Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедра Програмного забезпечення комп'ютерних систем
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня

магістра
(назва освітньо-кваліфікаційного рівня)

студента	*Петрушенка Артема В'ячеславовича*
	(ПІБ)
академічної групи	*121м-22-3*
	(шифр)
спеціальності	*121 Інженерія програмного забезпечення*
	(код і назва спеціальності)
освітньої програми	*«Інженерія програмного забезпечення»*
	(назва освітньої програми)
на тему:	*Дослідження продуктивності мобільних додатків в залежності від різних методів керуванням з застосуванням фреймворку Flutter*

_____ *Петрушенко Артем В'ячеславович*

Керівники	**Прізвище, ініціали**	**Оцінка за шкалою**		**Підпис**
		рейтинговою	**інституційною**	
розділів кваліфікаційної роботи				
спеціальний	*проф. Мещеряков Л.І.*			
Рецензент	*доц. Коротенко Г.М.*			
Нормоконтролер	*проф. Лактіонов І.С.*			

Дніпро
2023

Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

ЗАТВЕРДЖЕНО:
Завідувач кафедри
Програмного забезпечення комп'ютерних систем
(повна назва)
_____ Алексєєв М. О.
(підпис) (прізвище, ініціали)
« » _____ 2023 року

ЗАВДАННЯ
на виконання кваліфікаційної роботи

спеціальності *121 Інженерія програмного забезпечення*
(код і назва спеціальності)

студенту *121м-22-3* *Петрушенко А. В.*
(група) (прізвище та ініціали)

Тема кваліфікаційної роботи: Дослідження продуктивності мобільних додатків в залежності від різних методів керуванням з застосуванням фреймворку Flutter

1 ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Наказ ректора НТУ «Дніпровська політехніка» від «09» жовтня 2023 р. № 1227-С

2 МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт досліджень – процеси аналізу швидкодії методів управління станом у мобільних застосунках.

Предмет досліджень – методи управління станом BLoC, MobX, Redux фреймворку Flutter.

Мета роботи – підвищення та удосконалення процесів розробки мобільних застосунків за рахунок дослідження швидкодії використання методів управління станом BLoC, MobX та Redux фреймворку Flutter.

Вихідні дані для проведення роботи – теоретичні та експериментальні дослідження, управління станом BLoC, MobX, Redux фреймворку Flutter, методи обробки експериментальних даних.

3 ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Новизна запропонованих рішень кваліфікаційної роботи визначається вдосконаленням продуктивності та ефективності швидкодії мобільних застосунків з використанням власних алгоритмів методів управлінням станом BLoC, Redux і MobX фреймворку Flutter. **Новизна запропонованих рішень** визначається тим, що надані програмні алгоритми обумовлюють використання сучасних інструментів, підходів, а

також покращення їх продуктивності та ефективності у мобільних застосунках фреймворку Flutter.

Практична цінність результатів полягає у тому, що їх можна використовувати у подальшому під час вибору методу управління станом у бізнес-логіці мобільного застосунку.

4 ВИМОГИ ДО РЕЗУЛЬТАТІВ ВИКОНАННЯ РОБОТИ

У результаті роботи повинен бути розроблений програмний комплекс для вирішення задачі підвищення продуктивності та ефективності роботи мобільних застосунків з використанням методів управління станом фреймворку Flutter.

5 ЕТАПИ ВИКОНАННЯ РОБІТ

Найменування етапів робіт	**Строки виконання робіт (початок – кінець)**
Аналіз теми та постановка задачі	12.09.2023-30.09.2023
Розробка моделі, методів та програмного забезпечення розрахунку ефективності роботи мобільних застосунків	01.10.2023-31.10.2023
Використання програми та аналіз отриманих результатів	01.11.2023-12.12.2023

Завдання видав _____ *Мещеряков Л. І.*
(підпис) (прізвище, ініціали)

Завдання прийняв до виконання _____ *Петрушенко А. В.*
(підпис) (прізвище, ініціали)

Дата видачі завдання: *12.09.2023 р.*

Термін подання кваліфікаційної роботи до ЕК: _____

РЕФЕРАТ

Пояснювальна записка: 100 стор., 60 рис., 7 таблиць, 5 додатків, 50 джерел.

Об’єкт дослідження: процеси аналізу швидкодії методів управління станом у мобільних застосунках.

Предмет дослідження: методи управління станом BLoC, Redux і MobX.

Мета магістерської роботи: підвищення та удосконалення процесів розробки мобільних застосунків за рахунок дослідження швидкодії використання методів управління станом BLoC, MobX та Redux фреймворку Flutter.

Методи дослідження. Для розв'язання поставлених задач використані: аналіз швидкодії мобільних застосунків, методи управління станом додатку, об'єктно-орієнтоване програмування.

Новизна запропонованих рішень кваліфікаційної роботи визначається вдосконаленням продуктивності та ефективності швидкодії мобільних застосунків з використанням власних алгоритмів методів управлінням станом BLoC, Redux і MobX фреймворку Flutter.

Практична цінність результатів полягає у тому, що їх можна використовувати у подальшому під час вибору методу управління станом у бізнес-логіці мобільного застосунку.

Область застосування: програмна розробка кросплатформних мобільних застосунків з використанням методів управлінням станом BLoC, Redux і MobX фреймворку Flutter.

Значення роботи та висновки: розроблений мобільний застосунок за допомогою фреймворку Flutter надає гарні показники навіть при великому навантаженні.

