetDefaultDc();
    }

    ISessionStore sessionStore = new FileSessionStore();
    TgClient client = TgClient.GetBuilder()
        .SetAppId(appId)
        .SetClientInfo(clientInfo)
        .SetSessionUserId(sessionUserId)
        .SetProxy(proxy)
        .SetDefaultDc(dcData)
        .SetSessionStore(sessionStore)
        .SetLogger(logger)
        .Build();
    client.OnConnectionStatusChanged += Client_OnConnectionStatusChanged;
    await client.ConnectAsync();
    client.OnConnectionStatusChanged -= Client_OnConnectionStatusChanged;
    return client;
}

private void Client_OnConnectionStatusChanged(object sender,
    ConnectionStatusChangedEventArgs e)
{
    if (e.Status == ConnectionStatus.WaitingToReconnect)
    {
        (sender as TgClient).OnConnectionStatusChanged -=
Client_OnConnectionStatusChanged;
        throw new ClientNotConnectedException();
    }
}
```

Спершу ми отримуємо з користувацьких налаштувань все необхідне для

побудови екземпляру класу TgClient через TgClientBuilder. Після цього підписуємось на подію OnConnectionStatusChanged щоб зрозуміти коли черга виконання запитів зупиниться по причині неможливості підключення (вона очікуватиме на повторне підключення в ручному режимі). В такому випадку буде згенеровано виключення, яке ми й відловлюємо в методі ConnectToTelegramAsync() класу SplashActivity та пропонуємо користувачу спробувати підключитись заново.

При написанні залежного від платформи коду слід дотримуватись її угод про найменування. Саме тому назва екрану завантаження в Android проекті завершується словом Activity. Незалежні від платформи елементи також мають свої правила найменування елементів. Наприклад, екрани додатку в проектах Xamarin.Forms прийнято називати сторінками, а їх імена повинні завершуватись словом Page. Усі інші екрани додатку, окрім вже описаного екрану завантаження, будуть створюватись в головному проекті.

Для початку розглянемо екран входу в Telegram, який представлений сторінкою LoginPage. Весь процес входу можна розділити на три етапи: етап введення номеру телефону, коду з SMS та паролю (застарілий на даний момент спосіб входу). Ми відображатимемо та змінюватимемо властивості тих чи інших елементів користувацького інтерфейсу в залежності від етапу, на якому в даний момент перебуває користувач. Визначатимемо що саме потрібно робити при натисканні на кнопки переходу в залежності від поточного етапу:

```
void loginButton_Clicked(object sender, EventArgs e)
{
   switch (currentLoginStage)
   {
       case LoginStage.PhoneEntry:
           PhoneEntryStageLoginButtonClicked();
           break;
       case LoginStage.CodeEntry:
           CodeEntryStageLoginButtonClicked();
           break;
       case LoginStage.PasswordEntry:
           PasswordEntryStageLoginButtonClicked();
           break;
   }
}
```

Код методу відправки запиту на отримання коду:

```
async void PhoneEntryStageLoginButtonClicked()
{
    phone = phoneEntry.Text?.Trim();
    if (string.IsNullOrEmpty(phone) || phone.Length < 10)
    {
        await DisplayAlert("Помилка", "Введіть коректний номер", "OK");
        return;
    }

    loginButton.IsEnabled = false;
    loginButton.Text = "Зачекайте...";

    try
    {
        hash = await client.SendCodeRequestAsync(phone);
        SetLoginStage(LoginStage.CodeEntry);
    }
    catch(AggregateException ae) when (ae.InnerException is
ClientNotConnectedException)
    {
        await DisplayAlert("Помилка", "Не вдалося з'єднатися із Telegram. " +
            "Перевірте підключення до Інтернету.", "OK");
        ApplyPhoneEntryStage();
    }
    catch (AggregateException ae) when (ae.InnerException is PhoneNumberInvalid)
    {
        await DisplayAlert("Помилка", "Неправильний номер", "OK");
        ApplyPhoneEntryStage();
    }
    catch (AggregateException ae) when (ae.InnerException is ApiIdInvalid)
    {
        await DisplayAlert("Помилка", "Неправильний APP ID. Змініть його в
налаштуваннях клієнта.", "OK");
        ApplyPhoneEntryStage();
    }
}
```

При переході на головну сторінку (MainPage) відразу після відпрацювання її конструктора на виконання запускається метод OnStart():

```
private async void OnStart()
{
    telegramClientProvider = DependencyService.Get<ITelegramClientProvider>();
    var client = telegramClientProvider.TelegramClient;
    var synContext = await Device.GetMainThreadSynchronizationContextAsync();
    var backuper = new Backuper(client, synContext);
    telegramClientProvider.SetBackuper(backuper);
    try
    {
        ISettingsProvider settings = DependencyService.Get<ISettingsProvider>();
        int mainPeerId = settings.GetBackupPeerId();
        int thumbPeerId = settings.GetThumbnailPeerId();
        int dbPeerId = settings.GetDatabaseBackupPeerId();
        if (telegramClientProvider.LoggedUser == null)
```

```
        {
            var me = await client.GetMeAsync();
            telegramClientProvider.SetLoggedUser(me);
        }
        byte[] profilePicture = await
client.GetUserPhotoAsync(telegramClientProvider.LoggedUser, false);
        this.profilePicture = ImageSource.FromStream(() => new
MemoryStream(profilePicture));
        profilePic.Source = this.profilePicture;
        if (telegramClientProvider.Backuper != null)
        {
            await telegramClientProvider.Backuper.InitPeersAsync(mainPeerId,
thumbPeerId, dbPeerId);
            if (mainPeerId == 0 || thumbPeerId == 0 || dbPeerId == 0)
            {
                await DisplayAlert("Необхідне налаштування", "Налаштувань
резервного копіювання не було вияо. Створіть їх зараз.", "OK");
                await Navigation.PushModalAsync(new BackupPeerSelectionPage());
            }
        }
        else
        {
            logger.LogWarning($"[{DEBUG_TAG}] Backuper was null.");
        }
    }
    catch (Exception ex)
    {
        ex = ex is AggregateException ae ? ae.InnerException : ex;
        if (ex is UserUnauthorizedException unauthorizedEx)
        {
            logger.LogError($"[{DEBUG_TAG}] User unauthorized. Opening login
page...");
            await Navigation.PushModalAsync(new LoginPage());
            return;
        }
        logger.LogError($"[{DEBUG_TAG}] Error while loading app\n" +
            $"\tException was: {ex.Message}\n" +
            $"\tStack trace: {ex.StackTrace}");
        bool result = await DisplayAlert("Помилка", "Помилка при завантаженні
дотатку. Перегляньте лог щоб отримати більше інформації.", "Налаштування",
"Вийти");
        if (result)
        {
            await Navigation.PushModalAsync(new AdvancedSettingsPage());
        }
        else
        {
            Environment.Exit(-1);
        }
    }
}
```

Розберемо все по порядку. Додаток використовує ін’єкцію залежностей для отримання доступу до екземплярів сервісів та специфічних для платформи реалізацій інтерфейсів або абстрактних класів. Для реєстрації таких об’єктів та отримання доступу до них використовується статичний клас DependencyService, який надається фреймворком Xamarin. Всього в даному додатку було визначено

7 таких сервісів:

– ILocalMediaService (надає API для взаємодії з локальними медіа);

– IBackupedMediaService (надає API для взаємодії із вивантаженими медіа-файлами);

– IAutobackupService (дозволяє керувати роботою сервісу для автоматичного резервного копіювання);

– IPathService (надає специфічні для даної платформи шляхи);

– ITelegramClientProvider (надає доступ до екземплярів TgClient і Backuper та дозволяє відслідковувати їхню зміну через відповідні події);

– ISettingsProvider (надає API для читання та збереження усіх налаштувань додатка);

– AbsStatusBarColorChanger (дозволяє змінювати колір панелі статусу).

Деякі з цих сервісів реалізовано в головному проекті, але реалізація більшості з них (4 з 7) залежить від того, на якій платформі запущено додаток. Отже, все що потрібно для організації підтримки нової платформи це створення для неї свого власного екрану завантаження та імплементація для даної платформи 4 інтерфейсів, а саме: ILocalMediaService, IAutobackupService, IPathService та AbsStatusBarColorChanger.

Після отримання екземпляру класу, який імплементує інтерфейс ITelegramClientProvider, ми отримуємо з нього екземпляр класу TgClient і захоплюємо контекст UI потоку (щоб потім можна було за допомогою нього оновлювати елементи GUI із будь-якого іншого потоку). Ці два параметри використовуються для конструювання екземпляру Backuper. Після цього ми отримуємо інформацію про поточного користувача та зберігаємо її разом із екземпляром класу Backuper у відповідні поля, які надаються інтерфейсом ITelegramClientProvider. Наступним кроком стає завантаження та відображення фотографії поточного користувача. Отримавши із налаштувань ідентифікатори чатів для резервного копіювання, ми намагаємось отримати їхні хеші доступу, викликавши метод InitPeersAsync(). Якщо в налаштуваннях відсутні ці ідентифікатори, то користувача буде перенаправлено на сторінку вибору чатів

для резервного копіювання.

MainPage містить в собі лише загальний макет сторінки: верхній ряд (із заголовком та кнопками для сортування і переходу до налаштувань) та основна частина (елемент TabHostView). Модель даного представлення (клас MainPageVmo) зберігає посилання на екземпляри ще двох сторінок (LocalItemsView та BackupedItemsView), які й будуть виступати контентом для елементу TabHostView (перемикатися між цими двома сторінками можна буде через меню внизу сторінки MainPage).

LocalItemsView дозволяє переглядати локальні медіа-файли. Її модель представлення (клас LocalItemsVmo) надає API для отримання списку локальних медіа, їх групування та обробки сенсорних подій. Нижче представлено код методу для завантаження списку локальних медіа:

```
async void LoadLocalMedias()
{
    var stopWatch = new Stopwatch();
    logger?.LogInformation($"[{DEBUG_TAG}] Starting loading local files.");
    stopWatch.Start();

    await localMediaService.LoadLocalItems(App.AppMainCancellationToken);
    GroupItems();

    stopWatch.Stop();
    var ts = stopWatch.Elapsed;
    string elapsedTime = String.Format("{0:00}:{1:00}.{2:00}",
        ts.Minutes, ts.Seconds,
        ts.Milliseconds / 10);

    logger?.LogInformation($"[{DEBUG_TAG}] Local files loading completed in
{elapsedTime}");

    if (localMediaService.LocalMediaVmos.Count == 0)
    {
        IsEmptyViewVisible = true;
        EmptyViewText = "Медіа-файлів не знайдено";
    }
    else
    {
        IsEmptyViewVisible = false;
        IsMainLayoutVisible = true;
    }
}
```

Поле localMediaService представляє собою екземпляр класу що реалізує інтерфейс ILocalMediaService, отримується через DependencyService та

задається в конструкторі класу LocalItemsVmo. Після завершення завантаження отримані медіа-файли групуються та відображаються для користувача.

BackupedItemsView дозволяє переглядати вивантажені у Telegram медіа-файли. Для відображення елементів цієї сторінки використовуються ті ж самі об’єкти що і у попередній за тим виключенням що моделлю представлення для даної сторінки виступає клас BackupedItemsVmo, який отримує інформацію про вивантажені файли користуючись екземпляром класу що реалізує інтерфейс IBackupedMediaService. Повний код цієї та інших основних сторінок додатку показано у додатку В.

3.3 Реалізація розмітки мобільного додатка

Фреймворк Xamarin дозволяє створювати логіку додатків використовуючи при цьому мову C# замість Java чи Kotlin (у випадку розробки додатку для Android) або Swift (у випадку iOS). Тим не менше, процес створення інтерфейсів користувача у проекті для конкретної платформи нічим не відрізняється від процесу його створення в IDE створених спеціально для розробки під дану платформу. Тобто якщо говорити про Android, то для проектування інтерфейсу на цій платформі у Xamarin використовується точно така ж сама XML розмітка та точно такий самий набір нативних компонентів представлення, що й може запропонувати Android Studio.

Розмітка екрану завантаження додатка (файл splash_layout.xml із проекту Cloudgram.Android) виглядає наступним чином:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:background="@color/launcher_background"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <LinearLayout
        android:orientation="vertical"
        android:gravity="center"
```

```
        android:layout_weight="1"
        android:layout_width="match_parent"
        android:layout_height="match_parent">
        <ImageView
            android:layout_width="match_parent"
            android:layout_height="200dp"
            android:layout_gravity="center"
            android:src="@drawable/logo_new"/>
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginTop="25dp"
            android:textSize="42sp"
            android:textColor="@color/black"
            android:textAlignment="center"
            android:text="Cloudgram" />
        <TextView
            android:id="@+id/status"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginTop="5dp"
            android:layout_marginStart="15dp"
            android:layout_marginEnd="15dp"
            android:layout_gravity="center_horizontal"
            android:textSize="16sp"
            android:textColor="@color/defaultColor"
            android:textAlignment="gravity"/>
        <Button
            android:id="@+id/retryBtn"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginTop="30dp"
            android:text="Спробувати ще раз"
            android:textColor="@android:color/white"
            android:background="@color/defaultColor"
            android:paddingStart="10dp"
            android:paddingEnd="10dp"
            android:visibility="gone" />
        <Button
            android:id="@+id/settingsBtn"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginTop="10dp"
            android:text="Налаштування"
            android:textColor="@color/defaultColor"
            android:background="@android:color/transparent"
            android:shadowColor="@android:color/transparent"
            android:paddingStart="10dp"
            android:paddingEnd="10dp"
            android:visibility="gone" />
    </LinearLayout>
    <TextView
        android:id="@+id/version"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="12sp"
        android:textColor="@color/defaultColorLight"
        android:textAlignment="center"
        android:layout_margin="15dp"
        android:gravity="center_horizontal" />

</LinearLayout>
```

Для відображення елементів представлення використовується контейнер LinearLayout з вертикальною орієнтацією. Це означає що його дочірні елементи заповнюватимуть простір батьківського зверху до низу. Першим дочірнім елементом виступає логотип додатку, представлений елементом ImageView. Далі послідовно розміщено два TextView: один із назвою додатку, а інший використовуватиметься для відображення прогресу виконання поточної операції. Далі знаходиться дві спочатку невидимі кнопки: кнопка повторного підключення (починає відображатись лише якщо додатку не вдалося підключитися до Telegram) та кнопка переходу до тонких налаштувань клієнта (відображається після невдалої спроби підключення або у випадку якщо спроба підключення триває довше 5 секунд). В самому низу сторінки знаходиться TextView для відображення версії додатка.

Кросплатформні представлення із головного проекту для розмітки використовують мову XAML та свої власні компоненти, які в процесі компіляції під конкретну платформу будуть перетворені на нативні для даної платформи компоненти (для забезпечення максимальної продуктивності). Розмітка сторінки MainPage із головної бібліотеки показана нижче.

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:tabs="http://sharpnado.com"
             xmlns:views="clr-namespace:Cloudgram.Views"
             x:Class="Cloudgram.MainPage">
    <ContentPage.Resources>
        <ResourceDictionary>
            <Style x:Key="BottomTabStyle" TargetType="tabs:BottomTabItem">
                <Setter Property="SelectedTabColor" Value="White" />
                <Setter Property="UnselectedIconColor" Value="LightGray" />
                <Setter Property="UnselectedLabelColor" Value="LightGray" />
                <Setter Property="LabelSize" Value="14" />
                <Setter Property="IconSize" Value="28" />
                <Setter Property="IsTextVisible" Value="True" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>
    <StackLayout BackgroundColor="White" VerticalOptions="FillAndExpand"
HorizontalOptions="FillAndExpand">
        <StackLayout VerticalOptions="FillAndExpand">
            <views:StatusView x:Name="statusView"
HorizontalOptions="FillAndExpand" IsVisible="False" />
            <StackLayout Padding="15, 15, 20, 0" Orientation="Horizontal">
```

```
                <Label x:Name="titleLabel" Text="Локальні фото" Margin="0, 5, 5,
5" FontSize="Large" TextColor="Black" VerticalOptions="Center"
HorizontalOptions="StartAndExpand">
                    <Label.GestureRecognizers>
                        <TapGestureRecognizer Tapped="GoToGroup_Tapped" />
                    </Label.GestureRecognizers>
                </Label>
                <ImageButton Source="{Binding SortPath}" Margin="5, 0, 10, 0"
VerticalOptions="Center" BackgroundColor="Transparent" HeightRequest="23"
WidthRequest="23" HorizontalOptions="End" Aspect="AspectFit"
Clicked="SortButton_Clicked" />
                <Frame Padding="0" BorderColor="Black"
                                   CornerRadius="20"
                                   HeightRequest="40"
                                   WidthRequest="40">
                    <ImageButton x:Name="profilePic"
                                         Aspect="AspectFill"
                                         BackgroundColor="LightGray"
                                         HorizontalOptions="FillAndExpand"
                                         VerticalOptions="FillAndExpand"
                                         Clicked="ProfilePicButton_Clicked" />
                </Frame>
            </StackLayout>
            <tabs:ViewSwitcher x:Name="Switcher"
                               Animate="True"
                               VerticalOptions="FillAndExpand"
HorizontalOptions="FillAndExpand"
                               SelectedIndex="{Binding SelectedViewModelIndex,
Mode=TwoWay}">
                <views:LocalItemsView x:Name="localItemsView" Animate="True"
BindingContext="{Binding LocalItems}" />
                <views:BackupedItemsView x:Name="backupedItemsView"
Animate="True" BindingContext="{Binding BackupedItems}" />
            </tabs:ViewSwitcher>
        </StackLayout>
        <tabs:TabHostView x:Name="TabHost"
                          BackgroundColor="#1E90FF"
                          SelectedIndex="{Binding Source={x:Reference Switcher},
Path=SelectedIndex, Mode=TwoWay}">
            <tabs:TabHostView.Tabs>
                <tabs:BottomTabItem Style="{StaticResource BottomTabStyle}"
                                    IconImageSource="gallery.png"
                                    Label="Локальні" />
                <tabs:TabButton
                        x:Name="searchButton"
                        ButtonBackgroundColor="White"
                        ButtonCircleSize="60"
                        ButtonPadding="18"
                        IconImageSource="search.png"
                        Scale="1.3"
                        TranslationY="-14" />
                <tabs:BottomTabItem Style="{StaticResource BottomTabStyle}"
                                    IconImageSource="backup.png"
                                    Label="Вивантажені" />
            </tabs:TabHostView.Tabs>
        </tabs:TabHostView>
    </StackLayout>
</ContentPage>
```

Спершу ми визначаємо свій власний стиль у спеціальному словнику

ресурсів (даний стиль буде доступний в межах цієї сторінки) та вказуємо що він може бути застосований лише до елементів типу BottomTabItem. Потім ми визначаємо контейнер StackLayout, який є батьківським для решти елементів на сторінці. Даний елемент фактично робить те ж саме що й елемент LinearLayout в Android – дозволяє розміщувати свої дочірні елементи один за одним та в стовпчик. Першим дочірнім елементом виступає StatusView. Даний компонент створюється як невидимий для користувача, однак він починає відображатися у випадку втрати з'єднання з Інтернетом. Після цього створюється ще один елемент StackLayout, але вже із горизонтальною орієнтацією елементів (тобто його дочірні елементи будуть розміщуватись в рядок). В ньому знаходиться Label (аналог TextView) для заголовка, кнопка для зміни групування елементів та кнопка переходу до налаштувань. В елементі ViewSwitcher визначаються представлення, які будуть відображатися при натисканні на вкладки, визначені в елементі TabHostView.

Розмітку представлення LocalItemsView, яке відповідає за відображення локальних медіа-файлів, показано нижче.

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentView xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
xmlns:controls="clr-namespace:DLToolkit.Forms.Controls;
assembly=DLToolkit.Forms.Controls.FlowListView"
             x:Class="Cloudgram.Views.LocalItemsView">
    <ContentView.Content>
      <StackLayout HorizontalOptions="FillAndExpand"
VerticalOptions="FillAndExpand">
            <Label x:Name="emptyViewLabel" Text="{Binding EmptyViewText}"
FontSize="Small"  HorizontalTextAlignment="Center"
VerticalTextAlignment="Center"
                   TextColor="Black" VerticalOptions="FillAndExpand"
HorizontalOptions="FillAndExpand" IsVisible="{Binding IsEmptyViewVisible}" />
            <controls:FlowListView FlowColumnCount="{Binding ColumnsCount}"
                        x:Name="mainLayout"
                        IsVisible="{Binding IsMainLayoutVisible}"
                        Header=""
                        HasUnevenRows="True"
                        FlowUseAbsoluteLayoutInternally="True"
                        IsGroupingEnabled="True"
                        SeparatorVisibility="None"
                        SelectionMode="Single"
                        FlowItemsSource="{Binding LocalMedias}"
                        FlowLastTappedItem="{Binding MediaSelected}"
                        FlowItemTappedCommand="{Binding ItemTappedCommand}">
                <x:Arguments>
```

```
<ListViewCachingStrategy>RecycleElement</ListViewCachingStrategy>
                </x:Arguments>

                <controls:FlowListView.GroupHeaderTemplate>
                    <DataTemplate>
                        <ViewCell>
                            <StackLayout Margin="15, 15, 5, 15"
Orientation="Horizontal">
                                <Image Source="{Binding Path=Key.Icon}"
VerticalOptions="Center" HeightRequest="20" WidthRequest="20" Aspect="AspectFit"
/>
                                <Label Text="{Binding Path=Key.Name}" Margin="5,
0, 0, 0" FontSize="Small" TextColor="Black"
                                   VerticalOptions="Center" />
                                <Label Text="{Binding Path=Key.CountString}"
Margin="5, 0, 0, 0" FontSize="Small" TextColor="Gray"
                                   VerticalOptions="Center"
HorizontalOptions="FillAndExpand" />
                            </StackLayout>
                        </ViewCell>
                    </DataTemplate>
                </controls:FlowListView.GroupHeaderTemplate>

                <controls:FlowListView.FlowColumnTemplate>
                    <DataTemplate>
                        <DataTemplate.Bindings>

                        </DataTemplate.Bindings>
                        <Grid x:Name="cell" Margin="1">
                            <Grid.RowDefinitions>
                                <RowDefinition Height="{StaticResource
CellHeightRequest}"></RowDefinition>
                            </Grid.RowDefinitions>
                            <Image Source="{Binding ThumbnailPath}"
                                    HorizontalOptions="FillAndExpand"
                                    VerticalOptions="FillAndExpand"
                                    Aspect="AspectFill" >
                                <Image.GestureRecognizers>
                                    <TapGestureRecognizer Command="{Binding
Source={x:Reference mainLayout}, Path=BindingContext.ItemTappedCommand}"
CommandParameter="{Binding .}"/>
                                </Image.GestureRecognizers>
                            </Image>
                        </Grid>
                    </DataTemplate>

                </controls:FlowListView.FlowColumnTemplate>
            </controls:FlowListView>
        </StackLayout>
    </ContentView.Content>
</ContentView>
```

Для відображення медіа-файлів у вигляді галереї об’єктів було використано компонент FlowListView, всередині якого було визначено яким чином повинен виглядати заголовок групи елементів та загальний вигляд для кожного окремого елементу. Окрім того, за допомогою механізму прив’язки

даних було визначено звідки саме брати елементи та які саме їх властивості необхідно відображати.