Список ключових слів: мобільний застосунок, управління станом, BLoC, Dart, Flutter, MobX, Redux.

ABSTRACT

Explanatory note: 100 pages, 60 figures, 7 tables, 5 appendices, 50 sources.

The object of research: the processes of analyzing the performance of state management methods in mobile applications.

Research subject: BLoC, Redux and MobX state management methods.

The purpose of the master's work: to increase and improve the development processes of mobile applications due to the study of the speed of using the BLoC, MobX and Redux state management methods of the Flutter framework.

Research methods. To solve the tasks, the following are used: mobile application speed analysis, application state management methods, object-oriented programming.

The novelty of the proposed solutions for qualification work is determined by improving the performance and speed of mobile applications using proprietary algorithms of BLoC, Redux and MobX state management methods of the Flutter framework.

The practical value of the results is that they can be used later when choosing a state management method in the business logic of a mobile application.

Scope: software development of cross-platform mobile applications using BLoC, Redux and MobX state management methods of the Flutter framework.

Value of the work and conclusions: The mobile application developed using the Flutter framework provides good performance even under heavy load.

List of keywords: mobile app, state management, BLoC, Dart, Flutter, MobX, Redux.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

БД – база даних

ОС – операційна система

ПЗ – програмне забезпечення

ПП – програмний продукт

API – інтерфейс прикладного програмування

BLoC – компонент бізнес-логіки

ЗМІСТ

ВСТУП

Сьогодення у сфері інформаційних технологій (ІТ) є різноманітним та насиченим. Розробка програмного забезпечення (ПЗ) вже далеко зайшла за межі можливого і неможливого. Найпопулярнішим видом ПЗ є мобільні застосунки, адже смартфон саме той пристрій, який завжди під рукою. У світі мобільної розробки важливою складовою є ефективне управління станом застосунку. Вірно організоване управління станом є ключовим фактором, який впливає на швидкодію та загальну продуктивність ПЗ. Методів управлінням станом може бути багато навіть у межах одного фреймворку, тому так важливо розуміти, який з них швидше працює, надає навантаження на оперативну пам’ять або займає найбільше пам’яті вихідного коду. Дуже мало робіт направлене на дане дослідження. Це є вагомим недоліком, адже перевірка вище вказаних метрик є достатньо важливими параметрами, які можуть впливати на бізнес і кінцевого користувача програмним продуктом (ПП).

Одними із найпопулярніших засобів розробки для конструювання логічного та вірного управління станом є Flutter та Dart. Дана пара досі набирає свою популярність у всьому світі. Вони вже мають велику аудиторію та кількість офіційної та сторонньої документації, але не мають достатньої інформації про швидкодію методів управління станом на мобільних застосунках.

Таким чином актуальність обраної теми обґрунтована розвитком наукової діяльності, дослідженням методів управління станом на основі фреймворку Flutter. Результати даної роботи дозволяють у подальшому використовувати отримані результати для проведення додаткових досліджень, а також у визначенні методу управління станом, необхідного для розробки.

Метою роботи є підвищення та удосконалення процесів розробки мобільних застосунків за рахунок дослідження швидкодії використання методів управління станом BLoC, MobX та Redux фреймворку Flutter.

Для досягнення мети роботи необхідно вирішити наступні задачі:

– провести аналіз розробки кросплатформних застосунків з використанням фреймворків;

– провести аналіз існуючих методів управління станом на основі кросплатформних фреймворків;

– провести аналіз проблематики розробки кросплатформних застосунків на основі фреймворку Flutter;

– описати функціональні вимоги мобільного застосунку;

– спроєктувати архітектуру мобільного застосунку;

– виконати опис обраних методів управління станом;

– виконати опис обраних засобів розробки мобільного застосунку;

– виконати програмну реалізацію мобільного застосунку з використанням обраних методів управління станом;

– проаналізувати отримані результати;

– обґрунтувати економічну ефективність розробленого ПП.

Об’єктом дослідження є швидкодія мобільних застосунків , які розроблені на основі фреймворку Flutter.

Предметом дослідження є методи управління станом у фреймворку Flutter.

Методами дослідження є метрики, за допомогою яких виконується вимір часу виконання операцій та обчислення розмірів коду.

Новизною запропонованих рішень є проведене дослідження, яке показує, що методи управління станом надають різні результати у швидкодії та розміру вихідного коду.

Практичним значенням є використання результатів роботи у подальшому при виборі методу управління станом при розробці майбутніх мобільних застосунків.

Особистим внеском автора є пояснювальна записка, розроблені алгоритми, що використовувалися при реалізації мобільних застосунків та проведене дослідження, яке описано.

РОЗДІЛ 1
АНАЛІЗ ТЕОРЕТИЧНИХ ТА МЕТОДОЛОГІЧНИХ АСПЕКТІВ ДОСЛІДЖУВАНОЇ ЗАДАЧІ

1.1. Аналіз розробки кросплатформних застосунків з використанням фреймворків

Розробка кросплатформних застосунків з використанням фреймворків стала важливою та популярною задачею у світі розробки мобільних та вебзастосунків. Даний підхід дозволяє розробникам створювати застосунки, які можуть працювати на різних платформах, таких як Android та iOS, з мінімальними змінами в коді [1-3].