3.4 Розробка бази даних

Як вже було визначено в попередньому розділі, для збереження даних буде використовуватись база даних SQLite, а Entity Framework Core використовуватиметься в якості об’єктно-орієнтованої технології доступу до даних. Тобто нам не потрібно буде писати запити SQL, оскільки ми не працюватимемо з таблицями бази даних напряму. Ми працюватимемо із сутностями (звичайні класи моделі) та використовуватимемо LINQ для створення вибірок з елементів що зберігаються.

Перш за все, необхідно створити класи моделі. Кожен такий клас в результаті представлятиме окрему таблицю в базі даних. Код класу BackupedMedia показано нижче.

```
public class BackupedMedia
{
    public const byte DECRYPTION_ERROR_CODE = 101;

    public int BackupedMediaId { get; set; }
    public string FilePath { get; set; }
    public string ThumbnailPath { get; set; }
    public string MimeType { get; set; }
    public bool IsVideo { get; set; }
    public string Album { get; set; }
    public DateTime CreatedAt { get; set; }
    public int Size { get; set; }

    public bool IsEncrypted { get; set; }
    public string PasswordHash { get; set; }

    public ICollection<FileLocation> FileLocations { get; set; }
    public ICollection<BackupedMediaTag> BackupedMediaTags { get; set; }

    public BackupedMedia()
    {
        FileLocations = new List<FileLocation>();
        BackupedMediaTags = new List<BackupedMediaTag>();
    }

    public override int GetHashCode()
    {
```

```
        return BackupedMediaId ^ 7;
    }

    public override bool Equals(object obj)
    {
        if(obj is BackupedMedia backupedMedia)
        {
            return backupedMedia.BackupedMediaId == BackupedMediaId;
        }

        return false;
    }
}
```

Даний клас містить ті ж самі поля, що й однойменна сутність, яка була показана на ER діаграмі у попередньому розділі. Окрім того, даний клас також перевизначає методи Equals() та GetHashCode(), а тому екземпляри даного класу тепер порівнюватимуться не за посиланням (поведінка за замовчуванням для екземплярів типів посилань), а за їх ідентифікатором.

Далі показано код класу FileLocation, який містить інформацію про контекст, в якому зберігається вивантажений в Telegram файл.

```
public class FileLocation
{
    public int FileLocationId { get; set; }

    public int ChatId { get; set; }
    public int MessageId { get; set; }

    public int Size { get; set; }

    public int ThumbnailLocationId { get; set; }
    public ThumbnailLocation ThumbnailLocation { get; set; }

    public int BackupedMediaId { get; set; }
    public BackupedMedia BackupedMedia { get; set; }


    public static implicit operator FileContext(FileLocation fileLocation)
    {
        FileContext fileContext = new FileContext
        {
            MessageContext = new MessageContext
            {
                ChatId = fileLocation.ChatId,
                MessageId = fileLocation.MessageId
            },
            Size = fileLocation.Size,
            Thumbnail = fileLocation.ThumbnailLocation
        };
        return fileContext;
    }
```

```
    public static implicit operator MessageContext(FileLocation fileLocation)
    {
        MessageContext messageContext = new MessageContext
        {
            ChatId = fileLocation.ChatId,
            MessageId = fileLocation.MessageId
        };
        return messageContext;
    }
}
```

Окрім властивостей, які стануть стовпцями однойменної таблиці, даний клас перевизначає оператори приведення типів таким чином, щоби тип FileLocation міг бути неявно приведений до типу FileContext або MessageContext (типи з бібліотеки Cloudgram.Base). Схожим чином визначається також сутність ThumbnailLocation, яка міститиме інформацію про контекст в якому зберігається іконка вивантаженого файлу.

Нижче показано код класу Tag, який містить інформацію про тег та перевизначає відразу три методи: Equals(), GetHashCode() та ToString().

```
public class Tag
{
    public int TagId { get; set; }
    public string TagName { get; set; }
    public ICollection<BackupedMediaTag> BackupedMediaTags { get; set; }

    public Tag()
    {
        BackupedMediaTags = new List<BackupedMediaTag>();
    }

    public override int GetHashCode()
    {
        return TagId ^ 7;
    }

    public override bool Equals(object obj)
    {
        if(obj is Tag tag)
        {
            return tag.TagId == TagId;
        }
        return false;
    }

    public override string ToString()
    {
        return TagName;
    }
}
```

Клас BackupedMediaTag є проміжною сутністю, яка допомагає налаштувати зв’язок багато-до-багатьох між сутністю Tag та BackupedMedia.

```
public class BackupedMediaTag
{
    public int BackupedMediaId { get; set; }
    public BackupedMedia BackupedMedia { get; set; }
    public int TagId { get; set; }
    public Tag Tag { get; set; }

    public override int GetHashCode()
    {
        return BackupedMediaId ^ 7 + TagId ^ 7;
    }

    public override bool Equals(object obj)
    {
        if(obj is BackupedMediaTag bmt)
        {
            return bmt.BackupedMediaId == BackupedMediaId && bmt.TagId == TagId;
        }
        return false;
    }
}
```

Після того як всі сутності готові необхідно створити контекст даних який і представлятиме найнижчий рівень доступу до інформації з БД:

```
public class ApplicationContext : DbContext
{
    private string _databasePath;

    public DbSet<BackupedMedia> BackupedMedias { get; set; }
    public DbSet<FileLocation> FileLocations { get; set; }
    public DbSet<ThumbnailLocation> ThumbnailLocations { get; set; }
    public DbSet<Tag> Tags { get; set; }
    public DbSet<BackupedMediaTag> BackupedMediaTags { get; set; }

    public ApplicationContext(string databasePath)
    {
        _databasePath = databasePath;
    }

    protected override void OnConfiguring(DbContextOptionsBuilder
        optionsBuilder)
    {
        optionsBuilder.UseSqlite($"Filename={_databasePath}");
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Tag>()
            .HasIndex(t => t.TagName)
            .IsUnique();
```

```
        modelBuilder.Entity<FileLocation>()
            .HasOne(fl => fl.BackupedMedia)
            .WithMany(bm => bm.FileLocations)
            .HasForeignKey(fl => fl.BackupedMediaId);

        modelBuilder.Entity<FileLocation>()
            .HasOne(fl => fl.ThumbnailLocation)
            .WithMany(tl => tl.FileLocations)
            .HasForeignKey(fl => fl.ThumbnailLocationId);

        modelBuilder.Entity<BackupedMediaTag>()
            .HasKey(bmt => new { bmt.BackupedMediaId, bmt.TagId });
        modelBuilder.Entity<BackupedMediaTag>()
            .HasOne(bmt => bmt.BackupedMedia)
            .WithMany(bm => bm.BackupedMediaTags)
            .HasForeignKey(bmt => bmt.BackupedMediaId);
        modelBuilder.Entity<BackupedMediaTag>()
            .HasOne(bmt => bmt.Tag)
            .WithMany(t => t.BackupedMediaTags)
            .HasForeignKey(bmt => bmt.TagId);
    }
}
```

В перевизначеному методі OnModelCreating всі зв’язки між сутностями налаштовуються вручну за допомогою Fluent API. Клас ApplicationContext не використовується напряму. Вся взаємодія із БД відбувається через інший клас, який повинен імплементувати інтерфейс IBackupedMediaService. Оскільки в даному класі не повинно бути ніякої залежної від платформи логіки, він має бути визначений у головному проекті. Фрагмент коду класу BackupedMediaService (який реалізує вищевказаний інтерфейс) для додавання нового запису в БД показано нижче.

```
public async Task<BackupedMediaVmo> AddMediaAsync(LocalMedia localMedia,
    IEnumerable<FileContext> fileContexts, string passwordHash = null)
{
    BackupedMediaVmo backupedMediaVmo = null;
    await _semaphoreSlim.WaitAsync();
    try
    {
        using (ApplicationContext db = new ApplicationContext(_dbPath))
        {
            // Adding thumbnail
            FileContext fileContext = fileContexts.ElementAt(0);
            ThumbnailLocation thumbnailLocation = new ThumbnailLocation
            {
                ChatId = fileContext.Thumbnail.MessageContext.ChatId,
                MessageId = fileContext.Thumbnail.MessageContext.MessageId,
                Size = fileContext.Thumbnail.Size
            };
            await db.ThumbnailLocations.AddAsync(thumbnailLocation);
```

```
            // Adding file contexts
            List<FileLocation> fileLocations = new List<FileLocation>();
            foreach (var fc in fileContexts)
            {
                FileLocation fl = new FileLocation
                {
                    ChatId = fc.MessageContext.ChatId,
                    MessageId = fc.MessageContext.MessageId,
                    Size = fc.Size,
                    ThumbnailLocation = thumbnailLocation
                };
                fileLocations.Add(fl);
            }
            await db.FileLocations.AddRangeAsync(fileLocations);
            // Adding backuped media
            BackupedMedia backupedMedia = new BackupedMedia
            {
                FilePath = localMedia.Path,
                MimeType = localMedia.MimeType,
                IsVideo = localMedia.IsVideo,
                Size = localMedia.Size,
                Album = localMedia.Album,
                ThumbnailPath = localMedia.ThumbPath,
                CreatedAt = localMedia.CreatedAt,
                IsEncrypted = passwordHash != null,
                PasswordHash = passwordHash,
                FileLocations = fileLocations
            };
            await db.BackupedMedias.AddAsync(backupedMedia);
            await db.SaveChangesAsync();
            backupedMediaVmo = new BackupedMediaVmo(backupedMedia);
        }
    }
    finally
    {
        _semaphoreSlim.Release();
    }
    BackupedMediaVmos.Add(backupedMediaVmo);
    BackupStatistics?.AddBytes(backupedMediaVmo.BackupedMedia.Size);
    OnMediaBackuped?.Invoke(this, new
BackupedMediaVmoUpdatedEventArgs(backupedMediaVmo));
    return backupedMediaVmo;
}
```

3.5 Керівництво користувача

Cloudgram – це безкоштовний додаток з відкритим кодом (розповсюджується під ліцензією MIT), який не містить реклами та на даний момент доступний для завантаження з платформи Google Play Market. Цей додаток є неофіційним клієнтом Telegram та дозволяє виконувати автоматичне резервне копіювання медіа-файлів у вибраний користувачем чат. Це також означає що користуватися функціоналом Cloudgram зможуть лише ті

користувачі, які зареєстровані у Telegram та виконали вхід у свій акаунт з цього додатку. Варто відзначити, що я не зберігаю та не збираю жодної інформації про користувачів цього додатку та дані, з якими вони працюють. Вся необхідна для роботи додатку інформація зберігається або на пристрої користувача (в зашифрованому вигляді) або на серверах Telegram (і більше ніде, якщо тільки сам користувач не експортує локальну базу даних на якийсь інший пристрій).

Після входу в свій Telegram акаунт користувачу буде запропоновано вибрати чати в які буде виконуватись резервне копіювання медіа-файлів, їх іконок та локальної бази даних. Після цього додатком можна буде користуватися так само як і звичайною галереєю, за тим виключенням що тепер будь-який медіа-файл можна буде вивантажити у хмару Telegram прямо з його контекстного меню. Щоб не робити це кожного разу вручну, користувачу надається можливість налаштувати автоматичне резервне копіювання медіа-файлів із вибраних альбомів. Для цього на головній сторінці додатка слід натиснути на кнопку із зображенням профілю користувача та на сторінці що з'явиться вибрати пункт «Автобекап», де його можна буде увімкнути та вибрати необхідні альбоми.

Окрім того, додаток також дозволяє користувачам структурувати свою галерею шляхом додавання тегів до медіа-файлів. Для цього потрібно відкрити зображення для перегляду у повноекранному режимі та натиснути там на кнопку «Додати теги». Додаток також дозволяє виконувати пошук медіа-файлів за тегами. Для переходу на сторінку пошуку достатньо лише натиснути на кнопку із зображенням лупи у нижньому меню.

3.6 Технічні характеристики мобільного додатка

Для успішного запуску та стабільної роботи пристрій користувача повинен мати підключення до мережі Інтернет а також задовольняти наступні мінімальні системні вимоги:

– ОС Android 5.0 або новіша;

– будь-який двоядерний процесор;

– 500 МБ ОЗП;

– 500 МБ постійної пам’яті.

Рекомендовані вимоги:

– ОС Android 6;

– будь-який чотирьохядерний процесор;

– 1 ГБ ОЗП;

– 500 МБ постійної пам’яті.

Отже, в процесі написання даного розділу було реалізовано та здійснено опис компонентів бібліотеки Cloudgram.Base та мобільного додатка Cloudgram. Розроблений в результаті додаток задовольняє усі вимоги, які були поставлені до системи раніше. Окрім того, було написане керівництво користувача та складені вимоги до технічних характеристик пристрою, які дозволили б комфортно користуватися цим додатком. Повний код програмної системи наведено у додатку В.

4 ТЕСТУВАННЯ МОБІЛЬНОГО ДОДАТКА

4.1 Аналіз методів тестування мобільного додатка

На сьогоднішній день ринок не відчуває дефіциту смартфонів. Не дивлячись на те, що мобільний є у кожної дорослої людини (та й навіть у дітей), і навіть на той факт, що через пандемію обсяги поставок нових смартфонів дещо скоротилися, в магазинах все одно вистачає найрізноманітніших моделей смартфонів на будь-який смак.

Це ускладнює процес тестування мобільних додатків, оскільки окрім перевірки відповідності характеристик розробленого додатку характеристикам, які були поставлені в процесі формування технічного завдання, необхідно також приділяти чимало уваги розміру екрану, щільності пікселів (що впливатиме на те, як відображатиметься користувацький інтерфейс) та конфігурації пристрою (яких на даний момент існує величезна кількість).

Тестування, під час якого визначається чи задовольняє розроблений додаток вимогам, які були поставлені в ТЗ, називається функціональним. Існує декілька видів тестування, кожне з яких дозволяє перевірити якийсь конкретний аспект роботи як окремих модулів так і системи в цілому.

Модульне тестування (або, як його ще називають, компонентне) чудово підходить для тестування кожної окремо взятої функції додатку у контрольованих (як правило створених штучно) умовах. Це досягається шляхом заміни використовуваних у компоненті залежностей на спеціальні заглушки, які симулюють обмін даними із модулем, що тестується. Ми використовуватимемо модульне тестування для перевірки роботи бібліотеки Cloudgram.Base. Окрім того, модульне тестування буде використано для перевірки окремих компонентів самого додатка. У другому розділі ми обрали шаблон MVVM в тому числі й через легкість з яким відбувається тестування додатків, побудованих за цією архітектурою. Але й сам фреймворк Xamarin.Forms, разом із його правилами та рекомендаціями, допомагає писати легкий для подальшого

тестування код. Окремі проекти для кожної із підтримуваних платформ спрощують процес модульного тестування для залежного від платформи коду, а передбачений фреймворком механізм інтеграції подібних компонентів у спільну кодову базу (так званий Shared Project, де знаходиться уся бізнес-логіка додатка та незалежний від платформи код) спрощує процес проведення інтеграційного.

Окрім того, Xamarin.Forms надає спеціальний фреймворк для тестування кросплатформних користувацьких інтерфейсів під назвою Xamarin.UITest. Він використовує NUnit, надає API для взаємодії з UI через код на C# та дозволяє перевіряти на коректність реакцію UI на дії користувача. Оскільки даний проект все одно матиме залежність від NUnit, то даний фреймворк також використовуватиметься і для проведення модульного тестування решти компонентів додатка.

4.2 Тестування додатка

Почнемо тестування модулів додатка із розробки модульних тестів для бібліотеки Cloudgram.Base. Тестові сценарії показано у таблиці 4.1.

Таблиця 4.1 – Тестові сценарії бібліотеки Cloudgram.Base

Назва сценарію	Модуль	Метод	Вхідні дані	Очікуваний результат
1	2	3	4	5
Отримання інформації про користувача	TgClient	GetMeAsync	Пріоритет запиту	Екземпляр класу TLUserFull
Отримання списку діалогів	TgClient	GetUser DialogsAsync	Максимальна кількість діалогів, які можуть бути отримані за один раз та пріоритет запиту	Екземпляр класу TLDialogsSlice
Отримання зображення профілю користувача	TgClient	GetUser PhotoAsync	Екземпляр TLUserFull та пріоритет запиту	Масив байтів, які представляють зображення (або null, якщо воно відсутнє)
Отримання зображення каналу	TgClient	GetChannel PhotoAsync	Екземпляр TLChannel та пріоритет запиту	Масив байтів, які представляють зображення (або null)

Кінець таблиці 4.1

1	2	3	4	5
Вставка запиту в чергу	TgRequest Executor	CreatePriority Request	Тіло запиту та пріоритет	Вставка запиту в чергу та повернення екземпляру фейкового завдання
Вивантаження файлу в чат	TgUploader	UploadAsync	Екземпляр FileData з інформацією про вивантажуваний файл, метод зворотнього виклику, пріоритет запиту та токен для відміни	Вміст файлу буде зчитано, вивантажено в Telegram та як результат отримано перечислення екземплярів FileContext
Завантаження файлу	TgDownloader	DownloadAsync	Перечислення екземплярів типу FileContext, метод зворотнього виклику, пріоритет запиту та токен для відміни	Контекст буде використано для поетапного завантаження файлу та його збереження в масив байтів, який і буде результатом виконання методу
Отримання інформації про чат	Backuper	ResolvePeer ByIdAsync	Ідентифікатор чату та пріоритет запиту	Екземпляр класу Peer із інформацію про чат та його хешом доступу
Створення нового чату	Backuper	CreateBackup ChatAsync	Назва чату та його опис	Екземпляр класу Peer із повною інформацією про створений чат
Отримання списку чатів для бекапу	Backuper	GetPossible BackupPeers	Пріоритет запиту	Перечислення екземплярів класу Peer
Отримання посилання на файл	Backuper	ResolveFile Reference	Контекст повідомлення (екземпляр MessageContext) і пріоритет запиту	Екземпляр класу FileContext (або null у випадку якщо файл не знайдено)
Створення завдання вивантаження	Backuper	CreateUpload Task	Екземпляр FileData та пріоритет запиту	Буде створено нове завдання на вивантаження файлу, яке буде повернене викликаючому коду
Створення завдання завантаження	Backuper	CreateDownload Task	Перечислення із екземплярами FileContext та пріоритет запиту	Буде створено нове завдання на завантаження файлу, яке буде повернене викликаючому коду
Видалення файлу	Backuper	DeleteAsync	Екземпляр MessageContext та пріоритет запиту	Повідомлення буде видалено з Telegram
Завантаження іконки	Backuper	Download ThumbnailAsync	Екземпляр ThumbnailContext та пріоритет запиту	Іконка файлу, представлена масивом байтів

Для функціонального тестування додатку використовувався реальний пристрій Poco X3 (під керуванням ОС Android 10), а також два емулятори

Google Pixel 2 (із Android 5.1 і 7.1) та емулятор Google Pixel 3 (Android 11). Тестування деяких елементів графічного інтерфейсу додатку було автоматизовано шляхом використання фреймворку Xamarin.UITest. Для цього кожному елементу UI, з яким буде відбуватися взаємодія в ході тестування, слід призначити його власний AutomationId. Потім його можна використовувати в ході написання тестів. Наприклад, код методу для тестування відображення помилки про занадто короткий тег показано далі:

```
[Test]
public void TagTooShort_ShowError()
{
    string tooShortTagText = "t";
    // очікуємо на відображення поля введення тегу
    app.WaitForElement(c => c.Marked("TagEntryField"));
    // заповнюємо поле
    app.EnterText(c => c.Marked("TagEntryField"), tooShortTagText);
    // можна зберігати скріншоти контрольних точок для їх подальшого аналізу
    app.Screenshot("Entering too short tag");
    // програмно натискаємо на кнопку додавання тега
    app.Tap(c => c.Marked("AddTagButton"));
    app.Screenshot("Add tag button click results");
    // перевіряємо чи з'явилося повідомлення про помилку
    AppResult[] result = app.Query(c => c.Class("Label").Text("Тег повинен
складатися мінімум із двох символів"));
    Assert.IsTrue(result.Any(), "Tag too short error message isn't being
displayed.");
}
```

Опис сценаріїв функціонального тестування додатка Cloudgram наведено у таблиці 4.2.