Існують деякі ключові аспекти аналізу розробки кросплатформних застосунків [1-3] за допомогою фреймворків:

– вибір фреймворку: вибір повинен відбуватися з урахуванням вимог проєкту, досвіду розробників та екосистеми фреймворку; існує безліч фреймворків для кросплатформної розробки, таких як Flutter, React Native, Xamarin тощо;

– продуктивність: різні фреймворки можуть мати різну продуктивність у залежності від складності застосунку та оптимізації, що доступні в кожному фреймворку;

– інтерфейс користувача: існує різні інструменти для створення користувальницького інтерфейсу, важливо лише оцінити наскільки просто можна створити гарні та інтуїтивно зрозумілі інтерфейси;

– доступ до апаратних ресурсів: деякі фреймворки можуть мати обмеження в доступі до апаратних ресурсів пристрою, таким як камера, певні датчики, геолокація тощо; необхідно впевнитися, що обраний фреймворк підтримує необхідний функціонал;

– екосистема та спільнота: дані аспекти можуть значно спростити розробку, надаючи готові рішення, бібліотеки, підтримку тощо;

– вартість і ліцензія: деякі фреймворки можуть бути повністю безкоштовними, але ліцензії, які вони надають можуть надавати додаткові витрати;

– безпека: забезпечення безпеки є критичним аспектом; необхідно оцінити наскільки обраний фреймворк забезпечує захист даних і користувальницьку конфіденційність;

– підтримка та оновлення: активний розвиток фреймворків є важливим аспектом, адже технології не стоять на місці, тому важливо переконатися, що обраний фреймворк має регулярні оновлення та підтримку.

Аналіз розробки кросплатформних застосунків потребує уважного вивчення всіх вищеперерахованих аспектів, а також обліку конкретних вимог та цілей проєкту.

Розробка будь-якого ПЗ виконується за певними етапами, які проходить ПП протягом всього свого життєвого циклу. Етапи розробки кросплатформних застосунків наведено на рисунку 1.1.

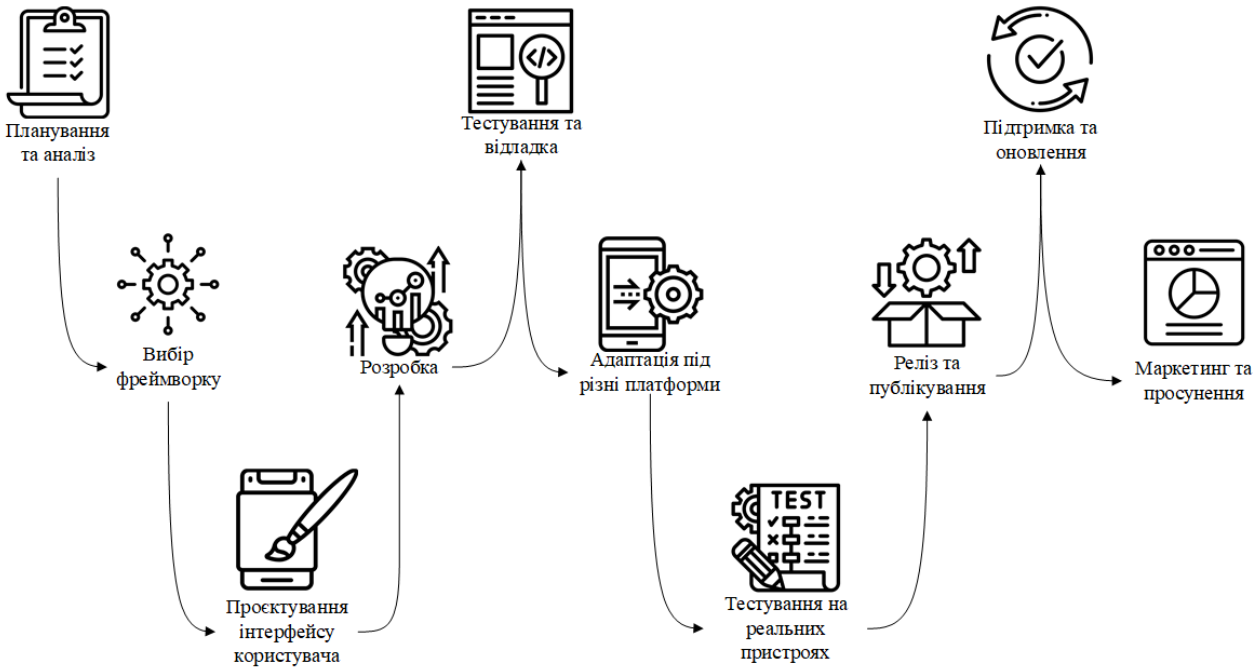

Рис. 1.1. Графічне представлення етапів розробки кросплатформних застосунків

За рисунком 1.1 наведені етапи розробки кросплатформних застосунків [4], що описані в таблиці 1.1.

Таблиця 1.1

Таблиця етапів розробки кросплатформних застосунків

№ п/п	Найменування етапу	Опис
1	2	3
1	Планування та аналіз	Визначення цілей та вимог до застосунків, дослідження цільової аудиторії та її вимог, визначення основних функцій та характеристик застосунку
2	Вибір фреймворку	Визначення доступних кросплатформних фреймворків, порівняння їх функціоналу, продуктивності, екосистему
3	Проєктування інтерфейсу користувача	Створення макетів та дизайну інтерфейсу користувача, урахування різниці дизайну під різні платформи
4	Розробка	Розробка застосунку, використовуючи обраний фреймворк, логіку, обробку подій, взаємодію із сервером та управління станом
5	Тестування та відладка	Тестування на ранніх етапах допоможе знайти додаткові помилки в коді. Приділення уваги продуктивності та сумісності із різними пристроями
6	Адаптація під різні платформи	Розробка специфічних компонентів або модулів під кожну платформу, за необхідністю

Продовження таблиці 1.1

1	2	3
7	Тестування на реальних пристроях	Проведення тестування на реальних пристроях кінцевого продукту
8	Реліз і публікація	Підготовка застосунку до релізу, публікування його в магазинах застосунків, стеження за процесом затвердження та випуску застосунку
9	Підтримка та оновлення	Підтримка застосунку, реагування на зворотній зв’язок користувачів, випуск оновлень
10	Маркетинг та просунення	Розробка маркетингової стратегії для залучення користувачів, популяризація застосунку за допомогою різних каналів в соціальній мережі або реклами

Кожен із наведених етапів (див. табл. 1.1) потребує детального планування та виконання, щоб створити успішний кросплатформний застосунок, який буде задовольняти потреби користувачів і працювати на різних платформах.

Розробка кросплатформних застосунків є динамічним процесом. Кожен проєкт може мати свої унікальні вимоги. Важливо адаптувати підхід до розробки у відповідності з конкретними задачами та цілями проєкту. Наприклад, у кожної окремої розробки є свої додаткові потреби [5-7]:

– керування версіями та вихідним кодом: використання систем таких як Git допомагає сумісній роботі над проєктом та забезпечує безпечне зберігання коду;

– безпека: забезпечення безпеки застосунку, приділяючи увагу автентифікації, захисту даних тощо;

– оптимізація продуктивності: оптимізація застосунку шляхом керування пам’яттю, запитів до бази даних (БД), зниження потреб енергії;

– локалізація: надання локалізації для різних мов та культур, формату дати, часу, валют;

– монетизація: використання монетизації для продажу застосунку, реклами, підписки;

– збір та аналіз даних: використання аналітичних інструментів для збору аналізу даних про поведінку користувачів для покращення застосунку;

– облік вимог платформ: урахування потреб для різних платформ, рекомендацій до розробки;

– адаптація до змін: слідкування за оновленнями операційних систем (ОС) для забезпечення сумісності застосунку із ними;

– технічна підтримка: забезпечення підтримки для користувачів, щоб допомогти їм вирішити їх проблеми та питання, що виникли.

За даним аналізом можна стверджувати, що програмістам слід обирати фреймворк та інші засоби розробки таким чином, щоб виконати цілі та стратегії проєкту та досягнути успіхів у розробці кросплатформних застосунків.

1.2 Аналіз методів управління станом на основі кросплатформних фреймворків

Управління станом у мобільних застосунках – це процес управління та зберігання даних і стану застосунку на мобільному пристрої. Стан ПП включає в себе інформацію, яка може змінитися за часом та впливати на поведінку та відображення застосунку. Управління станом у мобільних застосунках є важливим етапом розробки. Воно забезпечує правильну роботу ПЗ та збереження даних між сеансами. Також управління станом може впливати на користувальницький досвід, так як вірно розроблене управління станом дозволяє застосунку реагувати на дії користувача та зміну середовища [5-7].

Існує декілька методів та інструментів для керування станом [8]:

– локальний стан: метод, що використовується на рівні конкретного компонента або віджету;

– глобальний стан: дозволяє розподілити стан між різними компонентами або віджетами;

– реактивне програмування: використання потоку даних та подій для управління станом и реагування на їх зміну;

– використання маршрутизації та посилань: часто використовується у вебзастосунках, може представляти певні стани застосунку та переходи між посиланнями, що забезпечує їх зміну;

– модель подій: базується на відправці та прослуховуванні подій.

Сьогодні існує багато фреймворків, які мають методи управління станом, найпопулярнішими із них є Flutter, React Native, Xamarin тощо. Кожен із них має свої переваги та недоліки та може бути найкращим вибором у залежності від конкретних вимог проєкту [8].

1.2.1. Методи управління станом у фреймворку Flutter

Управління станом у фреймворку Flutter відноситься до тих даних, які контролюють поведінку та користувальницький інтерфейс застосунку. Стан у них змінюється на відповідь введення користувача та ініційовані події. Для розробників важливо ефективно керувати станом для створення адаптивних та продуктивних ПП [9].

Для виконання управління станом у Flutter є декілька підходів, починаючи з найпростішого setState(), закінчуючи складнішими, такими як Business Logic Component (BLoC), MobX, Redux, GetX [10-12]:

– setState():базовий метод управління станом, виклик через який дозволяє оновити стан віджету та викликати його перемальовування;

– BLoC: розділяє бізнес-логіку та користувальницький інтерфейс, а також забезпечує односпрямований потік даних;

– Redux: надає єдине сховище для всього стану застосунку та забезпечує передбачуваність і односпрямований потік даних;

– MobX: базується на реактивному програмуванні, надає інструменти для створення спостережуваних об’єктів та реактивних віджетів;

– GetX: надає простий та швидкий спосіб управління станом.

У методів управління станом фреймворку Flutter є свої сильні та слабкі сторони.