Таблиця 4.2 – Тестові сценарії основного функціоналу Cloudgram

Іденти-фікатор	Функціональна вимога	Вихідні дані	Очікуваний результат
1	2	3	4
CFT-01	Перегляд локальних медіа-файлів	Користувач відкриває головну сторінку додатка	Користувач бачить усі локальні медіа-файли
CFT-02	Перегляд медіа-файлу	Користувач торкається до мініатюри медіа-файлу	Відкривається екран перегляду медіа-файлу
CFT-03	Поділитися медіа-файлом	Користувач натискає кнопку «Поділитися» на екрані перегляду медіа-файлу	Відображається діалог відправки файлу через інші встановлені додатки
CFT-04	Вивантаження медіа-файлу	Користувач натискає кнопку «Вивантажити» на екрані перегляду медіа-файлу	Відобразити діалогове вікно з пропозицією вибору способу завантаження: із попереднім шифруванням, або без нього

Кінець таблиці 4.2

1	2	3	4
CFT-05	Завантаження медіа-файлу	Користувач натискає кнопку «Завантажити» на екрані перегляду медіа-файлу (кнопка доступна лише для вивантажених файлів які відстуні локально)	Файл завантажується та зберігається в альбом Cloudgram
CFT-06	Редагування тегів	1. Користувач натискає кнопку «Додати теги» на екрані перегляду медіа-файлу 2. Користувач редагує теги	1. Відкрити екран редагування тегів 2. Зміни вносяться у БД
CFT-07	Видалити медіа-файл	Користувач натискає кнопку «Видалити» на екрані перегляду медіа-файлу	Відобразити діалогове вікно в якому пропонується видалити медіа-файл з Telegram, локального пристрою або відразу з обох місць (доступні для вибору лише ті розміщення, в яких даний файл наразі існує)
CFT-08	Пошук за тегом	1. Користувач натискає на кнопку із зображенням лупи, яка знаходиться у нижній частині головного екрану 2. Користувач вводить тег для пошуку	1. Відкрити екран пошуку за тегами 2. Користувачу відображаються результати пошуку
CFT-09	Перейти до налаштувань	Користувач натискає на кнопку із зображенням свого профілю, яка знаходиться у правій верхній частині головного екрану	Відкрити екран налаштувань додатка
CFT-10	Налаштувати автоматичне резервне копіювання	1. На сторінці налаштувань додатка натиснути на кнопку «Автобекап» 2. Обрати альбоми для автоматичного резервного копіювання	1. Відкрити екран налаштувань автоматичного резервного копіювання 2. Зберегти в налаштуваннях додатка вибрані користувачем альбоми
CFT-11	Звільнити місце на пристрої	1. На сторінці налаштувань додатка натиснути на кнопку «Звільнити місце» 2. Підтвердити дію	1. Відкриється екран звільнення місця 2. Розпочати процедуру звільнення місця
CFT-12	Змінити чати для резервного копіювання	1. На сторінці налаштувань додатка натиснути на кнопку «Налаштувати чати для резервного копіювання» 2. Вибрати чати	1. Відкриється сторінка вибору та налаштування чатів для резервного копіювання 2. Зберегти в налаштуваннях додатка вибрані користувачем чати
CFT-13	Перейти до просунутих налаштувань	На сторінці налаштувань додатка натиснути на кнопку «Просунуті налаштування»	Відкриється сторінка із просунутими налаштуваннями
CFT-14	Вийти з акаунту	На сторінці налаштувань натиснути кнопку «Вихід»	Дані сесії буде видалено, а користувача буде перенаправлено на сторінку входу в Telegram

4.3 Аналіз результатів тестування мобільного додатка

Результати модульного тестування бібліотеки Cloudgram.Base наведено у таблиці 4.3.

Таблиця 4.3 – Результати модульного тестування Cloudgram.Base

Назва сценарію	Метод	Вхідні дані	Отриманий результат	Результат
1	2	3	4	5
Отримання інформації про користувача	GetMeAsync	Нормальний пріоритет запиту	Екземпляр класу TLUserFull, який представляє поточного користувача	Правильно
Отримання списку діалогів	GetUser DialogsAsync	5 (кількість діалогів, які можуть бути отримані за один раз) та нормальний пріоритет запиту	Екземпляр класу TLDialogsSlice (який містить інформацію про 5 чатів)	Правильно
Отримання зображення профілю користувача	GetUser PhotoAsync	Екземпляр TLUserFull отриманий із першого модульного тесту	Масив байтів, які представляють зображення профілю користувача	Правильно
Отримання зображення каналу	GetChannel PhotoAsync	Екземпляр TLChannel отриманий із тестування створення чату	Масив байтів, які представляють зображення чату	Правильно
Вставка запиту в чергу	CreatePriority Request	Запит на відправку повідомлення	Вставка запиту в чергу та повернення екземпляру Task<TLAbsUpdates>	Правильно
Вивантаження файлу в чат	UploadAsync	Екземпляр FileData з інформацією про вивантажуваний файл (решта параметрів не обов’язкові)	Вміст файлу було зчитано, вивантажено в Telegram та як результат отримано перечислення екземплярів FileContext	Правильно
Завантаження файлу	DownloadAsync	Перечислення екземплярів типу FileContext (решта параметрів не обов’язкові)	Контекст було використано для поетапного завантаження файлу та його збереження в масив байтів, який і став результатом виконання методу	Правильно
Отримання інформації про чат	ResolvePeer ByIdAsync	Ідентифікатор чату отриманий із тестування створення чату	Екземпляр класу Peer із інформацію про чат та його хешом доступу	Правильно
Створення нового чату	CreateBackup ChatAsync	Назва чату («Тестовий чат») та його опис («Тестовий опис»)	Екземпляр класу Peer із повною інформацією про створений чат	Правильно

Кінець таблиці 4.3

1	2	3	4	5
Отримання списку чатів для бекапу	GetPossible BackupPeers	Нормальний пріоритет	Перечислення екземплярів класу Peer, які представляють чати, в які можна виконувати резервне копіювання	Правильно
Отримання посилання на файл	ResolveFile Reference	Контекст повідомлення (отриманий із тестування вивантаження файлів)	Екземпляр класу FileContext, що представляє посилання на вивантажений раніше файл	Правильно
Створення завдання вивантаження	CreateUpload Task	Екземпляр FileData	Було створено нове завдання на вивантаження файлу	Правильно
Створення завдання завантаження	CreateDownload Task	Перечислення із екземплярами FileContext	Було створено нове завдання на завантаження файлу	Правильно
Завантаження іконки	Download ThumbnailAsync	Екземпляр ThumbnailContext (отриманий із тестування вивантаження файлів)	Іконка файлу, представлена масивом байтів	Правильно
Видалення файлу	DeleteAsync	Екземпляр MessageContext (отриманий із тестування вивантаження файлів)	Повідомлення було видалено з Telegram	Правильно

Отже, в процесі написання даного розділу були визначені та описані основні методи проведення тестування мобільних додатків. Для написання модульних та інтеграційних тестів було обрано фреймворк NUnit, а для тестування користувацького інтерфейсу було використано новітній фреймворк Xamarin.UITest.

Як видно з результатів модульного тестування Cloudgram.Base, дана бібліотека працює саме так, як і передбачалося.

Результати проведення функціональних тестів на реальних пристроях та емуляторах повністю задовольняють очікуваним результатам (так само як і результати тестів проведених з використанням фреймворку Xamarin.UITest).

Виходячи із вказаного вище, можна зробити висновок що розроблений додаток працює саме так як і передбачалося та задовольняє вимоги, які були поставлені у технічному завданні.

ВИСНОВКИ

В процесі виконання дипломного проекту було проаналізовано предметну область, з'ясовані причини частого виникнення проблеми нестачі пам'яті у користувачів смартфонів, а також досліджені наявні програмні засоби для резервного копіювання. В результаті було визначено їх переваги та недоліки, а також доведено потребу у розробці спеціалізованого програмного продукту, який призначатиметься саме для роботи з медіа-файлами. Сформовані вимоги були описані у технічному завданні та формувалися, опираючись на інші успішні додатки.

Наступним кроком стало проведення аналізу переваг та недоліків найпопулярніших архітектур мобільних додатків, і встановлено, що для розроблюваної системи найбільше підходить шаблон MVVM.

Окрім того, було проведено аналіз Telegram API, що дало можливість визначити які саме дані необхідно буде зберігати у базі даних для досягнення цілей, поставлених у завданні до дипломного проектування.

У результаті детального аналізу було проведено аналіз технологій і засобів, які будуть використовуватись в процесі розробки програмного додатка, визначено та описано основні модулі додатка і те, яким чином вони взаємодіють між собою. Також було прийнято рішення перенести модуль мережевої взаємодії з Telegram в окрему .NET Standart бібліотеку під назвою Cloudgram.Base та розробляти незалежно від основного додатку.

На основі отриманих даних було реалізовано та здійснено опис компонентів бібліотеки Cloudgram.Base та мобільного додатка Cloudgram. Розроблений в результаті додаток було протестовано та доведено що він задовольняє усім вимогам, які були поставлені до нього у технічному завданні.

В результаті виконання дипломного проекту було розроблено мобільну галерею для ОС Android, яка дозволяє її користувачам в автоматичному та ручному режимі вивантажувати медіа-файли в хмарне сховище Telegram, а

також організовувати свою галерею шляхом додавання до них тегів, за якими потім можна буде виконувати пошук. Розроблений додаток було опубліковано на платформі Google Play Market.

Таким чином, завдання поставлені у процесі дипломного проектування були виконані, а мета досягнута. Оскільки в процесі реалізації додатка використовувались новітні технології та засоби, отримані знання та вміння знадобляться у подальшій професійній діяльності.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Manouchehr Rafie. AI-Powered Camera Sensors. Computing at the Edge: Smart Cameras, Robotic Vehicles and End-Point Devices / R. Manouchehr. – VP, Advanced Technologies, 2020. – 31 p.

2. Smartphone CIS Sensors to Top Five Billion Units in 2020 as Quad-Camera Smartphone Designs Ramp Despite COVID-19» [Online] / Counterpoint. – Available: https://www.counterpointresearch.com/smartphone-cis-sensors-top-five-billion-mark-in-2020/

3. Смартфоны теряют 20% памяти из-за фотографий плохого качества [Электронный ресурс] / Блог Avast. – Режим доступа: https://press.avast.com/ru-ru/смартфоны-теряют-20-памяти-из-за-фотографий-плохого-качества

4. Moving Chat History from Other Apps [Online] / Telegram Official Blog. – Available: https://telegram.org/blog/move-history?ln=r

5. Shimrit Ben-Yair. Updating Google Photos' storage policy to build for the future [Online] / Google Blog. – Available: https://blog.google/products/photos/storage-changes/

6. David Britch. The Model-View-ViewModel Pattern [Online] / Microsoft Docs. – Available: https://docs.microsoft.com/en-us/xamarin/xamarinforms/enterprise-application-patterns/mvvm

7. Rakshit Soral. Ending the debate: MVC vs MVP vs MVVM for iOS application development [Online] / Simform. – Available: https://www.simform.com/mvc-mvp-mvvm-ios-app-development/

8. Telegram APIs [Online] / Telegram API. – Available: https://core.telegram.org/api

9. Telegram API Terms of Service [Online] / Telegram API. – Available: https://core.telegram.org/api/terms

10. End-to-End Encryption, Secret Chats [Online] / Telegram API. – Available: https://core.telegram.org/api/end-to-end

11. MTProto Mobile Protocol [Online] / Telegram API. – Available: https://core.telegram.org/mtproto

12. TL Language [Online] / Telegram API. – Available: https://core.telegram.org/mtproto/TL

13. Polymorphism in TL [Online] / Telegram API. – Available: https://core.telegram.org/mtproto/TL-polymorph

14. Type Serialization [Online] / Telegram API. – Available: https://core.telegram.org/mtproto/TL-types

15. Uploading and Downloading Files [Online] / Telegram API. – Available: https://core.telegram.org/api/files

16. Profile Videos, 2 GB File Sharing, Group Stats, Improved People Nearby and More [Online] / Telegram Official Blog. – Available: https://telegram.org/blog/profile-videos-people-nearby-and-more/ru?setln=en

17. David Ortinau. Xamarin Architecture [Online] / Xamarin Docs. – Available: https://docs.microsoft.com/en-us/xamarin/cross-platform/app-fundamentals/building-cross-platform-applications/architecture

18. Genevieve Warren. .NET Standart [Online] / Microsoft Docs. – Available: https://docs.microsoft.com/en-us/dotnet/standard/net-standard

19. Дэвид Пайн. Библиотека параллельных задач (TPL) [Электронный ресурс] / Microsoft Docs. – Режим доступа: https://docs.microsoft.com/ru-ru/dotnet/standard/parallel-programming/task-parallel-library-tpl

20. Билл Вагнер. Асинхронная модель на основе задач [Электронный ресурс] / Microsoft Docs. – Режим доступа: https://docs.microsoft.com/ru-ru/dotnet/-standard/asynchronous-programming-patterns/task-based-asynchronous-pattern-tap

ДОДАТОК А
(обов’язковий)

ТЕХНІЧНЕ ЗАВДАННЯ

Введення

Робота виконується в рамках проекту розробки мобільного додатку «Cloudgram» для ОС Android. Технічне завдання розроблено у відповідності до стандарту ГОСТ 19.201–78.

1 Підстава для розробки

Підставою для розробки є «Завдання на дипломний проект», затверджене завідувачем кафедри інженерії програмного забезпечення. Найменування розробки: «Мобільний додаток для організації медіа-файлів та їх автоматичного резервного копіювання у хмару з використанням API Telegram».

2 Призначення розробки

2.1 Функціональне призначення

Функціональним призначенням додатку є завантаження та вивантаження медіа-файлів у хмару Telegram, їх маркування через додавання тегів, пошук серед них, а також можливість звільняти місце на пристрої шляхом видалення з нього файлів, які вже було вивантажено у хмару.

2.2 Експлуатаційне призначення

Програма повинна експлуатуватися на мобільних пристроях під керуванням ОС Android. Кінцевим користувачем додатку може виступати будь-яка особа, яка також є користувачем Telegram.

3 Вимоги до програми

3.1 Вимоги до функціональних характеристик

Програма повинна забезпечувати можливості виконання таких функцій:

– функція перегляду альбомів та медіа-файлів;

– функція додавання довільної кількості тегів до одного або багатьох медіа-файлів, їх редагування та видалення;

– функція групування медіа-файлів по доданим користувачем тегам у окремі альбоми;

– функція пошуку за тегами в окремих альбомах або по всій галереї;

– можливість ділитися знайденими зображеннями через месенджери та соціальні мережі;

– можливість авторизуватися через Telegram;

– функція вивантаження медіа-файлів у Telegram (в тому числі в автоматичному режимі);

– функція завантаження медіа-файлів на пристрій із Telegram;

– функція звільнення місця через видалення з пристрою файлів, які вже зберігаються у хмарі;

– функція видалення медіа-файлу із хмари;

– функція виходу із акаунту Telegram;

– функція налаштування використання проксі-сервера;

– можливість встановлення власного API ID та API Hash;

– функція імпорту та експорту бази даних;

– функція резервного копіювання бази даних у Telegram;

– можливість вибору чатів в які буде виконуватись резервне копіювання медіа-файлів та бази даних.

3.2 Вимоги до надійності

Додаток повинен забезпечувати наступні вимоги до надійності:

– усі деструктивні дії (видалення медіа-файлу або тегів) повинні отримувати додаткове підтвердження від користувача перед тим, як ці зміни будуть застосовані;

– обробляти помилкові дії користувача і повідомляти його про це;

– можливість самовідновлення після збоїв;

– можливість резервного копіювання бази даних.

3.3 Умови експлуатації та вимоги до технічних засобів

Додаток націлений на смартфони та планшети під керуванням операційної системи Android з розрахунку, що на одному пристрої може бути встановлено не більше ніж одна копія додатку. Працездатність та коректна робота додатку не гарантується у разі використання користувачем сторонніх засобів для клонування даного програмного забезпечення. При цьому для виконання переважної більшості функцій пристрій повинен мати стабільне з’єднання з Інтернетом.

Для успішного запуску додатку пристрій повинен відповідати наступним рекомендованим вимогам:

– ОС Android 5.0 або вище;

– будь-який чотирьохядерний процесор;

– 1 ГБ ОЗП;

– 500 МБ постійної пам’яті.

3.4 Вимоги до інформаційної та програмної сумісності

При розробці додатку використовуватиметься високорівнева об’єктно-орієнтована мова C#, а також Xamarin – фреймворк з відкритим кодом, призначений для створення додатків для iOS, Android та Windows на платформі .NET. В якості СКБД використовуватиметься SQLite, а Entity Framework Core буде використовуватись як об’єктно-орієнтована технологія для спрощення роботи із нею.

3.5 Спеціальні вимоги

Програма повинна мати зручний та приємний матеріальний дизайн інтерфейсу, зрозумілий для будь-якого користувача.

4 Вимоги до програмної документації

У момент здачі проекту замовнику надається наступний набір документів:

– текст програми;

– опис програми;

– технічне завдання;

– керівництво користувача.

5 Стадії та етапи розробки

Стадії та етапи розробки програмної системи для організації медіа-файлів та їх автоматичного резервного копіювання у хмару з використанням API Telegram подані у таблиці А.1.

Таблиця А.1 – Стадії та етапи розробки проекту

Стадія розробки	Етапи робіт	Зміст робіт
1	2	3
Технічне завдання 02.01.21 – 31.01.21	Обґрунтування необхідності розробки програми	Коротка характеристика програмного забезпечення; підстава і призначення розробки; вимоги до програмної системи і документація; стадії і етапи розробки програми; порядок контролю і приймання
Ескізний проект 01.02.21 – 26.02.21	Розробка ескізного проекту	Попередня розробка структури вхідних і вихідних даних; уточнення середовища програмування; розробка і опис загальної алгоритмічної структури

Кінець таблиці А.1

1	2	3
Технічний проект 29.02.21 – 19.03.21	Розробка технічного проекту	Уточнення структури вхідних і вихідних даних; розробка докладного алгоритму; розробка структури програми
Робочий проект 20.03.21 – 15.04.21	Розробка програмного забезпечення	Реалізація програмного забезпечення; відлагодження; проведення попереднього тестування
Розробка програмної документації 16.04.21 – 22.04.21	Розробка документації до програмного забезпечення	Розробка необхідної документації, передбаченої технічним завданням
Тестування системи 23.04.21 – 30.04.21	Проведення тестування програмного забезпечення	Розробка методики тестування; проведення основних тестів; коректування програмного забезпечення
Впровадження	Підготовка і передача програми	Підготовка і розгортання програмного забезпечення
Розробка програмної документації 16.04.21 – 22.04.21	Розробка документації до програмного забезпечення	Розробка необхідної документації, передбаченої технічним завданням

6 Порядок контролю та приймання

Контроль здійснюється кінцевими користувачами системи, підключеними на етапі тестування додатку. Прийом комплексу здійснюється після проходження додатком модерації у Google Play Market.