Перевагами використання методів управління станом у фреймворку Flutter [12] є наступні аспекти:

– однорідність інтерфейсу: забезпечення однакових та консистентних інтерфейсів для всіх платформ, що означає однакову роботу стану на всіх пристроях;

– продуктивність: пропонування високої продуктивності завдяки компіляції в нативний код;

– інструменти для управління станом: надання різноманітних засобів управління станом, що дозволяє розробникам обирати той підхід, який їх задовільнить;

– живе перенавантаження: можливість миттєвого перенавантаження застосунку в реальному часі, що робить процес розробки та відладки більш ефективним та швидким;

– обширна спільнота та документація: полегшення пошуку необхідної інформації через велику спільноту та документацію, як офіційну так і сторонню.

Недоліками використання методів управління станом у фреймворку Flutter [12] є наступні аспекти:

– надмірність коду: у залежності від обраного методу управління станом, код може стати надмірним та об’ємним, що може ускладнювати розуміння та підтримку застосунку;

– складність вибору підходу: через велике різноманіття методів вибір починаючих розробників може бути складним;

– складність управління станом у великих проєктах: потреба більш кропіткого проєктування та організації коду;

– складність відладки: деякі методи управління станом можуть бути складні у відладці через асинхронні операції та потік даних.

За наданими перевагами та недоліками можна сказати, що методи управління станом у Flutter є різноманітними, але необхідно підходити до їх вибору прискіпливо, щоб обрати вірний підхід. У цілому фреймворк надає ті інструменти та методи, які можуть бути легко адаптованими під конкретні вимоги проєкту.

1.2.2. Методи управління станом у фреймворку React Native

Методи управління станом у фреймворку React Native є важливими аспектом мобільних застосунків. Існують наступні найбільш розповсюджені методи управління станом у React Native [13-14]:

– локальний стан компонентів: найпростіший спосіб управління станом для оновлення локального стану компонентів на низькому рівні;

– Redux: частіше за все використовується для централізованого зберігання та керування станом;

– MobX: забезпечує реактивне програмування та більш гнучкий підхід до управління станом;

– управління навігацією: використання бібліотек для керування навігацією, вони надають засоби для передачі параметрів між екранами.

Вибір методу управління станом залежить від вимог проєкту, розміру застосунку та досвіду розробки. Рекомендується обирати метод, який якнайкраще відповідає конкретним потребам проєкту.

Управління станом у React Native має свої переваги та недоліки, які слід враховувати під час розробки мобільних застосунків [15].

Перевагами управління станом у React Native є [16-17]:

– компонентний підхід: фреймворк заснований на компонентах та керування станом здійснюється через локальний стан компонентів, що робить код більш модульним та зрозумілим;

– простота та швидкість розробки: надання функціонального підходу до управління станом за допомогою станів компонентів та хуків, що може прискорити процес розробки та зробити код більш легким для розуміння;

– екосистема: існує багато бібліотек та інструментів для управлінням стану, що надає розробникам великий вибір підходів;

– широка підтримка спільноти: велика спільнота розробників, що означає великий обсяг матеріалів, бібліотек та рішень для управління станом.

Недоліками управління станом у React Native є [16-17]:

– складність вибору підходу: вибір методу управління станом може бути утрудненим, особливо для починаючих розробників;

– продуктивність: через надмірність оновлень стан може викликати перемальовування компонентів та уповільнити застосунок;

– складність відладки: може бути утрудненою задачею з множиною компонентів і станів, а помилки в управлінні станів можуть важко виявлятися та виправлятися;

– залежність від сторонніх бібліотек: їх використання може створювати залежність від сторонніх розробників та підтримки цих бібліотек;

– перекомпонування: використання механізмів перекомпонування для визначення того, які компоненти необхідно перемалювати при зміні стану, адже невірне використання може викликати надмірне перекомпонування та знизити продуктивність.

За перерахованими перевагами та недоліками можна визначити, немає універсального підходу, тому розробники можуть обирати ті інструменти та методи, які найкращим чином відповідають їх задачам та вимогам.

1.2.3. Методи управління станом у фреймворку Xamarin

У фреймворку Xamarin для управління станом застосунку можна використовувати наступні різноманітні підходи [18]:

– «ViewModels» з прив’язкою даних: дозволяє автоматично оновлювати інтерфейс при зміні даних;

– події та делегати: створення подій у класі та підписка на нього в уявленнях;

– «Singleton» для глобального стану: створення класу для зберігання глобального стану застосунку та звертання до нього із різних частин застосунку;

– сторонні бібліотеки: надають більш просунуті інструменти для управління станом у застосунках.

Вибір вищенаведених методів залежить від розміру застосунку та вимог до нього. При цьому важливо стежити за чистотою коду та забезпечувати підтримку програмного продукту.

Управління станом фреймворку Xamarin може бути реалізовано різноманітними способами, кожен з яких має свої переваги та недоліки. Перевагами методів управління Xamarin [18] є:

– простота використання: використання прив’язки даних для автоматичного оновлення користувальницького інтерфейсу при зміні даних у модель, що робить код більш зрозумілим та зменшує кількість рутини;

– кросплатформність: дозволяє створювати кросплатформні застосунки, використовуючи додатково мови програмування;

– відладка та інструменти: надає потужні засоби відладки та інструменти для аналізу продуктивності, які допомагають виявити та усунути проблемі із станом.