ДОДАТОК Б
(обов’язковий)

КОД (ЛІСТИНГ) БІБЛІОТЕКИ CLOUDGRAM.BASE

Клас ConnectionStatus

```
public enum ConnectionStatus
{
    NotConnected, WaitingToReconnect, Connecting, Connected
}
```

Клас ConnectionStatusChangedEventArgs

```
public class ConnectionStatusChangedEventArgs : EventArgs
{
    public ConnectionStatus PreviousStatus { get; }
    public ConnectionStatus Status { get; }

    public ConnectionStatusChangedEventArgs(ConnectionStatus currentStatus,
ConnectionStatus previousStatus)
    {
        Status = currentStatus;
        PreviousStatus = previousStatus;
    }
}
```

Інтерфейс ILogger

```
public interface ILogger
{
    void LogDebug(string debug);
    void LogInformation(string info);
    void LogWarning(string warning);
    void LogError(string error);
    void LogFatal(string error);
}
```

Клас TgClient

```
public partial class TgClient : IDisposable
{
     public readonly TelegramClient _client;
     private readonly RequestQueue _requestQueue;
     private readonly CancellationTokenSource _executorTokenSource;
     internal readonly ILogger logger;

     private AutoResetEvent _reconnectionEvent;
     private bool _waitingToReconnect;

     private AutoResetEvent _waitHandler;
     private int _requestNumber;

     internal TLUser CachedUser => _client?.Session?.TLUser;

     private ConnectionStatus _connectionStatus;
     public ConnectionStatus ConnectionStatus
     {
```

```
        get => _connectionStatus;
        set
        {
            if(_connectionStatus != value)
            {
                OnConnectionStatusChanged?.Invoke(this,
                    new ConnectionStatusChangedEventArgs(value,
_connectionStatus)
                );

                _connectionStatus = value;
            }
        }
    }

    internal Peer BackupPeer { get; set; }
    internal Peer ThumbnailPeer { get; set; }
    internal Peer DatabasePeer { get; set; }

    public int QueuedRequestsCount => _requestQueue?.Count ?? 0;
    public bool IsUserAuthorized => _client.IsUserAuthorized();

    public event EventHandler<ConnectionStatusChangedEventArgs>
OnConnectionStatusChanged;

    public static TgClientBuilder GetBuilder() => TgClientBuilder.GetBuilder();

    internal TgClient(int apiId, string apiHash, ISessionStore store = null,
        string sessionUserId = "session", TcpClientConnectionHandler handler =
null,
        DataCenterIPVersion dcIpVersion = DataCenterIPVersion.Default,
ClientInfo clientInfo = null,
        string connectionAddress = null, int connectionPort = 0, ILogger logger
= null)
    {
        _client = new TelegramClient(apiId, apiHash, store, sessionUserId,
handler,
            dcIpVersion, clientInfo, connectionAddress, connectionPort);

        this.logger = logger;

        _requestQueue = new RequestQueue();
        _waitHandler = new AutoResetEvent(false);
        _reconnectionEvent = new AutoResetEvent(false);
        _executorTokenSource = new CancellationTokenSource();

        Task.Run(() => RequestExecutor(_executorTokenSource.Token),
                    _executorTokenSource.Token);

        ConnectionStatus = ConnectionStatus.NotConnected;
        OnConnectionStatusChanged += TgClient_OnConnectionStatusChanged;
    }

    private void TgClient_OnConnectionStatusChanged(object sender,
ConnectionStatusChangedEventArgs e)
    {
        if(e.Status == ConnectionStatus.Connected && _waitingToReconnect)
        {
            _reconnectionEvent.Set();
            logger?.LogInformation("Reconnection event was set.");
        }
    }

    public bool IsConnected()
    {
        bool connected = _client?.IsConnected ?? false;
```

```
            ConnectionStatus = connected ? ConnectionStatus.Connected
                : ConnectionStatus.NotConnected;

            return connected;
        }

        public Task ConnectAsync()
        {
            return CreatePriorityRequest<bool>(() =>
            {
                Connect().GetAwaiter().GetResult();
                return true;
            }, RequestPriority.Realtime, "ConnectionRequest");
        }

        public Task ManualConnectAsync()
        {
            return Connect();
        }

        public Task ManualReconnectAsync(TcpClientConnectionHandler handler = null,
            CancellationToken token = default)
        {
            return Reconnect(handler, token);
        }

        private async Task Connect()
        {
            var previousConnectionStatus = ConnectionStatus;
            try
            {
                ConnectionStatus = ConnectionStatus.Connecting;
                await _client.ConnectAsync();
                ConnectionStatus = ConnectionStatus.Connected;
            }
            catch(Exception ex)
            {
                if(previousConnectionStatus == ConnectionStatus.WaitingToReconnect)
                {
                    ConnectionStatus = ConnectionStatus.WaitingToReconnect;
                }
                else
                {
                    ConnectionStatus = ConnectionStatus.NotConnected;
                }

                throw new ClientNotConnectedException(-1, ex.Message);
            }
        }

        private async Task Reconnect(TcpClientConnectionHandler handler = null,
            CancellationToken token = default)
        {
            var previousConnectionStatus = ConnectionStatus;
            try
            {
                ConnectionStatus = ConnectionStatus.Connecting;
                await _client.ReconnectAsync(handler, token);
                ConnectionStatus = ConnectionStatus.Connected;
            }
            catch (Exception ex)
            {
                if (previousConnectionStatus ==
ConnectionStatus.WaitingToReconnect)
                {
```

```
                ConnectionStatus = ConnectionStatus.WaitingToReconnect;
            }
            else
            {
                ConnectionStatus = ConnectionStatus.NotConnected;
            }

            throw new ClientNotConnectedException(-1, ex.Message);
        }
    }

    public Task<string> SendCodeRequestAsync(string phoneNumber)
    {
        return CreatePriorityRequest<string>(() =>
        {
            return _client.SendCodeRequestAsync(phoneNumber)
                .GetAwaiter().GetResult();
        }, RequestPriority.Realtime, "SendCodeRequest");
    }

    public Task<TLUser> SignInViaCodeAsync(string phoneNumber,
        string hash, string code)
    {
        return CreatePriorityRequest<TLUser>(() =>
        {
            return _client.MakeAuthAsync(phoneNumber, hash, code)
                .GetAwaiter().GetResult();
        }, RequestPriority.Realtime, "SignInViaCodeRequest");
    }

    public Task<TLUserFull> GetMeAsync(RequestPriority priority =
RequestPriority.Normal)
    {
        return CreatePriorityRequest<TLUserFull>(() => {
            return _client.GetMeAsync().GetAwaiter().GetResult();
        }, priority, "GetMeRequest");
    }


    public Task<TLAbsDialogs> GetUserDialogsAsync(int limit, RequestPriority
priority = RequestPriority.Normal,
        CancellationToken token = default)
    {
        return CreatePriorityRequest<TLAbsDialogs>(() => {
            return _client.GetUserDialogsAsync(limit: limit, token:
token).GetAwaiter().GetResult();
        }, priority, "GetUserDialogsRequest");
    }

    internal Task<T> SendRequestAsync<T>(TLMethod method, RequestPriority
priority = RequestPriority.Normal,
        CancellationToken token = default)
    {
        return CreatePriorityRequest<T>(() =>
        {
            return _client.SendRequestAsync<T>(method,
token).GetAwaiter().GetResult();
        }, priority, method.GetType().Name);
    }

    public Task<byte[]> GetUserPhotoAsync(TLUserFull user, bool bigSize,
        RequestPriority priority = RequestPriority.Normal, CancellationToken
token = default)
    {
        token.ThrowIfCancellationRequested();
```

```
            var t = Task.Run(() =>
            {
                var photo = user.ProfilePhoto as TLPhoto;

                if (photo != null)
                {
                    var photoLocation = new TLInputPhotoFileLocation
                    {
                        FileReference = photo.FileReference,
                        AccessHash = photo.AccessHash,
                        Id = photo.Id,
                        ThumbSize = (photo.Sizes.Last() as TLPhotoSize).Type
                    };
                    byte[] fileBytes = GetFileBytes(photoLocation, priority,
token).Result;
                    return fileBytes;
                }

                return null;
            }, token);

            return t;
        }

        public Task<byte[]> GetChannelPhotoAsync(TLChannel channel, bool bigSize,
            RequestPriority priority = RequestPriority.Normal, CancellationToken
token = default)
        {
            token.ThrowIfCancellationRequested();

            var t = Task.Run(() =>
            {
                var photo = channel.Photo as TLChatPhoto;
                if (photo != null)
                {
                    var photoLocation = (TLFileLocationToBeDeprecated)(bigSize ?
photo.PhotoBig
                        : photo.PhotoSmall);
                    byte[] fileBytes = GetFileBytes(photoLocation, (int)priority,
token).Result;
                    return fileBytes;
                }
                return null;
            }, token);

            return t;
        }

        public Task<bool> LogoutAsync()
        {
            return CreatePriorityRequest<bool>(() =>
            {
                return _client.LogoutAsync().GetAwaiter().GetResult();
            }, RequestPriority.High, "LogoutRequest");
        }

        public Task ClearSession()
        {
            return CreatePriorityRequest<bool>(() =>
            {
                _client.ClearSession().GetAwaiter().GetResult();
                return true;
            }, RequestPriority.High, "LogoutRequest");

        }
```

```
     public void Dispose()
     {
         _executorTokenSource.Cancel();
         _client.Dispose();
     }
}
```

Клас Backuper

```
public class Backuper
{
    const string DEBUG_TAG = "Backuper";

    private readonly TgClient _client;
    private SynchronizationContext _synchronizationContext;
    private List<Peer> _cachedPeers;

    public event EventHandler<FloodRequestStateChangedEventArgs>
OnFloodWaitedRequestStateChanged;
    public event EventHandler<ConnectionStatusChangedEventArgs>
OnConnectionStatusChanged;

    public Backuper(TgClient loggedClient, SynchronizationContext
synchronizationContext = null)
    {
        _client = loggedClient;
        _synchronizationContext = synchronizationContext;
        _client.OnFloodWaitedRequestStateChanged +=
_client_OnFloodWaitedRequestStateChanged;
        _client.OnConnectionStatusChanged += _client_OnConnectionStatusChanged;
    }

    private void _client_OnConnectionStatusChanged(object sender,
ConnectionStatusChangedEventArgs e)
    {
        if (_synchronizationContext != null)
        {
            _synchronizationContext.Post(obj =>
            {
                OnConnectionStatusChanged?.Invoke(this, e);
            }, null);
        }
        else OnConnectionStatusChanged?.Invoke(this, e);
    }

    private void _client_OnFloodWaitedRequestStateChanged(object sender,
FloodRequestStateChangedEventArgs e)
    {
        if(_synchronizationContext != null)
        {
            _synchronizationContext.Post(obj =>
            {
                OnFloodWaitedRequestStateChanged?.Invoke(this, e);
            }, null);
        } else OnFloodWaitedRequestStateChanged?.Invoke(this, e);
    }

    public void SetSynchronizationContext(SynchronizationContext
synchronizationContext)
    {
        _synchronizationContext = synchronizationContext;
    }
```

```
    public Peer GetMainBackupPeer() => _client.BackupPeer;
    public void SetMainBackupPeer(Peer backupPeer)
    {
        _client.BackupPeer = backupPeer;
    }

    public Peer GetThumbnailBackupPeer() => _client.ThumbnailPeer;
    public void SetThumbnailBackupPeer(Peer backupPeer)
    {
        _client.ThumbnailPeer = backupPeer;
    }

    public Peer GetDatabaseBackupPeer() => _client.DatabasePeer;
    public void SetDatabaseBackupPeer(Peer databsePeer)
    {
        _client.DatabasePeer = databsePeer;
    }

    public async Task InitPeersAsync(int mainPeerId, int thumbPeerId, int
dbPeerId)
    {
        var res = await GetPossibleBackupPeersAsync(RequestPriority.Realtime);
        if (res != null)
        {
            SetMainBackupPeer(res.FirstOrDefault(p => p.Id == mainPeerId));
            SetThumbnailBackupPeer(res.FirstOrDefault(p => p.Id ==
thumbPeerId));
            SetDatabaseBackupPeer(res.FirstOrDefault(p => p.Id == dbPeerId));
        }
    }

    public async Task<Peer> ResolvePeerByIdAsync(int channelId, bool
getAccessHashOnly = false,
        RequestPriority requestPriority = RequestPriority.Normal)
    {
        Peer peer = null;

        if(channelId == -1)
        {
            return Peer.SavedMessages;
        }

        var peers = await GetPossibleBackupPeersAsync();
        peer = peers?.FirstOrDefault(p => p.Id == channelId);

        return peer;
    }

    public async Task<Peer> CreateBackupChannelAsync(string title, string
description)
    {
        var newChannel = new TLRequestCreateChannel()
        {
            Broadcast = true,
            Megagroup = false,
            Title = title,
            About = description
        };

        var upd = (await _client.SendRequestAsync<TLAbsUpdates>(newChannel)) as
TLUpdates;
        var channel = upd.Chats.ElementAtOrDefault(0) as TLChannel;
        if(channel != null)
        {
            Peer peer = new Peer
```

```
            {
                Id = channel.Id,
                AccessHash = channel.AccessHash,
                Title = channel.Title,
                ParticipantsCount = 1,
                Description = description,
                Type = PeerType.PrivateChannel
            };

            if(_cachedPeers != null)
            {
                if (_cachedPeers.Count > 0)
                {
                    _cachedPeers?.Insert(0, peer);
                }
                else
                {
                    _cachedPeers.Add(peer);
                }
            }

            return peer;
            //return await ChannelToPeer(channel);
        }

        return null;
    }

    private object lockPeersLoadingObject = new object();
    public Task<ReadOnlyCollection<Peer>>
GetPossibleBackupPeersAsync(RequestPriority requestPriority =
RequestPriority.High,
        bool downloadChannelPhoto = false)
    {
        return Task.Run(() =>
        {
            lock (lockPeersLoadingObject)
            {
                if (_cachedPeers != null)
                {
                    _client.logger?.LogInformation($"[{DEBUG_TAG}] Using cached
peers.");
                    return _cachedPeers.AsReadOnly();
                }

                _client.logger?.LogInformation($"[{DEBUG_TAG}] Caching
peers...");
                List<Peer> peers = new List<Peer>();

                //var absDialogs = await _client.GetUserDialogsAsync(100);
                //var dialogs =  absDialogs as TLDialogsSlice;
                var req = new TLRequestGetAllChats { ExceptIds = new
TLVector<int>() };
                var res = _client.SendRequestAsync<TLAbsChats>(req,
requestPriority).GetAwaiter().GetResult();
                var chats = res as TLChats;
                if (chats == null) return null;
                var channels = chats.Chats.OfType<TLChannel>();

                foreach (var channel in channels)
                {

                    if (channel.Creator && channel.Broadcast && !channel.Left &&
!channel.Restricted
                        && channel.Username == null)
                    {
```

```
                        Peer peer = ChannelToPeerAsync(channel,
downloadChannelPhoto, requestPriority).GetAwaiter().GetResult();
                        peers.Add(peer);
                    }
                }

                _cachedPeers = peers;
                _client.logger?.LogInformation($"[{DEBUG_TAG}] Peers cached.");
                return _cachedPeers.AsReadOnly();
            }
        });
    }

    private async Task<Peer> ChannelToPeerAsync(TLChannel channel, bool
downloadChannelPhoto = false,
        RequestPriority requestPriority = RequestPriority.Normal)
    {
        Peer peer = new Peer
        {
            Id = channel.Id,
            AccessHash = channel.AccessHash,
            Title = channel.Title
        };

        if (downloadChannelPhoto)
        {
            peer.Photo = await _client.GetChannelPhotoAsync(channel, false,
requestPriority);
        }

        var reqFullChannel = new TLRequestGetFullChannel()
        {
            Channel = new TLInputChannel()
            {
                ChannelId = peer.Id,
                AccessHash = peer.AccessHash ?? 0
            }
        };

        var fullChat = await
_client.SendRequestAsync<TgSharp.TL.Messages.TLChatFull>(reqFullChannel,
requestPriority);
        var fullChannel = fullChat.FullChat as TLChannelFull;
        peer.ParticipantsCount = fullChannel.ParticipantsCount ?? 0;
        peer.Description = fullChannel.About;

        return peer;
    }

    public async Task<FileContext> ResolveFileReference(MessageContext
messageContext,
        bool retrieveThumbnailContext)
    {
        FileContext fileContext = null;
        TLMethod request;
        TLMessage message;

        if(messageContext.ChatId == -1)
        {
            request = new TgSharp.TL.Messages.TLRequestGetMessages
            {
                Id = new TLVector<TLAbsInputMessage>(new[] { new
TLInputMessageID { Id = messageContext.MessageId } })
            };
```

```
            var foundMessages = await
_client.SendRequestAsync<TLMessages>(request);
            message = foundMessages.Messages?.FirstOrDefault() as TLMessage;
        }
        else
        {
            Peer peer;
            if (messageContext.ChatId == _client.BackupPeer?.Id) peer =
_client.BackupPeer;
            else peer = await ResolvePeerByIdAsync(messageContext.ChatId,
getAccessHashOnly: true);

            if(peer == null)
            {
                return null;
            }

            var inputChannel = new TLInputChannel
            {
                ChannelId = peer.Id,
                AccessHash = peer.AccessHash ?? 0
            };
            request = new TgSharp.TL.Channels.TLRequestGetMessages
            {
                Channel = inputChannel,
                Id = new TLVector<TLAbsInputMessage>(new[] { new
TLInputMessageID { Id = messageContext.MessageId } })
            };

            var foundMessages = await
_client.SendRequestAsync<TLChannelMessages>(request);

            message = foundMessages.Messages?.FirstOrDefault() as TLMessage;
        }

        if (message != null)
        {
            if (message.Media != null && message.Media is TLMessageMediaDocument
mdoc)
            {
                var document = mdoc.Document as TLDocument;
                fileContext = new FileContext
                {
                    MessageContext = messageContext,
                    FileId = document.Id,
                    AccessHash = document.AccessHash,
                    Version = 0,
                    FileReference = document.FileReference,
                    Size = document.Size
                };

                if(retrieveThumbnailContext)
                {
                    var thumbSize = document.Thumbs.Last() as TLPhotoSize;
                    if (thumbSize == null)
                        return fileContext;

                    var thumbLocation = thumbSize.Location as
TLFileLocationToBeDeprecated;
                    fileContext.Thumbnail = new ThumbnailContext
                    {
                        IsAutoGen = true,
                        FileId = thumbLocation.LocalId,
                        Secret = 0,
                        VolumeId = thumbLocation.VolumeId
                    };
```

```
                }
            }
        }

        return fileContext;
    }

    public UploadTask CreateUploadTask(FileData file, RequestPriority priority =
RequestPriority.Normal)
    {
        var uploadTask = new UploadTask(_client, file, priority,
_synchronizationContext);
        return uploadTask;
    }

    public DownloadTask CreateDownloadTask(FileContext fileContext,
        RequestPriority priority = RequestPriority.Normal)
    {
        return CreateDownloadTask(new[] { fileContext }, priority);
    }

    public DownloadTask CreateDownloadTask(IEnumerable<FileContext>
fileContexts,
        RequestPriority priority = RequestPriority.Normal)
    {
        var downloadTask = new DownloadTask(_client, fileContexts, priority,
            _synchronizationContext);
        return downloadTask;
    }

    public async Task<byte[]> DownloadThumbnailAsync(ThumbnailContext
thumbnailContext,
        RequestPriority priority = RequestPriority.Normal, CancellationToken
token = default)
    {
        byte[] result;

        if (thumbnailContext.IsAutoGen)
        {
            var inputFileLocation = new TLInputFileLocation
            {
                LocalId = (int)thumbnailContext.FileId,
                Secret = thumbnailContext.Secret,
                VolumeId = thumbnailContext.VolumeId
            };

            result = await _client.GetFileBytes(inputFileLocation, priority,
token);
        }
        else
        {
            var fileContext = await
ResolveFileReference(thumbnailContext.MessageContext, false);
            var downloadResult = await CreateDownloadTask(fileContext,
priority).Run();
            result = downloadResult.FileData;
        }

        return result;
    }

    public BackupTask CreateBackupTask(IEnumerable<FileData> inputFiles,
        RequestPriority priority = RequestPriority.Lowest)
    {
        var backupTask = new BackupTask(_client, inputFiles, priority,
_synchronizationContext);
```

```
        return backupTask;
    }

    public async Task Delete(MessageContext messageContext)
    {
        TLMethod request;
        if (messageContext.ChatId == -1)
        {
            request = new TgSharp.TL.Messages.TLRequestDeleteMessages
            {
                Id = new TLVector<int>(new[] { messageContext.MessageId }),
                Revoke = true
            };
        }
        else
        {
            var peer = await ResolvePeerByIdAsync(messageContext.ChatId,
getAccessHashOnly: true);
            var inputChannel = new TLInputChannel
            {
                ChannelId = peer.Id,
                AccessHash = peer.AccessHash ?? 0
            };
            request = new TgSharp.TL.Channels.TLRequestDeleteMessages
            {
                Channel = inputChannel,
                Id = new TLVector<int>(new[] { messageContext.MessageId })
            };
        }

        await _client.SendRequestAsync<TLAffectedMessages>(request);
    }
}
```