Недоліками методів управління Xamarin [18] є:

– складність: у великих проєктах управління станом може стати складною задачею, що може потребувати додаткового часу та зусиль на розробку;

– оновлення та асинхронність: необхідність стежити за потоками даних і синхронізацією через оновлення даних та асинхронні операції;

– надмірна складність: впровадження надмірної складності в управлінні станом, що може ускладнити код та його розуміння;

– вибір підходу: ускладнення вибору підходу через велику кількість підходів управління станом;

– залежність від сторонніх бібліотек: ускладнення оновлення застосунку через використання сторонніх бібліотек.

Важливо підходити до вибору методу управління станом в Xamarin з урахуванням вимог проєкту. Хороше розуміння переваг і недоліків різних підходів допоможе прийняти правильне рішення для застосунка.

1.2.4. Порівняння фреймворків Flutter, React Native та Xamarin

«Stack Overflow» проводить щорічне опитування розробників, в якому розробники відповідають на питання про те, як вони навчаються, які інструменти використовують тощо. Ця статистика часто використовується для того, щоб повідомити про задоволеність розробників технологіями, які вони використовують або потенційно будуть використовувати [19].

На рисунках 1.2-1.3 наведені результати опитування про використовувані фреймворків та іншого інструментарію розробниками.

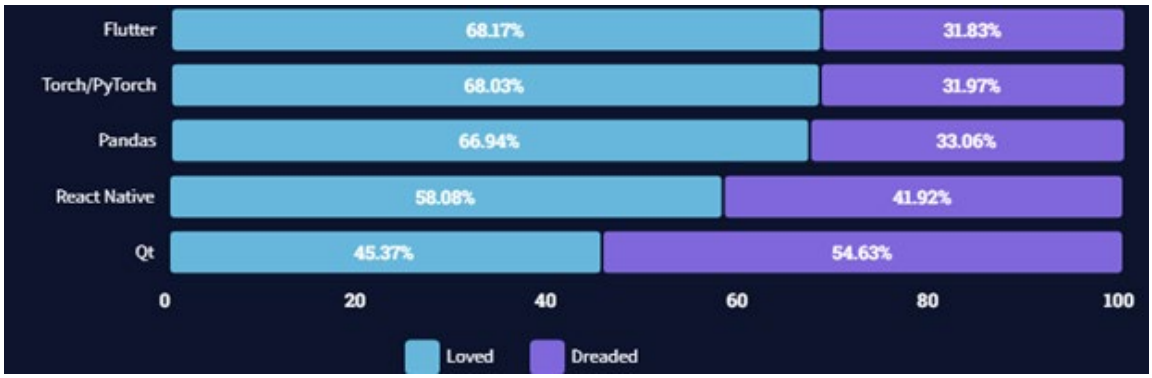

Рис. 1.2. Графік результатів опитування «Stack Overflow» серед розробників щодо використання фреймворків Flutter та React Native [19]

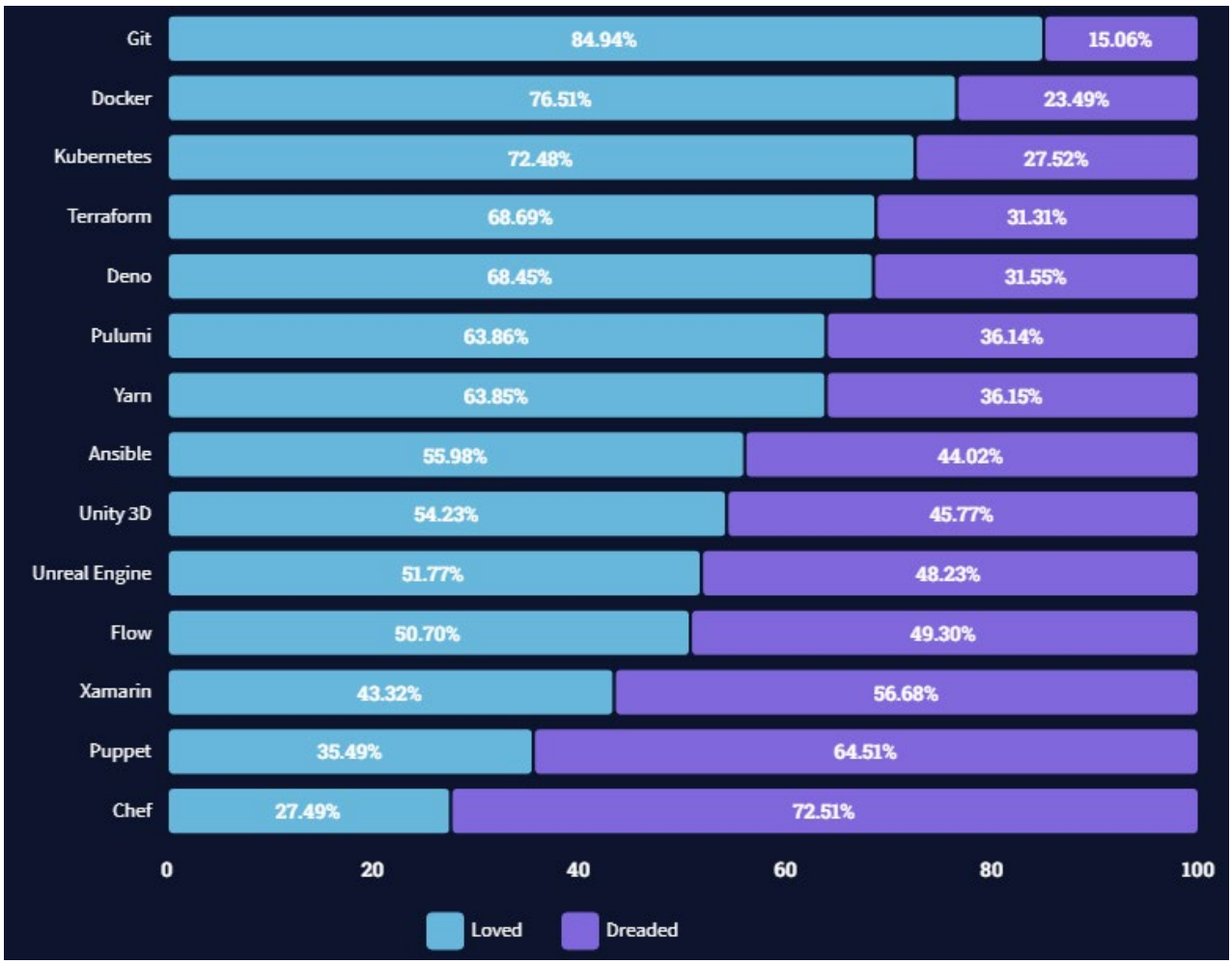

Рис. 1.3. Графік результатів опитування «Stack Overflow» серед розробників щодо використання фреймворку Xamarin [19]

За рисунками 1.2-1.3 видно, що найулюбленішим фреймворком є Flutter, який набрав 68,17%, React Native отримав 58,08%, а Xamarin опинився на останньому місці, набравши 43,32% [19].

За минулорічним дослідженням було побудовано графік популярності фреймворків, починаючи з 2009 року [20]. Результати наведені на рисунку 1.4.

За графіком (див. рис. 1.4) можна стверджувати, що Flutter займає перше місце за популярністю, ніж два інших фреймворки. Також, за графіком можна побачити, що популярність і використання Flutter та React Native постійно зростає, у той час як популярність Xamarin постійно йде на спад.

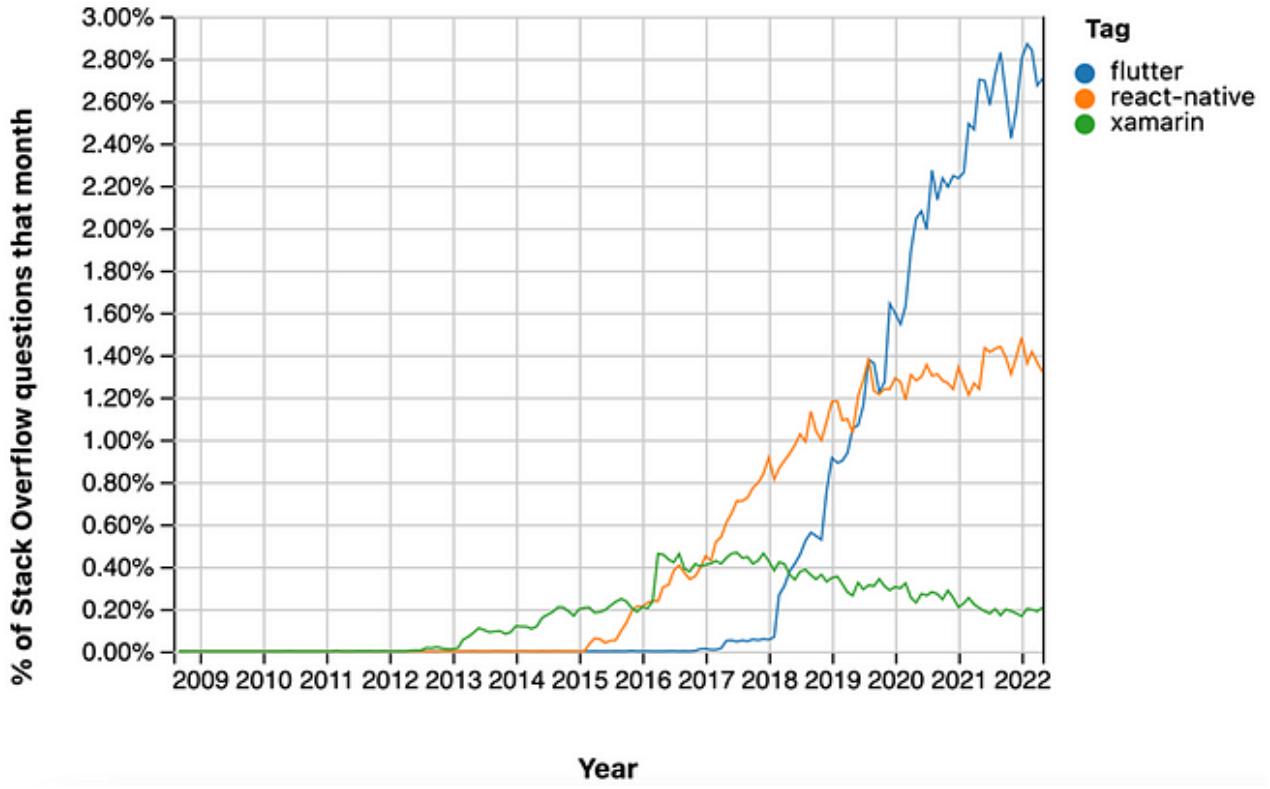

Рис. 1.4. Графіки популярності фреймворків Flutter, React Native та Xamarin [20]

Ще одне дослідження від «Statista», яке залучило розробників для перевірки популярності фреймворків кросплатформної розробки. Дослідження проводилося на опитуванні з 2019 до 2022 року, а результати публікувалися на початку 2023 року [21]. Результати проведеного опитування наведенні на рисунку 1.5.