Клас TransmissionTask

```
public abstract class TransmissionTask : IContinuousRequest
{
    private SynchronizationContext _synchronizationContext;

    protected readonly CancellationTokenSource cts;
    protected readonly TgClient client;
    protected readonly CancellationToken cancellationToken;
    protected Task transmissionTask;

    public RequestPriority Priority { get; protected set; }

    private TransmissionState _state;
    public TransmissionState State
    {
        get => _state;
        protected set
        {
            if (_state != value)
            {
                _state = value;
                RaiseStateChanged(new
TransmissionStateChangedEventArgs(_state));
            }
        }
    }
```

```
    public bool IsCancellationRequested => cts.IsCancellationRequested;
    public object Result { get; protected set; }

    public event EventHandler<TransmissionStateChangedEventArgs>
OnTransmissionStateChanged;
    public event EventHandler<ProgressChangedEventArgs> OnProgressChanged;
    public event EventHandler OnStarted;
    public event EventHandler OnCompleted;
    public event EventHandler<RequestErrorEventArgs> OnError;

    public TransmissionTask(TgClient client, RequestPriority
transmissionTaskPriority,
        SynchronizationContext synchronizationContext,
        CancellationTokenSource cts = null)
    {
        this.client = client;
        _synchronizationContext = synchronizationContext;
        Priority = transmissionTaskPriority;

        if(cts == null)
        {
            this.cts = new CancellationTokenSource();
        }
        else
        {
            this.cts = cts;
        }

        cancellationToken = this.cts.Token;

        SetMainTask();
        SetCompletingTask();
    }

    protected abstract void SetMainTask();

    protected virtual void SetCompletingTask()
    {
        transmissionTask.ContinueWith(tt =>
        {
            if (tt.Status == TaskStatus.RanToCompletion)
            {
                State = TransmissionState.Completed;
                RaiseCompleted();
            }
            else if (tt.Status == TaskStatus.Canceled)
            {
                State = TransmissionState.Cancelled;
                RaiseError(tt.Exception);
            }
            else if (tt.Status == TaskStatus.Faulted)
            {
                State = TransmissionState.Error;
                RaiseError(tt.Exception);
            }
        });
    }

    public virtual void Start()
    {
        if (State != TransmissionState.WaitingToStart)
        {
            throw new TransmissionTaskAlreadyStartedException(this);
        }

        transmissionTask.Start();
```

```
    }

    public virtual Task Run()
    {
        if(State == TransmissionState.WaitingToStart)
        {
            Start();
        }

        return transmissionTask;
    }

    protected virtual void RaiseStateChanged(TransmissionStateChangedEventArgs
e)
    {
        if(_synchronizationContext != null)
        {
            _synchronizationContext.Post((obj) =>
OnTransmissionStateChanged?.Invoke(this, e), null);
        } else OnTransmissionStateChanged?.Invoke(this, e);
    }

    protected virtual void RaiseProgressChanged(ProgressChangedEventArgs e)
    {
        if (_synchronizationContext != null)
        {
            _synchronizationContext.Post((obj) =>
OnProgressChanged?.Invoke(this, e), null);
        }
        else OnProgressChanged?.Invoke(this, e);
    }

    protected virtual void RaiseStarted()
    {
        if (_synchronizationContext != null)
        {
            _synchronizationContext.Post((obj) => OnStarted?.Invoke(this,
EventArgs.Empty), null);
        }
        else OnStarted?.Invoke(this, EventArgs.Empty);
    }

    protected virtual void RaiseCompleted()
    {
        if (_synchronizationContext != null)
        {
            _synchronizationContext.Post((obj) => OnCompleted?.Invoke(this,
EventArgs.Empty), null);
        }
        else OnCompleted?.Invoke(this, EventArgs.Empty);
    }

    protected virtual void RaiseError(Exception ex)
    {
        if (_synchronizationContext != null)
        {
            _synchronizationContext.Post((obj) => OnError?.Invoke(this, new
RequestErrorEventArgs(ex)), null);
        }
        else OnError?.Invoke(this, new RequestErrorEventArgs(ex));
    }

    public virtual void Cancel()
    {
        cts.Cancel();
    }
```

```
}
```

Клас UploadTask

```
public sealed class UploadTask : TransmissionTask
{
    private FileData _inputFileData;
    public FileData InputFileData { get => _inputFileData; }

    private UploadResult _result;
    new public UploadResult Result
    {
        get => _result;
        private set
        {
            _result = value;
            base.Result = value;
        }
    }

    internal UploadTask(TgClient client, FileData inputFileData, RequestPriority
priority,
        SynchronizationContext synchronizationContext,
        CancellationTokenSource cts = null) : base(client, priority,
synchronizationContext, cts)
    {
        _inputFileData = inputFileData;
        State = TransmissionState.WaitingToStart;
    }

    protected override void SetMainTask()
    {
        transmissionTask = new Task<UploadResult>(() =>
        {
            cancellationToken.ThrowIfCancellationRequested();
            State = TransmissionState.Started;
            RaiseStarted();

            var fileContexts = client.UploadAsync(_inputFileData,
RaiseProgressChanged, Priority, cancellationToken).Result;
            var uploadResult = new UploadResult(this, fileContexts);
            Result = uploadResult;
            return uploadResult;
        });
    }

    new public Task<UploadResult> Run()
    {
        return base.Run() as Task<UploadResult>;
    }
}
```

Клас DownloadTask

```
public sealed class DownloadTask : TransmissionTask
{
    private IEnumerable<FileContext> _fileContexts;
    public IEnumerable<FileContext> FileContexts { get => _fileContexts; }
```

```
    private DownloadResult _result;
    new public DownloadResult Result
    {
        get => _result;
        set
        {
            _result = value;
            base.Result = value;
        }
    }

    internal DownloadTask(TgClient client, IEnumerable<FileContext>
fileContexts,
        RequestPriority priority, SynchronizationContext synchronizationContext)
: base(client, priority, synchronizationContext)
    {
        _fileContexts = fileContexts;
        State = TransmissionState.WaitingToStart;
    }

    protected override void SetMainTask()
    {
        transmissionTask = new Task<DownloadResult>(() =>
        {
            DownloadResult downloadResult;
            cancellationToken.ThrowIfCancellationRequested();
            State = TransmissionState.Started;
            RaiseStarted();

            byte[] fileBytes = client.DownloadAsync(FileContexts,
RaiseProgressChanged, Priority, cancellationToken).Result;
            downloadResult = new DownloadResult(this, fileBytes);

            Result = downloadResult;
            return downloadResult;
        });
    }

    new public Task<DownloadResult> Run()
    {
        return base.Run() as Task<DownloadResult>;
    }
}
```

Клас BackupTask

```
public sealed class BackupTask : TransmissionTask, IDisposable
{
    private IEnumerable<FileData> _inputFileDatas;
    public IEnumerable<FileData> InputFileDatas { get =>
_inputFileDatas.ToList().AsReadOnly(); }

    public event EventHandler<EventArgs> OnFileUploaded;

    private int _currentFileNumber;
    private int _totalFiles;

    private TaskCompletionSource<BackupResult> _waitCompletionSource;
    private Task _mainTask;

    private List<UploadResult> _uploadResults;
    private List<UploadErrorResult> _uploadErrorResults;
```

```
private BackupResult _result;
new public BackupResult Result
{
    get => _result;
    set
    {
        base.Result = value;
        _result = value;
    }
}

public BackupTask(TgClient client, IEnumerable<FileData> inputFileDatas,
    RequestPriority priority, SynchronizationContext synchronizationContext)
        : base(client, priority, synchronizationContext)
{
    _inputFileDatas = inputFileDatas;
    _totalFiles = _inputFileDatas.Count();
    _uploadResults = new List<UploadResult>(_totalFiles);
    _uploadErrorResults = new List<UploadErrorResult>();

    State = TransmissionState.WaitingToStart;
}

protected override void SetMainTask()
{
    _waitCompletionSource = new TaskCompletionSource<BackupResult>();
    transmissionTask = _waitCompletionSource.Task;
    _mainTask = new Task(() =>
    {
        if(cancellationToken.IsCancellationRequested)
        {
            _waitCompletionSource.SetCanceled();
            return;
        }
        State = TransmissionState.Started;
        RaiseStarted();
        BackupNext();
    }, cancellationToken, TaskCreationOptions.LongRunning);
}

public override void Start()
{
    if (State != TransmissionState.WaitingToStart)
    {
        throw new TransmissionTaskAlreadyStartedException(this);
    }

    _mainTask.Start();
}

private void BackupNext()
{
    if(cancellationToken.IsCancellationRequested)
    {
        _waitCompletionSource.SetCanceled();
        return;
    }

    FileData inputFileData = InputFileDatas.ElementAt(_currentFileNumber++);

    UploadTask uploadTask = new UploadTask(client, inputFileData, Priority,
        synchronizationContext: null, cts);
    uploadTask.OnStarted += FileStarted;
    uploadTask.OnCompleted += FileCompleted;
    uploadTask.OnProgressChanged += FileProgressChanged;
```

```
        uploadTask.OnError += FileError;

        uploadTask.Start();
    }

    private void FileError(object sender, RequestErrorEventArgs e)
    {
        var uploadTask = sender as UploadTask;
        _uploadErrorResults.Add(new UploadErrorResult(uploadTask, e.Exception));

        FileData currentFile = InputFileDatas.ElementAt(_currentFileNumber - 1);
        RaiseProgressChanged(new ProgressChangedEventArgs(currentFile,
_currentFileNumber,
            _totalFiles, 0, currentFile.Size));
        Unsubscribe(uploadTask);

        CheckCompletion();
    }

    private void FileProgressChanged(object sender, ProgressChangedEventArgs e)
    {
        FileData currentFile = InputFileDatas.ElementAt(_currentFileNumber - 1);
        RaiseProgressChanged(new ProgressChangedEventArgs(currentFile,
_currentFileNumber,
            _totalFiles, e.CompletedFileBytes, e.TotalFileBytes));
    }

    private void FileCompleted(object sender, EventArgs e)
    {
        UploadTask uploadTask = sender as UploadTask;
        _uploadResults.Add(uploadTask.Result);

        OnFileUploaded?.Invoke(uploadTask, e);

        FileData currentFile = InputFileDatas.ElementAt(_currentFileNumber - 1);
        RaiseProgressChanged(new ProgressChangedEventArgs(currentFile,
_currentFileNumber,
            _totalFiles, currentFile.Size, currentFile.Size));
        Unsubscribe(uploadTask);

        CheckCompletion();
    }

    private void CheckCompletion()
    {
        if (_currentFileNumber < _totalFiles)
        {
            BackupNext();
        }
        else
        {
            BackupResult backupResult = new BackupResult(_uploadResults,
_uploadErrorResults);
            Result = backupResult;
            _waitCompletionSource.TrySetResult(backupResult);
            RaiseCompleted();
        }
    }

    private void FileStarted(object sender, EventArgs e)
    {
        FileData currentFile = InputFileDatas.ElementAt(_currentFileNumber - 1);
        RaiseProgressChanged(new ProgressChangedEventArgs(currentFile,
_currentFileNumber,
            _totalFiles, 0, currentFile.Size));
    }
```

```
    private void Unsubscribe(UploadTask uploadTask)
    {
        uploadTask.OnStarted -= FileStarted;
        uploadTask.OnCompleted -= FileCompleted;
        uploadTask.OnProgressChanged -= FileProgressChanged;
        uploadTask.OnError -= FileError;
    }

    new public Task<BackupResult> Run()
    {
        return base.Run() as Task<BackupResult>;
    }

    public void Dispose()
    {
        Result = null;
        _inputFileDatas = null;
        _waitCompletionSource = null;
        _mainTask = null;
        _uploadResults = null;
        _uploadErrorResults = null;
    }
}
```

ДОДАТОК В
(обов’язковий)

КОД (ЛІСТИНГ) ДОДАТКУ CLOUDGRAM

Код класу BackupedMediaService

```
public class BackupedMediaService : IBackupedMediaService, IDisposable
{
  const string DEBUG_TAG = "BackupedMediaService";

  SemaphoreSlim _semaphoreSlim;
  ApplicationContext db;
  string _dbPath;
  ILogger logger;

  public bool IsLoadingCompleted { get; private set; }

  public BackupStatistics BackupStatistics { get; private set; }

  public HashSet<Tag> Tags { get; private set; } = new HashSet<Tag>();
  public List<BackupedMediaVmo> BackupedMediaVmos { get; private set; } = new
List<BackupedMediaVmo>();

  public event EventHandler<BackupedMediaVmoUpdatedEventArgs> OnMediaBackuped;
  public event EventHandler<BackupedMediaVmoUpdatedEventArgs>
OnMediaDeletedFromDb;
  public event EventHandler<BackupedMediaVmoUpdatedEventArgs>
OnMediaTagsChanged;
  public event EventHandler<BackupedMediaVmoLoadedEventArgs> OnMediaLoaded;
  public event EventHandler OnLoadCompleted;

  public BackupedMediaService(string dbPath = null, bool invokeFromService =
false)
  {
      if(!invokeFromService)
      {
          logger = DependencyService.Get<ILogger>();
          _dbPath = dbPath ??
DependencyService.Get<IPathService>().GetDbPath(App.DB_FILE_NAME);
      }
      else
      {
          _dbPath = dbPath;
      }

      _semaphoreSlim = new SemaphoreSlim(1, 1);
      db = new ApplicationContext(_dbPath);
  }

  public async Task<List<Tag>> ResolveTagsAsync(IEnumerable<string> tagTexts)
  {
      List<Tag> tags = new List<Tag>();

      await _semaphoreSlim.WaitAsync();
      try
      {
          //using (ApplicationContext db = new ApplicationContext(_dbPath))
          {
              foreach (var tagText in tagTexts)
              {
                  var entry = await db.Tags.FirstOrDefaultAsync(t => t.TagName
== tagText);
                  if (entry != null)
                  {
                      tags.Add(entry);
                      continue;
                  }
```

```
                Tag tag = new Tag() { TagName = tagText };
                db.Tags.Add(tag);
                await db.SaveChangesAsync();

                this.Tags.Add(tag);
                tags.Add(tag);
            }
        }
    }
    finally
    {
        _semaphoreSlim.Release();
    }

    return tags;
  }

  public async Task AddTagsToMediaAsync(BackupedMediaVmo backupedMediaVmo,
IEnumerable<Tag> resolvedTags)
  {
    await _semaphoreSlim.WaitAsync();
    try
    {
        //using (ApplicationContext db = new ApplicationContext(_dbPath))
        {
            foreach (var tag in resolvedTags)
            {
                if (backupedMediaVmo.Tags.Contains(tag))
                {
                    continue;
                }

                backupedMediaVmo.BackupedMedia.BackupedMediaTags.Add(new
BackupedMediaTag
                {
                    BackupedMedia = backupedMediaVmo.BackupedMedia,
                    Tag = tag
                });
            }

            await db.SaveChangesAsync();
        }
    }
    finally
    {
        _semaphoreSlim.Release();
    }
  }

  public async Task RemoveTagsFromMediaAsync(BackupedMediaVmo backupedMediaVmo,
IEnumerable<Tag> resolvedTags)
  {
    await _semaphoreSlim.WaitAsync();
    try
    {
        //using (ApplicationContext db = new ApplicationContext(_dbPath))
        {
            foreach (var tag in resolvedTags)
            {
                BackupedMediaTag entry = backupedMediaVmo.BackupedMedia
                    .BackupedMediaTags.FirstOrDefault(bmt => bmt.Tag.TagId ==
tag.TagId);
                if (entry != null)
                {

backupedMediaVmo.BackupedMedia.BackupedMediaTags.Remove(entry);
```

```
                }
            }

            await db.SaveChangesAsync();
        }
    }
    finally
    {
        _semaphoreSlim.Release();
    }
  }

  public async Task SetTagsForMediaAsync(BackupedMediaVmo backupedMediaVmo,
IEnumerable<Tag> resolvedTags)
  {
      BackupedMedia backupedMedia;
      IEnumerable<Tag> tagsToAdd =
resolvedTags.Except(backupedMediaVmo.Tags).ToList();
      IEnumerable<Tag> tagsToRemove =
backupedMediaVmo.Tags.Except(resolvedTags).ToList();

      await _semaphoreSlim.WaitAsync();
      try
      {
          //using (ApplicationContext db = new ApplicationContext(_dbPath))
          {
              backupedMedia = await
db.FindAsync<BackupedMedia>(backupedMediaVmo.Id);

              foreach (var tag in tagsToRemove)
              {
                  BackupedMediaTag entry = await
db.BackupedMediaTags.FirstOrDefaultAsync(bmt =>
                      bmt.TagId == tag.TagId && bmt.BackupedMediaId ==
backupedMediaVmo.Id);
                  if (entry != null)
                  {
                      backupedMedia.BackupedMediaTags.Remove(entry);

backupedMediaVmo.BackupedMedia.BackupedMediaTags.Remove(entry);
                  }
              }

              foreach (var tag in tagsToAdd)
              {
                  var bmt = new BackupedMediaTag
                  {
                      BackupedMediaId = backupedMediaVmo.Id,
                      BackupedMedia = backupedMediaVmo.BackupedMedia,
                      TagId = tag.TagId,
                      Tag = tag
                  };

                  backupedMedia.BackupedMediaTags.Add(bmt);
                  backupedMediaVmo.BackupedMedia.BackupedMediaTags.Add(bmt);
              }

              await db.SaveChangesAsync();
              OnMediaTagsChanged?.Invoke(this, new
BackupedMediaVmoUpdatedEventArgs(backupedMediaVmo));
          }
      }
      finally
      {
          _semaphoreSlim.Release();
      }
```

```
}

public async Task<BackupedMediaVmo> AddMediaAsync(LocalMedia localMedia,
    IEnumerable<FileContext> fileContexts, string passwordHash = null)
{
    BackupedMediaVmo backupedMediaVmo = null;

    await _semaphoreSlim.WaitAsync();
    try
    {
        //using (ApplicationContext db = new ApplicationContext(_dbPath))
        {
            // Adding thumbnail
            FileContext fileContext = fileContexts.ElementAt(0);
            ThumbnailLocation thumbnailLocation = new ThumbnailLocation
            {
                ChatId = fileContext.Thumbnail.MessageContext.ChatId,
                MessageId = fileContext.Thumbnail.MessageContext.MessageId,
                Size = fileContext.Thumbnail.Size
            };
            await db.ThumbnailLocations.AddAsync(thumbnailLocation);

            // Adding file contexts
            List<FileLocation> fileLocations = new List<FileLocation>();
            foreach (var fc in fileContexts)
            {
                FileLocation fl = new FileLocation
                {
                    ChatId = fc.MessageContext.ChatId,
                    MessageId = fc.MessageContext.MessageId,
                    Size = fc.Size,
                    ThumbnailLocation = thumbnailLocation
                };

                fileLocations.Add(fl);
            }
            await db.FileLocations.AddRangeAsync(fileLocations);

            // Adding backuped media
            BackupedMedia backupedMedia = new BackupedMedia
            {
                FilePath = localMedia.Path,
                MimeType = localMedia.MimeType,
                IsVideo = localMedia.IsVideo,
                Size = localMedia.Size,
                Album = localMedia.Album,
                ThumbnailPath = localMedia.ThumbPath,
                CreatedAt = localMedia.CreatedAt,
                IsEncrypted = passwordHash != null,
                PasswordHash = passwordHash,
                FileLocations = fileLocations
            };

            await db.BackupedMedias.AddAsync(backupedMedia);
            await db.SaveChangesAsync();
            backupedMediaVmo = new BackupedMediaVmo(backupedMedia);
        }
    }
    finally
    {
        _semaphoreSlim.Release();
    }

    BackupedMediaVmos.Add(backupedMediaVmo);
    BackupStatistics?.AddBytes(backupedMediaVmo.BackupedMedia.Size);
```

```
      OnMediaBackuped?.Invoke(this, new
BackupedMediaVmoUpdatedEventArgs(backupedMediaVmo));
      return backupedMediaVmo;
  }

  public async Task LoadBackupedMediaVmosAsync()
  {
      IsLoadingCompleted = false;
      List<BackupedMedia> backupedItems = null;
      Tags.Clear();
      BackupedMediaVmos.Clear();

      await _semaphoreSlim.WaitAsync();
      try
      {
          //using (ApplicationContext db = new ApplicationContext(_dbPath))
          {
              var tags = await db.Tags.Include(t =>
t.BackupedMediaTags).ToListAsync();
              foreach (var tag in tags)
              {
                  Tags.Add(tag);
              }

              var thumbs = await db.ThumbnailLocations.ToListAsync();

              backupedItems = await db.BackupedMedias.Include(bm =>
bm.FileLocations)
                  .Include(bm => bm.BackupedMediaTags)
                  .ThenInclude(bmt => bmt.Tag)
                  .ToListAsync();

              BackupStatistics = new BackupStatistics
              {
                  BackupedFilesCount = db.BackupedMedias.Count(),
                  BackupedBytesCount = db.BackupedMedias.Sum(bm =>
(long)bm.Size)
              };
          }
      }
      finally
      {
          _semaphoreSlim.Release();
      }

      BackupedMediaVmos.AddRange(backupedItems.Select(bi => new
BackupedMediaVmo(bi)));
      IsLoadingCompleted = true;
      OnLoadCompleted?.Invoke(this, EventArgs.Empty);
  }

  public async Task<bool> RemoveAsync(BackupedMediaVmo backupedMediaVmo)
  {
      await _semaphoreSlim.WaitAsync();
      try
      {
          //using (ApplicationContext db = new ApplicationContext(_dbPath))
          {
              var en = db.Remove(backupedMediaVmo.BackupedMedia);
              await db.SaveChangesAsync();
              bool result = BackupedMediaVmos.Remove(backupedMediaVmo);
              BackupStatistics.BackupedFilesCount -= 1;
              BackupStatistics.BackupedBytesCount -=
backupedMediaVmo.BackupedMedia.Size;
              OnMediaDeletedFromDb?.Invoke(this, new
BackupedMediaVmoUpdatedEventArgs(backupedMediaVmo));
```

```
                return en.State == EntityState.Deleted;
            }
        }
        finally
        {
            _semaphoreSlim.Release();
        }
    }

    public async Task<UpdateResult> UpdateAsync(BackupedMediaVmo backupedMediaVmo)
    {
        await _semaphoreSlim.WaitAsync();
        try
        {
            //using (ApplicationContext db = new ApplicationContext(_dbPath))
            {
                var en = db.Update(backupedMediaVmo.BackupedMedia);
                await db.SaveChangesAsync();
                if (en.State == EntityState.Added)
                {
                    return UpdateResult.Added;
                }
                else if (en.State == EntityState.Modified)
                {
                    return UpdateResult.Modified;
                }
                else
                {
                    return UpdateResult.Unknown;
                }
            }
        }
        finally
        {
            _semaphoreSlim.Release();
        }
    }

    public void Dispose()
    {
        if(db != null)
        {
            db.Dispose();
              }
        }

        ~BackupedMediaService()
        {
            Dispose();
        }
    }
```

Клас TelegramClientProvider

```
public class TelegramClientProvider : ITelegramClientProvider
{
    private TgClient _client;
    public TgClient TelegramClient
    {
        get => _client;
```

```
        private set
        {
            if (_client != value)
            {
                _client = value;
                OnClientChanged?.Invoke(this, EventArgs.Empty);
            }
        }
    }

    private Backuper _backuper;
    public Backuper Backuper
    {
        get => _backuper;
        private set
        {
            if (_backuper != value)
            {
                _backuper = value;
                OnBackuperChanged?.Invoke(this, EventArgs.Empty);
            }
        }
    }

    public TLUserFull LoggedUser { get; private set; }
    public event EventHandler OnClientChanged;
    public event EventHandler OnBackuperChanged;

    public void SetTelegramClient(TgClient client)
    {
        TelegramClient = client;
    }

    public void SetBackuper(Backuper backuper)
    {
        Backuper = backuper;
    }

    public void SetLoggedUser(TLUserFull user)
    {
        LoggedUser = user;
    }
}
```

Клас BackupedMediaVmo

```
public class BackupedMediaVmo : INotifyPropertyChanged
{
        const string DEBUG_TAG = "BackupedMediaVmo";

        ILogger logger;
        IBackupedMediaService backupedMediaService;
        string password;

        public BackupedMedia BackupedMedia { get; }

        private LocalMediaVmo _localMediaVmo;
        public LocalMediaVmo LocalMediaVmo
        {
            get => _localMediaVmo;
            set
            {
                _localMediaVmo = value;

                if (_localMediaVmo != null)
                {
                    if (FilePath != _localMediaVmo.FilePath)
                    {
                        FilePath = _localMediaVmo.FilePath;
                    }
                }

                OnPropertyChanged("FilePath");
                OnPropertyChanged("IsFileExists");
                OnPropertyChanged("IsFileNotExists");
                OnPropertyChanged("FileName");
                OnPropertyChanged("MediaSource");
            }
        }

        public event PropertyChangedEventHandler PropertyChanged;

        public BackupedMediaVmo(BackupedMedia backupedMedia)
        {
            BackupedMedia = backupedMedia;
            logger = DependencyService.Get<ILogger>();
```

```
            backupedMediaService =
DependencyService.Get<IBackupedMediaService>();

            if(IsThumbnailExists == false)
            {
                if (IsEncrypted == false)
                {
                    BackupedMediaVmo_OnLoadCompleted(null, null);
                }
            }
        }

        private async void BackupedMediaVmo_OnLoadCompleted(object sender,
EventArgs e)
        {
            DependencyService.Get<ILocalMediaService>().OnLoadCompleted -=
                BackupedMediaVmo_OnLoadCompleted;

            if (!IsThumbnailExists)
            {
                var clientProvider =
DependencyService.Get<ITelegramClientProvider>();

                if (clientProvider.Backuper != null)
                    await UpdateThumbnailAsync();
                else
                    clientProvider.OnBackuperChanged += async delegate { await
UpdateThumbnailAsync(); };
            }
        }

        public void OnPropertyChanged(string prop = "")
        {
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(prop));
        }

        public int Id => BackupedMedia.BackupedMediaId;

        public bool IsLocalMediaConnected => LocalMediaVmo != null;

        public ImageSource MediaSource
        {
            get
            {
```

```
                if(IsFileExists)
                {
                    return ImageSource.FromStream(() => new
MemoryStream(File.ReadAllBytes(FilePath)));
                }
                else
                {
                    return ImageSource.FromFile("media_unavailable.png");
                }
            }
        }

        public string FilePath
        {
            get
            {
                if(IsFileExists)
                {
                    return BackupedMedia.FilePath;
                }
                else if(IsThumbnailExists)
                {
                    return BackupedMedia.ThumbnailPath;
                }
                else if(IsEncrypted)
                {
                    return "encrypted_media.png";
                }
                else
                {
                    return "media_unavailable.png";
                }
            }
            set
            {
                if (BackupedMedia.FilePath != value)
                {
                    BackupedMedia.FilePath = value;
                    OnPropertyChanged("FilePath");
                    OnPropertyChanged("IsFileExists");
                    OnPropertyChanged("IsFileNotExists");
                    OnPropertyChanged("FileName");
                    OnPropertyChanged("MediaSource");
                }
```

```
            }
        }

        public string FileName => Path.GetFileName(BackupedMedia.FilePath);

        public bool IsFileExists => File.Exists(BackupedMedia.FilePath);
        public bool IsFileNotExists => !IsFileExists;

        public ImageSource ThumbnailSource
        {
            get
            {
                if(IsThumbnailExists)
                {
                    return ImageSource.FromStream(() => new
MemoryStream(File.ReadAllBytes(ThumbnailPath)));
                }
                else
                {
                    return ImageSource.FromFile("media_unavailable.png");
                }
            }
        }


        public string ThumbnailPath
        {
            get
            {
                if (IsThumbnailExists)
                {
                    return BackupedMedia.ThumbnailPath;
                }
                else
                {
                    if (IsEncrypted)
                    {
                        return "encrypted_media.png";
                    }
                    else
                    {
                        return "media_unavailable.png";
                    }
                }
```

```
            }
            set
            {
                if (BackupedMedia.ThumbnailPath != value)
                {
                    BackupedMedia.ThumbnailPath = value;
                    OnPropertyChanged("ThumbnailPath");
                    OnPropertyChanged("IsThumbnailExists");
                    OnPropertyChanged("ThumbnailSource");
                    OnPropertyChanged("FilePath");
                }
            }
        }
        public bool IsThumbnailExists =>
File.Exists(BackupedMedia.ThumbnailPath);

        public string Album
        {
            get => BackupedMedia.Album;
            set
            {
                if (BackupedMedia.Album != value)
                {
                    BackupedMedia.Album = value;
                    OnPropertyChanged("Album");
                }
            }
        }

        public string MimeType
        {
            get => BackupedMedia.MimeType;
            set
            {
                if (BackupedMedia.MimeType != value)
                {
                    BackupedMedia.MimeType = value;
                    OnPropertyChanged("MimeType");
                }
            }
        }

        public bool IsVideo => BackupedMedia.IsVideo;
        public bool IsEncrypted => BackupedMedia.IsEncrypted;
```

```
        public string PasswordHash => BackupedMedia.PasswordHash;

        public DateTime CreatedAt
        {
            get => BackupedMedia.CreatedAt;
            set
            {
                if (BackupedMedia.CreatedAt != value)
                {
                    BackupedMedia.CreatedAt = value;
                    OnPropertyChanged("CreatedAt");
                }
            }
        }

        public HashSet<Tag> Tags => new
HashSet<Tag>(BackupedMedia.BackupedMediaTags
            .Select(bmt => bmt.Tag).ToList());


        public async Task AddTagsAsync(IEnumerable<string> tags)
        {
            var backupedMediaService =
DependencyService.Get<IBackupedMediaService>();
            var resolvedTags = await
backupedMediaService.ResolveTagsAsync(tags);
            await backupedMediaService.AddTagsToMediaAsync(this, resolvedTags);
            OnPropertyChanged("Tags");
        }

        public async Task RemoveTagsAsync(IEnumerable<Tag> resolvedTags)
        {
            var backupedMediaService =
DependencyService.Get<IBackupedMediaService>();
            await backupedMediaService.RemoveTagsFromMediaAsync(this,
resolvedTags);
            OnPropertyChanged("Tags");
        }

        public async Task SetTagsAsync(IEnumerable<string> tags)
        {
            var backupedMediaService =
DependencyService.Get<IBackupedMediaService>();
```

```
            var resolvedTags = await
backupedMediaService.ResolveTagsAsync(tags);
            await SetTagsAsync(resolvedTags);
        }

        public async Task SetTagsAsync(IEnumerable<Tag> resolvedTags)
        {
            var backupedMediaService =
DependencyService.Get<IBackupedMediaService>();
            await backupedMediaService.SetTagsForMediaAsync(this, resolvedTags);
            OnPropertyChanged("Tags");
        }

        public async Task<byte[]> DownloadThumbnailAsync(RequestPriority
requestPriority = RequestPriority.High)
        {
            byte[] fileBytes = null;

            var backuper =
DependencyService.Get<ITelegramClientProvider>().Backuper;
            ThumbnailLocation thumbnailLocation =
BackupedMedia.FileLocations.ElementAt(0)?.ThumbnailLocation;
            if (thumbnailLocation != null)
            {
                ThumbnailContext thumbnailContext = thumbnailLocation;
                fileBytes = await
backuper.DownloadThumbnailAsync(thumbnailContext, requestPriority);
            }

            return fileBytes;
        }

        public async Task UpdateThumbnailAsync(string password = null)
        {
            password = password ?? string.Empty;
            logger?.LogWarning($"[{DEBUG_TAG}] Starting loading thumbnail for
{FileName}");

            if(IsEncrypted)
            {
                if(!Cipher.VerifyHash(password, PasswordHash))
                {
                    throw new WrongPasswordException(this);
                }
```

```
            }

            try
            {
                byte[] fileBytes = await DownloadThumbnailAsync();
                if (IsEncrypted)
                {
                    fileBytes = await Cipher.DecryptAsync(fileBytes, password);
                }

                var localMediaService =
DependencyService.Get<ILocalMediaService>();
                string thumbFileName =
localMediaService.GetThumbnailFileName(Id, FileName);
                string thumbnailFilePath = await
localMediaService.SaveThumbnailAsync(fileBytes, thumbFileName);
                ThumbnailPath = thumbnailFilePath;
                await
DependencyService.Get<IBackupedMediaService>().UpdateAsync(this);
                OnPropertyChanged("ThumbnailPath");
                OnPropertyChanged("IsThumbnailExists");
                OnPropertyChanged("ThumbnailSource");

                logger?.LogWarning($"[{DEBUG_TAG}] Thumbnail for {FileName}
downloaded.");
            }
            catch (CryptographicException cex)
            {
                logger.LogError($"[{DEBUG_TAG}] Error while decrypting thumbnail
of {FileName}\n" +
                    $"\tException was: {cex.Message}\n" +
                    $"\tStack trace: {cex.StackTrace}");

                throw new WrongPasswordException(this);
            }
            catch (Exception ex)
            {
                logger?.LogError($"[{DEBUG_TAG}] Error while downloading
thumbnail for {FileName}.\n" +
                    $"\tException was: {ex.Message}\n" +
                    $"\tStack trace: {ex.StackTrace}");
            }
        }
```

```
        public async Task<DownloadTask> PrepareDownload(string password = null)
        {
            if(DownloadTask != null)
            {
                if(DownloadTask.State == TransmissionState.WaitingToStart)
                {
                    return DownloadTask;
                }
                else
                {
                    return null;
                }
            }

            password = password ?? string.Empty;
            if(IsEncrypted)
            {
                if(!Cipher.VerifyHash(password, PasswordHash))
                {
                    throw new WrongPasswordException(this);
                }
                else
                {
                    this.password = password;
                }
            }

            var backuper =
DependencyService.Get<ITelegramClientProvider>().Backuper;
            IEnumerable<FileLocation> fileLocations =
BackupedMedia.FileLocations;
            List<FileContext> fileContexts = new List<FileContext>();
            foreach(var fileLocation in fileLocations)
            {
                var fileRef = await backuper.ResolveFileReference(fileLocation,
false); // implicit cast
                if(fileRef == null)
                {
                    await backupedMediaService.RemoveAsync(this);
                    throw new BackupedFileNotFoundException(this);
                }
                fileContexts.Add(fileRef);
            }
```

```
            var downloadTask = backuper.CreateDownloadTask(fileContexts,
RequestPriority.Low);
            DownloadTask = downloadTask;

            downloadTask.OnError += DownloadTask_OnError;
            downloadTask.OnCompleted += DownloadTask_OnCompleted;

            return downloadTask;
        }

        private async void DownloadTask_OnCompleted(object sender, EventArgs e)
        {
            var downloadResult = (sender as DownloadTask).Result;
            var fileData = downloadResult.FileData;

            if(IsEncrypted)
            {
                try
                {
                    fileData = await Cipher.DecryptAsync(fileData,
this.password);
                }
                catch (CryptographicException cex)
                {
                    logger.LogError($"[{DEBUG_TAG}] Error while decrypting file
{FileName}\n" +
                        $"\tException was: {cex.Message}\n" +
                        $"\tStack trace: {cex.StackTrace}");
                    // for UI report
                    fileData = new[] { BackupedMedia.DECRYPTION_ERROR_CODE };
                    // throw new WrongPasswordException(this);
                }
            }

            var localMediaService = DependencyService.Get<ILocalMediaService>();
            var savedMedia = await localMediaService.SaveMediaAsync(fileData,
this);
            if (savedMedia == null)
            {
                LocalMediaVmo = null;
            }

            CancelDownloading();
        }
```

```
        private void DownloadTask_OnError(object sender,
Base.Events.RequestErrorEventArgs e)
        {
            CancelDownloading();
        }

        public void CancelDownloading()
        {
            if(DownloadTask != null)
            {
                DownloadTask.Cancel();
                DownloadTask.OnError -= DownloadTask_OnError;
                DownloadTask.OnCompleted -= DownloadTask_OnCompleted;
                DownloadTask = null;
            }
        }

        private DownloadTask _downloadTask;
        public DownloadTask DownloadTask
        {
            get => _downloadTask;
            private set
            {
                if (_downloadTask != value)
                {
                    _downloadTask = value;
                    OnPropertyChanged("DownloadTask");
                    OnPropertyChanged("IsDownloading");
                    OnPropertyChanged("IsNotDownloading");
                    OnPropertyChanged("IsDownloadPossibleNow");
                }
            }
        }

        public bool IsDownloading => DownloadTask != null;
        public bool IsNotDownloading => !IsDownloading;
        public bool IsDownloadPossibleNow => IsNotDownloading;

        private string _mainButtonText = "Завантажити";
        public string MainButtonText
        {
            get => _mainButtonText;
            set
```

```
            {
                if (_mainButtonText != value)
                {
                    _mainButtonText = value;
                    OnPropertyChanged("MainButtonText");
                }
            }
        }

        public async Task DeleteFromCloudAsync(string password = null)
        {
            password = password ?? string.Empty;
            var backuper =
DependencyService.Get<ITelegramClientProvider>().Backuper;
            var backupedMediaService =
DependencyService.Get<IBackupedMediaService>();

            if(IsEncrypted)
            {
                if(!Cipher.VerifyHash(password, PasswordHash))
                {
                    throw new WrongPasswordException(this);
                }
            }

            // deleting thumbnail
            var thumbnailLocation =
BackupedMedia.FileLocations.FirstOrDefault()?.ThumbnailLocation;
            if(thumbnailLocation != null)
            {
                await backuper.Delete(thumbnailLocation); // implicit cast
            }

            // deleting file (file parts)
            foreach (var fileLocation in BackupedMedia.FileLocations)
            {
                await backuper.Delete(fileLocation); // implicit cast
            }

            // updating database
            await backupedMediaService.RemoveAsync(this);
            if(LocalMediaVmo != null)
            {
                LocalMediaVmo.BackupedMediaVmo = null;
```

```
            }
        }

        public string CreatedAtKey
        {
            get
            {
                if (CreatedAt.Date == DateTime.Now.Date)
                {
                    return "Сьогодні";
                }

                if (CreatedAt.Date == DateTime.Now.Date - TimeSpan.FromDays(1))
                {
                    return "Вчора";
                }

                if (CreatedAt.Date == DateTime.Now.Date - TimeSpan.FromDays(2))
                {
                    return "Позавчора";
                }

                return CreatedAt.ToString("D");
            }
        }

        public override int GetHashCode()
        {
            return BackupedMedia.GetHashCode();
        }

        public override bool Equals(object obj)
        {
            if (obj is BackupedMediaVmo bmv)
            {
                return bmv.BackupedMedia == BackupedMedia;
            }

            return false;
        }
    }
```

Клас LocalMediaVmo

```
public class LocalItemsVmo : INotifyPropertyChanged
{
        const string DEBUG_TAG = "LocalItemsVmo";

        ILocalMediaService localMediaService;
        ILogger logger;
        ImageSource albumImg;
        ImageSource dateImg;
        Task loadTask;

        ImageSource _groupingIcon;
        public ImageSource GroupingIcon
        {
            get => _groupingIcon;
            set
            {
                if (_groupingIcon != value)
                {
                    _groupingIcon = value;
                    OnPropertyChanged("GroupingIcon");
                }
            }
        }
        public string Title => "Локальні медіа";
        public bool IsLoadingCompleted => localMediaService.IsLoadingCompleted;
        public ICommand ItemTappedCommand { get; set; }

        private int _columnsCount = 4;
        public int ColumnsCount
        {
            get => _columnsCount;
            set
            {
                if (_columnsCount != value)
                {
                    _columnsCount = value;
                    OnPropertyChanged("ColumnsCount");
                }
            }
        }
```

```
        private bool _isMainLayoutVisible = false;
        public bool IsMainLayoutVisible
        {
            get => _isMainLayoutVisible;
            set
            {
                if (_isMainLayoutVisible != value)
                {
                    _isMainLayoutVisible = value;
                    OnPropertyChanged("IsMainLayoutVisible");
                }
            }
        }

        private bool _isEmptyViewVisible = true;
        public bool IsEmptyViewVisible
        {
            get => _isEmptyViewVisible;
            set
            {
                if (_isEmptyViewVisible != value)
                {
                    _isEmptyViewVisible = value;
                    OnPropertyChanged("IsEmptyViewVisible");
                }
            }
        }

        private string _emptyViewText = "Завантаження...";
        public string EmptyViewText
        {
            get => _emptyViewText;
            set
            {
                _emptyViewText = value;
                OnPropertyChanged("EmptyViewText");
            }
        }

        private List<LocalMediaVmo> localMediaInfoViewModels;

        private ObservableCollection<Grouping<string, LocalMediaVmo>>
_localMedias_ByDate;
```

```
        private ObservableCollection<Grouping<string, LocalMediaVmo>>
_localMedias_ByAlbum;

        private ObservableCollection<Grouping<string, LocalMediaVmo>>
_localMedias;
        public ObservableCollection<Grouping<string, LocalMediaVmo>> LocalMedias
        {
            get => _localMedias;
            set
            {
                if (value != _localMedias)
                {
                    _localMedias = value;
                    OnPropertyChanged("LocalMedias");
                }
            }
        }

        private GroupBy _groupBy = GroupBy.Date;
        public GroupBy LocalGroupBy
        {
            get => _groupBy;
            set
            {
                if (_groupBy != value)
                {
                    if (value == GroupBy.Album)
                    {
                        LocalMedias = _localMedias_ByAlbum;
                        _groupBy = value;
                        GroupingIcon = albumImg;
                    }

                    if (value == GroupBy.Date)
                    {
                        LocalMedias = _localMedias_ByDate;
                        _groupBy = value;
                        GroupingIcon = dateImg;
                    }
                }
            }
        }
```

```
        public event PropertyChangedEventHandler PropertyChanged;

        public LocalItemsVmo(int columnsCount)
        {
            localMediaService = DependencyService.Get<ILocalMediaService>();
            logger = DependencyService.Get<ILogger>();

            ColumnsCount = columnsCount;

            albumImg = ImageSource.FromFile("gallery2.png");
            dateImg = ImageSource.FromFile("calendar.png");

            GroupingIcon = dateImg;

            ItemTappedCommand = new Command<LocalMediaVmo>(OnItemTapped);
            LocalMedias = new ObservableCollection<Grouping<string,
LocalMediaVmo>>();
            BindingBase.EnableCollectionSynchronization(LocalMedias, null,
ObservableCollectionCallback);

            localMediaService.OnLocalMediaLoaded +=
LocalMediaService_OnLocalMediaLoaded;
            localMediaService.OnMediaSaved +=
LocalMediaService_OnMediaDownloaded;
            localMediaService.OnLocalMediaDeleted +=
LocalMediaService_OnLocalMediaDeleted;

            LoadLocalMedias();
        }

        public List<object> GetCurrentGroups()
        {
            List<object> groups = new List<object>();

            foreach(var group in LocalMedias)
            {
                groups.Add(group);
            }

            return groups;
        }

        private void LocalMediaService_OnLocalMediaDeleted(object sender,
LocalMediaVmoUpdatedEventArgs e)
```

```
        {
            //LoadLocalMedias();

            LocalMediaVmo localMediaVmo = e.LocalMediaVmo;
            if (localMediaVmo != null)
            {
                RemoveMediaFromGroups(_localMedias_ByAlbum, localMediaVmo);
                RemoveMediaFromGroups(_localMedias_ByDate, localMediaVmo);
            }
            OnPropertyChanged("LocalMedias");
        }

        private void RemoveMediaFromGroups(ObservableCollection<Grouping<string,
LocalMediaVmo>> groups,
            LocalMediaVmo localMediaVmo)
        {
            foreach(var group in groups)
            {
                if (group.Remove(localMediaVmo))
                {
                    if(group.Count == 0)
                    {
                        groups.Remove(group);
                    }

                    return;
                }
            }
        }

        private void LocalMediaService_OnMediaDownloaded(object sender,
LocalMediaVmoUpdatedEventArgs e)
        {
            //LoadLocalMedias();

            LocalMediaVmo localMediaVmo = e.LocalMediaVmo;
            if (localMediaVmo != null)
            {
                AddMediaToGroup(_localMedias_ByAlbum, localMediaVmo,
App.DOWNLOAD_FOLDER_NAME);
                AddMediaToGroup(_localMedias_ByDate, localMediaVmo, "Сьогодні");
            }
        }
```

```
        private void AddMediaToGroup(ObservableCollection<Grouping<string,
LocalMediaVmo>> groups,
            LocalMediaVmo localMediaVmo, string key)
        {
            var entry = groups.FirstOrDefault(g => g.Name == key);
            if(entry != null)
            {
                entry.Add(localMediaVmo);
            }
            else
            {
                var group = new Grouping<string, LocalMediaVmo>(key,
groups.First().Icon,
                    new[] { localMediaVmo });

                if(groups.Count == 0)
                {
                    groups.Add(group);
                }
                else
                {
                    groups.Insert(0, group);
                }
            }
        }

        private async void LocalMediaService_OnLocalMediaLoaded(object sender,
LocalMediaVmoUpdatedEventArgs e)
        {
            await Device.InvokeOnMainThreadAsync(() =>
            {
                EmptyViewText = $"Завантаження... {e.CurrentIndex}%";
            });
        }

        public void ChangeGrouping(GroupBy localItemsGroupBy)
        {
            if(IsLoadingCompleted)
                LocalGroupBy = localItemsGroupBy;
        }

        async void LoadLocalMedias()
        {
            var stopWatch = new Stopwatch();
```

```
            logger?.LogInformation($"[{DEBUG_TAG}] Starting loading local
files.");
            stopWatch.Start();

            loadTask =
localMediaService.StartLoading(App.AppMainCancellationToken);
            await loadTask;
            GroupItems();

            stopWatch.Stop();
            var ts = stopWatch.Elapsed;
            string elapsedTime = String.Format("{0:00}:{1:00}.{2:00}",
                ts.Minutes, ts.Seconds,
                ts.Milliseconds / 10);

            logger?.LogInformation($"[{DEBUG_TAG}] Local files loading completed
in {elapsedTime}");

            if (localMediaService.LocalMediaVmos.Count == 0)
            {
                IsEmptyViewVisible = true;
                EmptyViewText = "Медіа файлів не знайдено";
            }
            else
            {
                IsEmptyViewVisible = false;
                IsMainLayoutVisible = true;
            }
        }

        private void GroupItems()
        {
            var res = localMediaService.LocalMediaVmos;

            lock(_localMedias)
            {
                localMediaInfoViewModels = res;

                var bms = DependencyService.Get<IBackupedMediaService>();
                if (bms.IsLoadingCompleted)
                {
                    MarkLocalBackupedItems();
                }
                else
```

```
                {
                    bms.OnLoadCompleted += delegate { MarkLocalBackupedItems();
};
                }

                var groupedByDate = localMediaInfoViewModels.GroupBy(g =>
g.CreatedAtKey);
                var groupingByDate = groupedByDate.Select(g => new
Grouping<string, LocalMediaVmo>(g.Key, dateImg, g));
                _localMedias_ByDate = new ObservableCollection<Grouping<string,
LocalMediaVmo>>(groupingByDate);
                LocalMedias = _localMedias_ByDate;

                var groupedByAlbum = localMediaInfoViewModels.GroupBy(g =>
g.Album);
                var groupingByAlbum = groupedByAlbum.Select(g => new
Grouping<string, LocalMediaVmo>(g.Key, albumImg, g));
                _localMedias_ByAlbum = new ObservableCollection<Grouping<string,
LocalMediaVmo>>(groupingByAlbum);
            }
        }

        private void MarkLocalBackupedItems()
        {
            var bms = DependencyService.Get<IBackupedMediaService>();
            foreach(var bi in bms.BackupedMediaVmos)
            {
                var entry = localMediaInfoViewModels.FirstOrDefault(lm =>
lm.LocalMediaInfo.Path.GetHashCode() == bi.BackupedMedia.FilePath.GetHashCode()
&&
                    lm.LocalMediaInfo.Path == bi.BackupedMedia.FilePath);
                if(entry != null)
                {
                    entry.BackupedMediaVmo = bi;
                    bi.LocalMediaVmo = entry;
                }
            }
        }

        private void ObservableCollectionCallback(IEnumerable collection, object
context,
            Action accessMethod, bool writeAccess)
        {
            lock (collection)
```

```
            {
                accessMethod?.Invoke();
            }
        }

        private async void OnItemTapped(LocalMediaVmo mediaInfo)
        {
            logger?.LogInformation($"[{DEBUG_TAG}] Opening detailed page for
{mediaInfo.Name}");
            await Application.Current.MainPage.Navigation.PushModalAsync(new
MediaDetailPage(mediaInfo));
        }

        public void OnPropertyChanged(string prop = "")
        {
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(prop));
        }
}
```

ДОДАТОК Г
(обов’язковий)

ПРЕЗЕНТАЦІЙНІ МАТЕРІАЛИ

Хмельницький Національний Університет
Факультет програмування та комп'ютерних
і телекомунікаційних систем
Кафедра інженерії програмного забезпечення

Дипломний проект на тему:

«Мобільний додаток для організації медіа-файлів та їх автоматичного резервного копіювання у хмару з використанням API Telegram»

Студент: Капітанець Степан Олександрович
Керівник: Радельчук Г. І., кандидат технічних наук, доцент

Мета та завдання проекту

Мета проекту - розробка мобільної галереї для ОС Android, яка дозволятиме її користувачам в автоматичному та ручному режимі вивантажувати медіа-файли в хмарне сховище Telegram, а також організовувати свою галерею шляхом додавання до них тегів, за якими потім можна буде виконувати пошук.

Завдання, які необхідно вирішити для досягнення мети:

1) провести аналіз предметної області для визначення її головних особливостей;
2) провести аналіз наявного програмного забезпечення;
3) сформувати технічне завдання;
4) спроектувати програмний продукт;
5) ознайомитися та провести аналіз Telegram API для визначення методів, які будуть використовуватись у розроблюваному мобільному додатку;
6) виконати програмну реалізацію проекту за допомогою фреймворку Xamarin;
7) провести тестові випробування додатку;
8) опублікувати додаток на платформі Google Play Market.

Актуальність теми

Актуальність теми полягає в тому, що більшість з існуючих рішень для резервного копіювання з використанням хмарного сховища не зручні або повністю не придатні для роботи із мобільною галереєю.

Окрім того, мало який подібний сервіс може похвалитись привабливою ціновою політикою, в той час як Google Photos (найпопулярніший та найзручніший на даний момент додаток для резервного копіювання медіа-файлів) починаючи із 1 червня 2021 року прибирає безкоштовний тариф, який надавав користувачам можливість вивантаження у хмару безлімітної кількості медіа-файлів.

Ці два фактори разом вже найближчим часом спричинять «голод» на дешеві та надійні альтернативи в даному сегменті мобільних додатків.

Перейдемо до огляду наявного програмного забезпечення.

Google Photos

В плані зовнішнього вигляду Google Photos відрізняється з-поміж інших додатків простим та приємним інтуїтивно зрозумілим дизайном. У ньому досить легко орієнтуватися, а більшість дій можна виконати всього за декілька простих кроків.

Із ключових можливостей додатку варто відзначити наступні:

- можливість зручного перегляду всієї колекції медіа-файлів, що дозволяє використовувати додаток замість стандартної галереї;
- автоматичне резервне копіювання фото та відео у хмару;
- розпізнавання об'єктів на фото, що значно спрощує їх пошук;
- можливість редагувати світлини прямо в додатку;
- можливість групування об'єктів у галереї за часом або за альбомом, до якого вони належать.

Dropbox

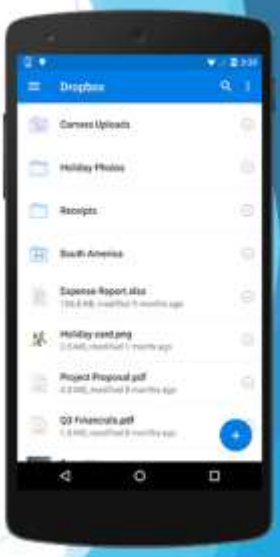
До ключових особливостей додатку можна віднести:

- автоматичне резервне копіювання;

- наявність окремого сховища для найбільш конфіденційних документів;

- можливість зберігати паролі.

До мінусів даного додатку відноситься об'єм безкоштовного дискового простору - всього 2 ГБ. Також до них можна віднести систему пошуку у безкоштовному тарифному плані, яка дозволяє шукати файли лише по їх назві. Перехід на професійний план дозволить шукати серед об'єктів, які були розпізнані на медіа-файлах автоматично. Додавати свої власні теги до файлів не можна, а тому досягти чогось подібного користуючись безкоштовним тарифним планом неможливо.



Mega

Mega - це хмарне сховище, яке відразу підкуповує своїми умовами любителів об'ємних дисків. Навіть безкоштовний тарифний план може похвалитися чималим обсягом дискового простору - цілих 50 ГБ, а платні тарифи, в перерахунку на вартість за кожен гігабайт, дешевші ніж у конкурентів. Тим не менш, за подібні об'єми частково доводиться платити зручністю користування.

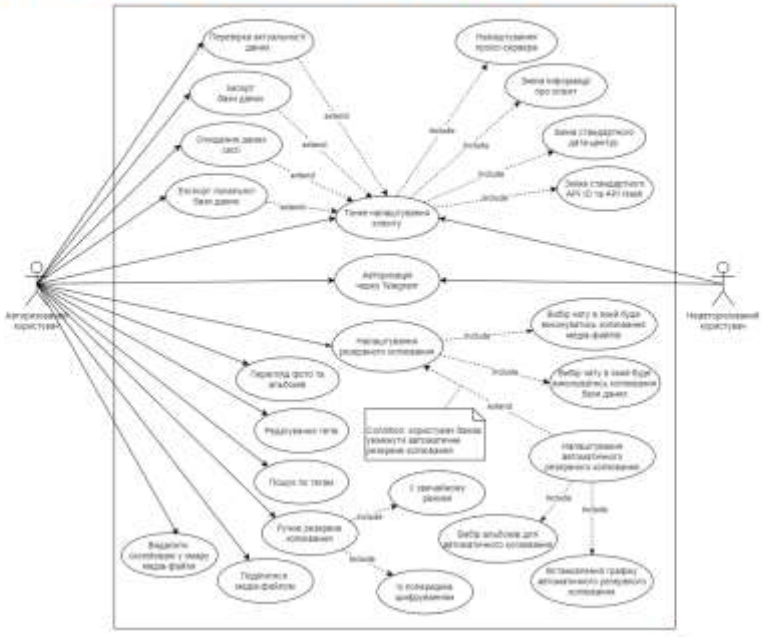
На Mega існують квоти на передачу даних. Окрім того, Mega не зможе замінити для користувача стандартну галерею (оскільки ця програма не призначена для роботи з медіа і переглядати та орієнтуватися серед фото та відеофайлів не зручно), а можливість якимось чином маркувати файли (щоби потім їх можна було простіше знайти) відсутня.

Результати аналізу подібного ПЗ

Проведений аналіз подібного програмного забезпечення показав, що розроблювана програмна система повинна:

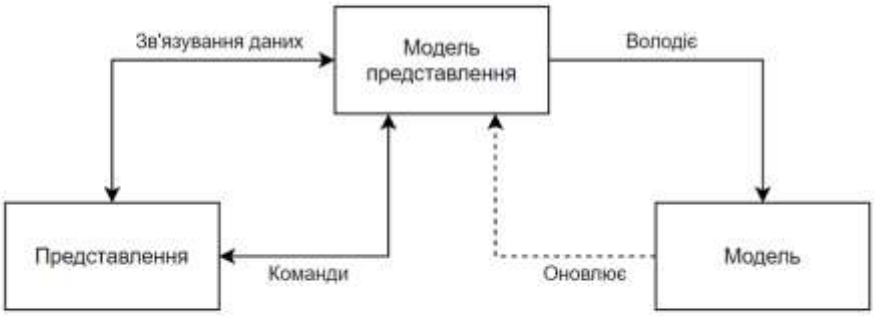
- мати щедрий об'єм безкоштовного дискового простору (Telegram не накладає жодних обмежень на кількість файлів, які користувач може вивантажити у їхнє хмарне сховище);
- бути взаємозамінною із стандартним додатком галереї;
- мати можливість організовувати медіа-файли для того, щоби серед них можна було простіше орієнтуватися та швидше шукати;
- мати функцію автоматичного резервного копіювання із можливістю ручного налаштовувати того, з яких саме альбомів слід синхронізувати медіа-файли;
- мати простий та інтуїтивно зрозумілий дизайн інтерфейсу;
- бути максимально надійною.

Діаграма варіантів використання



Архітектура додатка

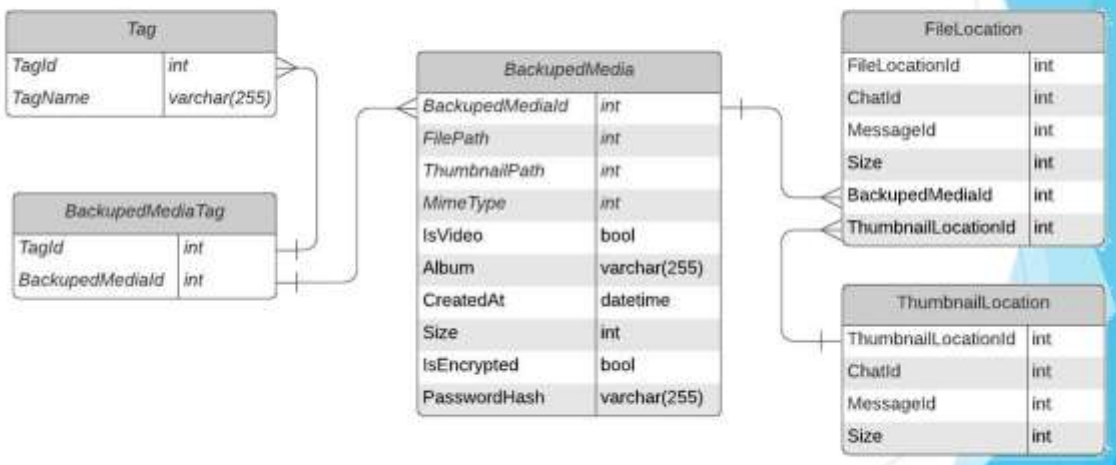
Для реалізації мобільного додатку було обрано архітектурний шаблон MVVM.



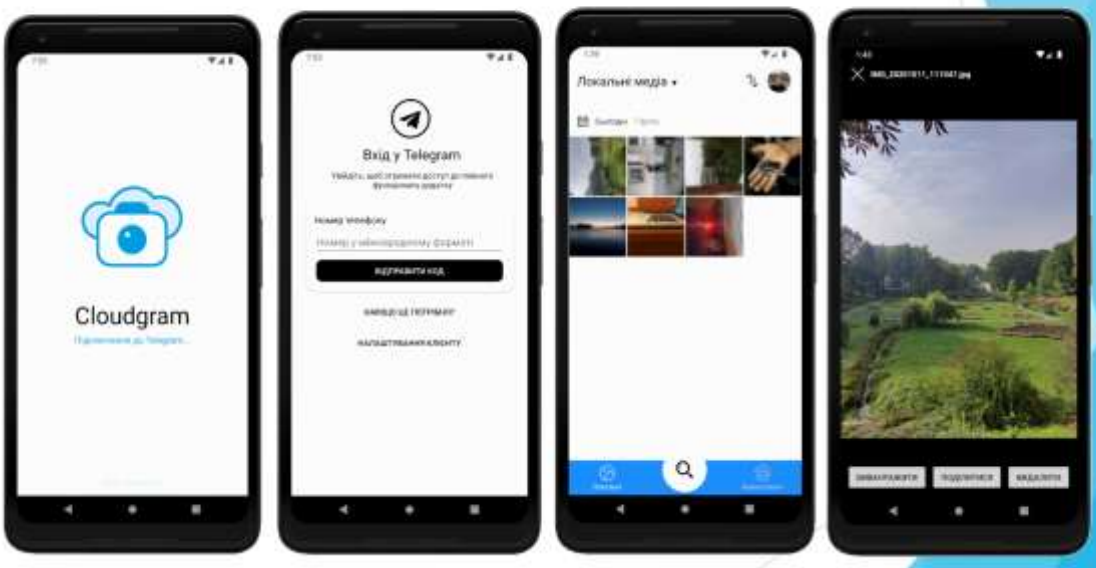
Модулі додатка

В результаті попереднього аналізу функціональної структури додатка було визначено 5 основних рівнів (модулів), з яких складатиметься додаток Cloudgram:

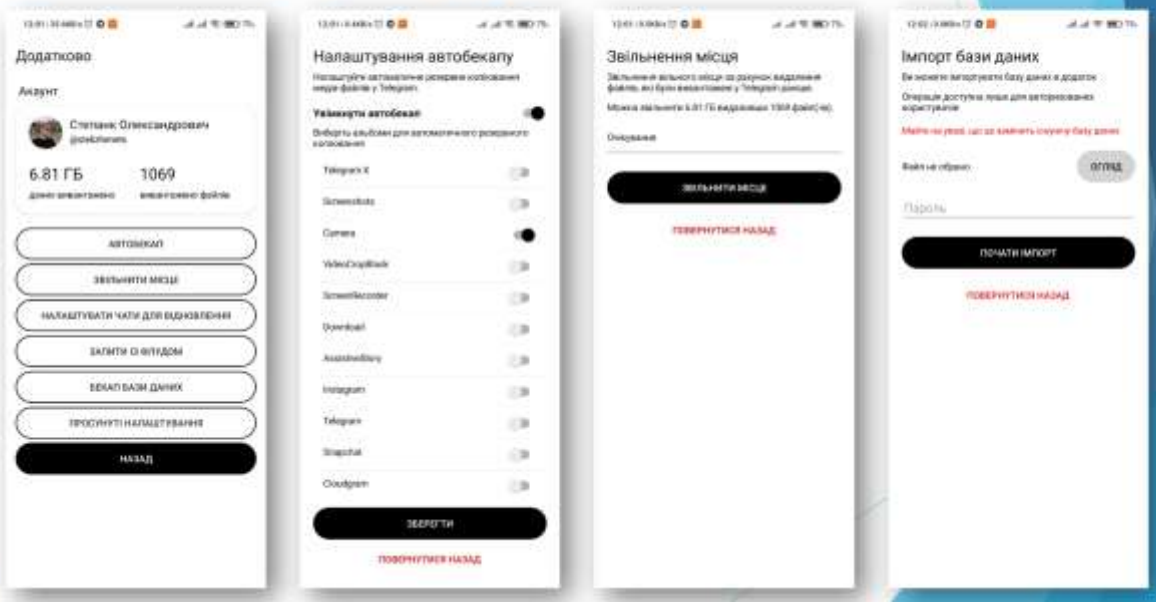
Логічна модель бази даних



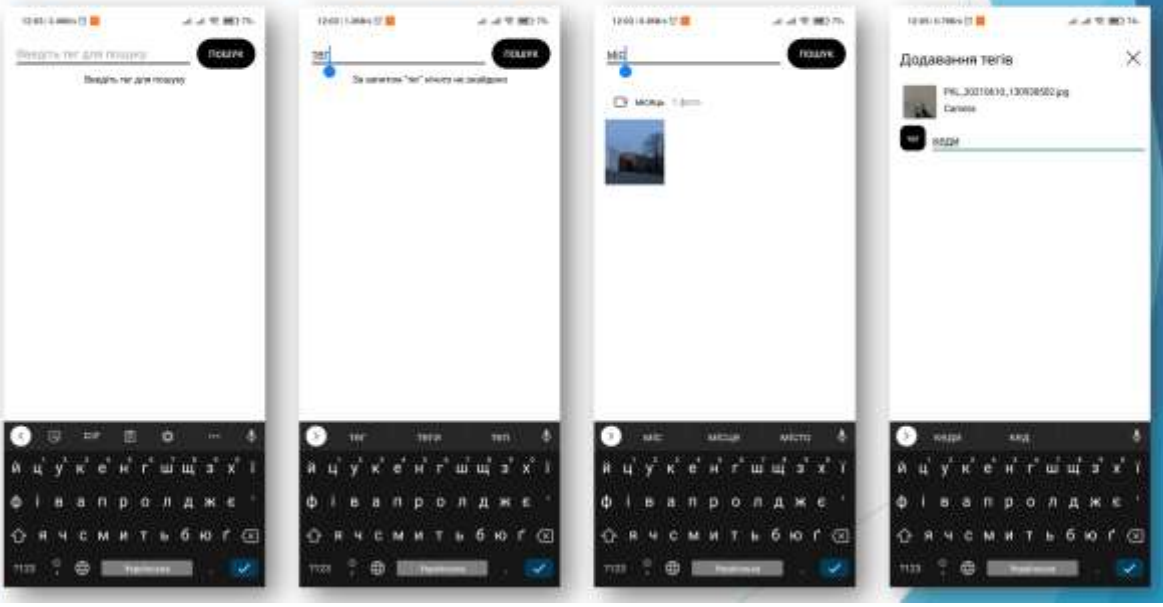
Дизайн інтерфейсу додатка



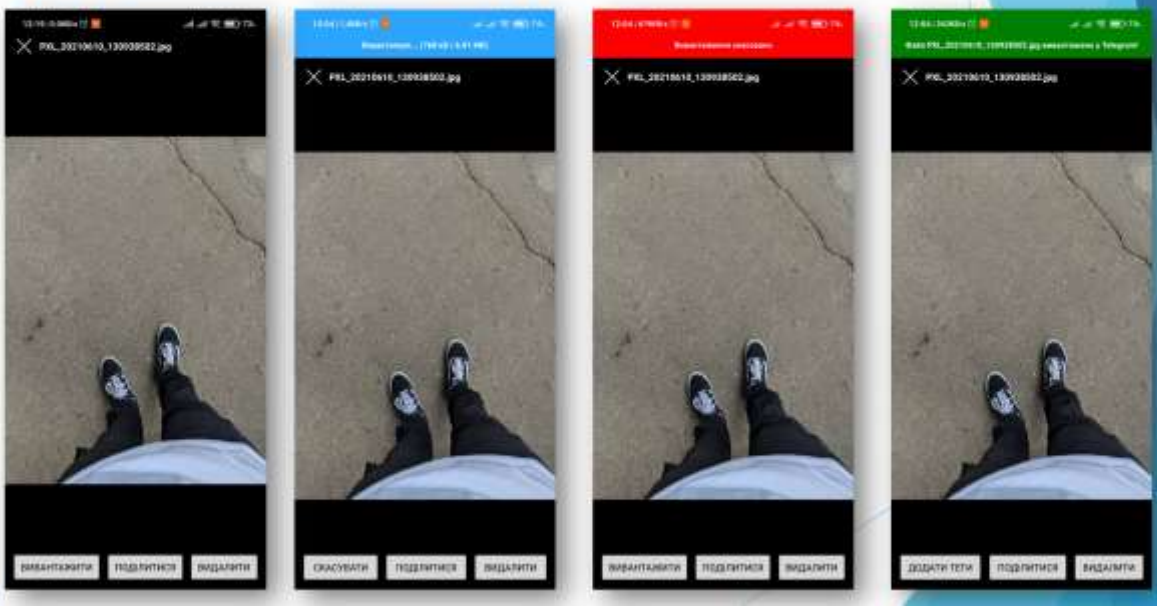
Дизайн інтерфейсу додатка



Дизайн інтерфейсу додатка



Дизайн інтерфейсу додатка



Дизайн інтерфейсу додатка

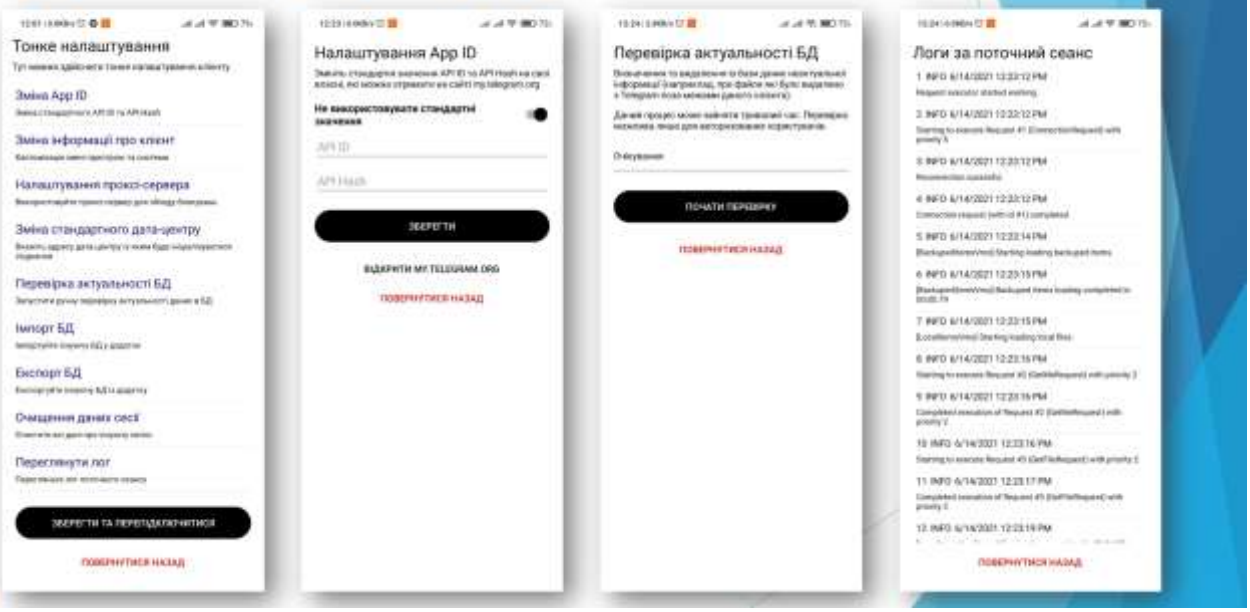
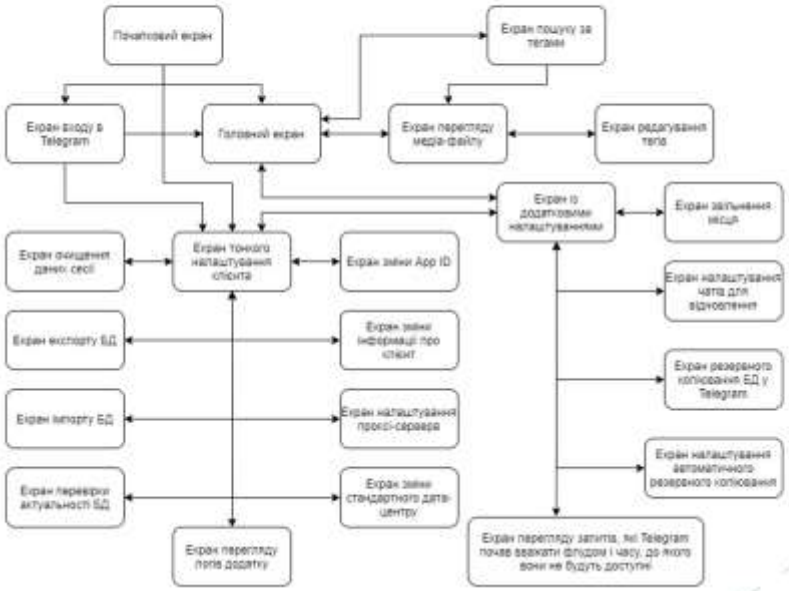


Схема переходів між екранами додатка



Висновки

В результаті виконання дипломного проекту була проаналізована предметна область, з'ясовані причини такого частого виникнення проблеми нестачі пам'яті у користувачів смартфонів, та були досліджені наявні програмні засоби для резервного копіювання. Окрім того, було визначено їх сильні та слабкі сторони, а також доведено потребу в розробці спеціалізованого програмного продукту, який призначатиметься саме для роботи із медіа-файлами. Сформовані вимоги до програмного забезпечення були описані за допомогою діаграми варіантів використання, та формувалися опираючись на інші успішні додатки. Також було проведено аналіз переваг та недоліків найпопулярніших архітектур мобільних додатків, і встановлено, що для розроблюваної системи найбільше підходить шаблон MVVM.

Окрім того, було проведено аналіз Telegram API, що дало можливість визначити які саме дані необхідно буде зберігати у базі даних для досягнення цілей, поставлених у завданні до дипломного проектування. Зібрана інформація допомогла спроектувати діаграму «сутність-зв'язок» на якій було відображено схему БД розроблюваного додатка. У результаті детального аналізу було визначено та описано основні модулі додатка і те, як вони взаємодіють між собою. Отриману інформацію було продемонстровано за допомогою діаграми пакетів.

Окрім вищезгаданого, було також виконано проектування основних елементів користувацького інтерфейсу та проведено їх юзабіліті дослідження. Список всіх екранів, а також способи переходів між ними, було продемонстровано за допомогою схеми переходів.

В результаті виконання дипломного проекту було розроблено та протестовано мобільний додаток який дозволятиме його користувачам в автоматичному та ручному режимі вивантажувати медіа-файли в хмарне сховище Telegram, а також організовувати свою галерею шляхом додавання до них тегів, за якими потім можна буде виконувати пошук.

Завідувачу кафедри інженерії програмного забезпечення проф. Бедратюку Л. П.

здобувача вищої освіти
Капітанця С. О.
Прізвище, ініціали

факультет ПКТС, 4 курс, група ІПЗ-17-1

ЗАЯВА

З правилами чинного Положення «Про дотримання академічної доброчесності в Хмельницькому національному університеті» від 26.09.2020 (зі змінами від 26.11.2020), згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування заходів дисциплінарної та академічної відповідальності, ознайомлений (а). Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на плагіат оповіщений (а) та надаю свою згоду на обробку та збереження університетом моєї роботи в інституційному репозитарії університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-технічних засобів (Unicheck та Anti-Plagiarism) та використання роботи для виявлення плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення плагіату в текстах робіт.

Робота для перевірки університетом надається в друкованому та електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

01.06
дата

підпис

Mon Jun 14 09:51:03 EEST 2021, Хіврич Володимир Русланович, Хмельницький національний університет, ХНУ

Anti-Plagiarism v-15.257

Максимальне співпадіння з одним документом 20.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 18%**

ID: 93543 Назва: Мобільний додаток для організації медіа-файлів та їх автоматичного резервного копіювання у хмару з використанням API Telegram Додано в БД: 2021-06-14 Автора: С. О. Капітанець Керівники: Г. І. Радельчук Консультанти: Опоненти:	Документ		Сумарний збіг по Базі Даних	
	Символи	Лексеми	Символи	Лексеми
	156459	1410	35485 (23%)	318 (23%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми
90209	Назва: Звіт з передипломної практики Капітанець С.О. Додано в БД: 2021-05-11 Автора: Капітанець С.О. Керівники: Радельчук Г. І. Консультанти: Опоненти:	31756 (20.0%)	277 (20.0%)

UNICHECK

Ім'я користувача:
Кафедра ІПЗ

ID перевірки:
1008287939

Дата перевірки:
14.06.2021 10:48:46 EEST

Тип перевірки:
Doc vs Internet + Library

Дата звіту:
14.06.2021 10:51:50 EEST

ID користувача:
100005589

Назва документа: **Дипломна Капітанця без додатків**

Кількість сторінок: **105** Кількість слів: **21681** Кількість символів: **173592** Розмір файлу: **3.20 MB** ID файлу: **1008357188**

Виявлено модифікації тексту (можуть впливати на відсоток схожості)

5.02%
Схожість

Найбільша схожість: **2.76%** з джерелом з Бібліотеки (ID файлу: **1008265198**)

2.41% Джерела з Інтернету	405	Сторінка 107
3.73% Джерела з Бібліотеки	91	Сторінка 110

0% Цитат

Вилучення цитат вимкнене

Вилучення списку бібліографічних посилань вимкнене

0%
Вилучень

Немає вилучених джерел

Модифікації

Виявлено модифікації тексту. Детальна інформація доступна в онлайн-звіті.

Замінені символи	1
Підозріле форматування	24 сторінки

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА ДИПЛОМНИЙ ПРОЕКТ
освітнього ступеня «Бакалавр»

Дипломник Капітанець Степан Олександрович

Тема Мобільний додаток для організації медіа-файлів та їх автоматичного резервного копіювання у хмару з використанням API Telegram

Спеціальність 121 – Інженерія програмного забезпечення

Обсяг дипломного проекту:

Кількість листів креслень _____; кількість сторінок записки 165

1. Короткий зміст пояснювальної записки та прийнятих рішень У дипломній роботі проведено аналіз предметної області, з'ясовані причини частого виникнення проблеми нестачі пам'яті у користувачів смартфонів, проведено аналіз схожого програмного забезпечення. На основі отриманих даних були сформовані функціональні вимоги до дипломного проекту, були порівняні архітектурні стилі для розробки мобільних додатків, визначені модулі системи та здійснена їх програмна реалізація. Було проведено тестування додатку та його реліз на платформі Google Play Market.

2. Висновок про відповідність проекту поставленому завданню Дипломна робота освітнього ступеня «бакалавр» у повній мірі відповідає поставленому завданню, як у теоретичній, так і в практичній її частині.

3. Характеристика виконання кожного розділу проекту, ступінь використання останніх досягнень науки і техніки та передових методів роботи У вступі описується актуальність дипломної роботи. У першому розділі аналізується предметна область, досліджуються проблеми та способи їх вирішення, описуються вимоги до програмної системи. У другому розділі проводиться аналіз архітектурних підходів до розробки програмного забезпечення, обґрунтовується вибір шаблону проектування, проводиться декомпозиція проекту, де виконується проектування бази даних і опис інтерфейсів користувача, проводиться аналіз способів реалізації та вибір технологій для розробки. У третьому розділі наведені ключові етапи реалізації програмної системи, що у повній мірі відповідають розробленим архітектурним рішенням. У четвертому розділі було виконано модульне тестування додатка, в результаті чого було підтверджено його коректну роботу.

4. Позитивні сторони проекту Розроблений додаток має простий та інтуїтивно зрозумілий дизайн інтерфейсу, а тематика дипломного проекту актуальна і визначає та вирішує існуючі проблеми і незручності.

5. Негативні сторони проекту Для розробки додатку використовувався кросплатформний фреймворк, але в результаті була розроблена лише версія додатку для ОС Android.

6. Оцінка графічного оформлення та пояснювальної записки проекту Графічне оформлення виконане відповідно до теми дипломної роботи. Пояснювальна записка оформлена згідно вимог чинних стандартів.

7. Відгук про дипломний проект в цілому Дипломна робота заслуговує позитивної оцінки. Увесь матеріал дипломної роботи структурований, чіткий, послідовний та у повній мірі розкриває обрану тему, що дозволяє чітко розуміти викладений матеріал у рамках тематики дипломної роботи. Графічні матеріали дозволяють чітко побачити доцільність та ефективність рішень, які були прийняті за основу для вирішення поставленої задачі.

8. Інші зауваження

9. Оцінка дипломного проекту Розглянувши позитивні та негативні сторони представленої дипломної роботи, можна зробити висновок, що вона заслуговує на оцінку «відмінно» (4,75/A).

РЕЦЕНЗЕНТ (прізвище, ім'я, по-батькові, посада, місце роботи) Мартинюк Валерій Володимирович, доктор технічних наук, професор, зав. кафедри автоматизації, комп'ютерно-інтегрованих технологій і телекомунікацій (АКІТ і ТК)

"09" червня 2021 р. _____
(підпис)

РІШЕННЯ ЕКСПЕРНОЇ КОМІСІЇ
КАФЕДРИ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ
ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатом звіту подібності щодо роботи, генерованого системою виявлення текстових збігів/ідентичності/схожості:

Назва: «Мобільний додаток для організації медіа-файлів та їх автоматичного резервного копіювання у хмару з використанням API Telegram»
Автор: Капітанець Степан Олександрович
Спеціальність: 121 – Інженерія програмного забезпечення
Освітня програма: Освітньо-професійна програма «Інженерія програмного забезпечення»
Науковий керівник: Радельчук Галина Іванівна, кандидат технічних наук, доцент
Після аналізу звіту подібності зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі, є законними і не є плагіатом. Робота приймається до захисту.	відповідає
2	Виявлені запозичення не є плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота приймається до захисту, але має бути відкоригована. Відкоригований варіант має бути поданий на кафедру за 2 дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої та електронної версії роботи	
3	Виявлені запозичення не є плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнені. Робота може бути допущена до захисту (наступного року) після того як буде відкоригована та допрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
5	Інше:	

Підтвердження:

Запозичення, виявлені в роботі, є законними і не є плагіатом, оскільки:

1) у тексті дипломного проекту системами перевірки на плагіат виявлено схожість з деякими документами в частині загальновживаних обов'язкових словосполучень у стандартних бланках (титулка, бланк завдання, відомість документів), у структурі змісту, назвах розділів/підрозділів тощо та в назвах публікацій у переліку джерел посилання;

2) в якості запозичень системою було зафіксовано деякі послідовності вихідного коду і посиланання на бібліотеки, які є стандартними мовними конструкціями програмування та не можуть розглядатися як об'єкт авторських прав і, відповідно, їх порушення;

3) усі запозичення є фрагментарними або мають належним чином оформленні посилання;

4) виявлені модифікації тексту не впливають на відсоток схожості.

Сумарний обсяг всіх запозичень, визначений системою виявлення збігів ідентичності/ схожості, складає 5,02% і адресується до 405 джерел, що, з урахуванням наведених обгрунтувань, відповідає характеру теми і свідчить на користь дипломного проекту.

Керівник  Г. І. Радельчук

Гарант ОП  Л. П. Бедратюк

Завідувач кафедри  Л. П. Бедратюк