Для кожного із фреймворку є по чотири результати дослідження окремо за відповідними кольорами на графіку:

– 2019 рік: колір голубий;
– 2020 рік: колір синій;
– 2021 рік: колір сірий;
– 2022 рік: колір червоний.

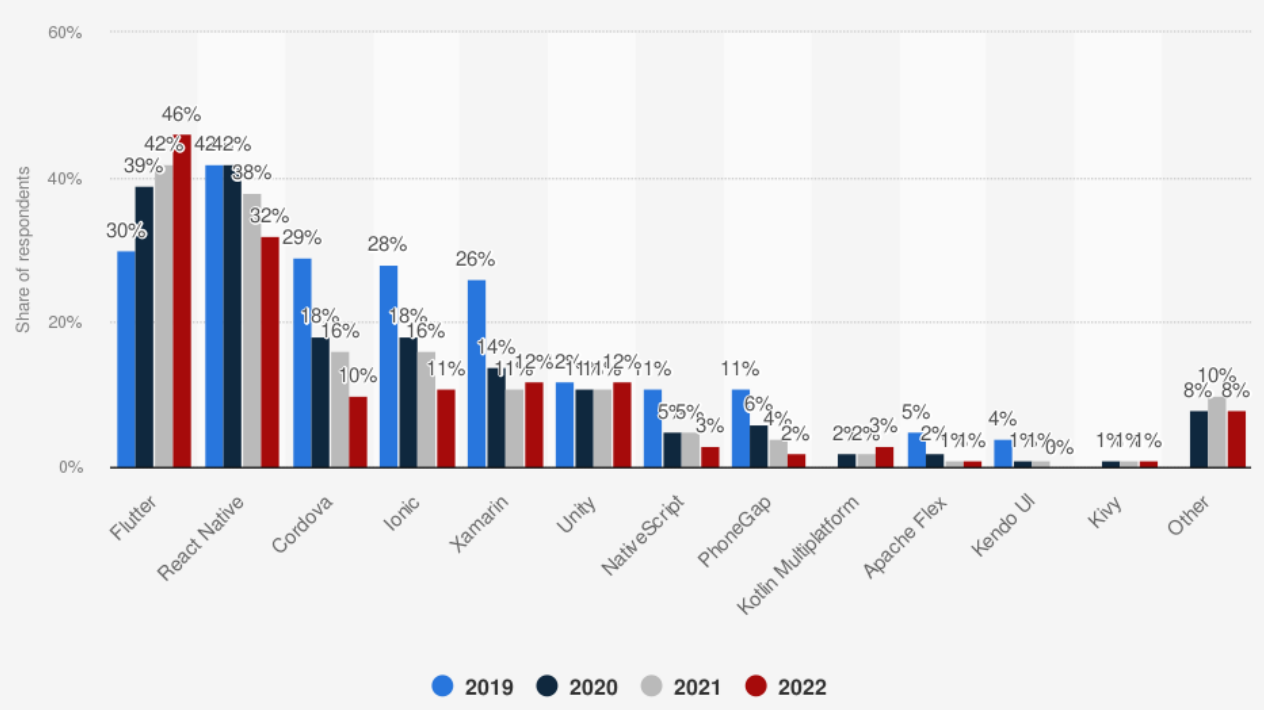

Рис. 1.5. Графік результатів «Statista» на опитування популярності фреймворків для кросплатформної розробки [21]

Результати дослідження можна вивести в таблицю 1.2.

Таблиця 1.2

Таблиця результатів дослідження популярності фреймворків для кросплатформної розробки

Фреймворк	Роки			
	2019	2020	2021	2022
Flutter	30%	39%	42%	46%
React Native	42%	42%	38%	29%
Xamarin	26%	14%	11%	12%

За результатами видно (див. рис. 1.5, табл. 1.2), що популярність Flutter у порівнянні з 2019 роком зріс на 16%, React Native – знизилася на 13%, а Xamarin – знизилася на 14% [21].

Якщо ж порівнювати фреймворки за характеристиками, тоді отримаємо результати [22], що наведені в таблиці 1.3.

Таблиця 1.3

Таблиця порівняння фреймворків Flutter, React Native та Xamarin

Характеристики	Фреймворки		
	Flutter	React Native	Xamarin
Мова програмування	Dart	JavaScript, TypeScript	C# з .NET
Користувальницький інтерфейс	Відсутність елементів інтерфейсу за замовчуванням	Підтримка елементів інтерфейсу за замовчуванням	
Продуктивність	Бінарна компіляція коду	Взаємодія з нативом через так звані мости	
Графічний інтерфейс	Власні віджети	Нативні контролери інтерфейсу	
Придатність	Всі програми		

Всі перераховані вище технології мають свої плюси і мінуси: Flutter – швидкий і плавний, React Native – сумісний із популярною мовою програмування JavaScript, а Xamarin – безкоштовний для невеликої команди, але вибір правильної технології у руках розробників відповідно потреб проєкту. За проведеними дослідженнями та аналізом, можна стверджувати, що Flutter більш вдало впорається із поставленою задачею на розробку мобільного застосунку.

1.3. Аналіз проблематики розробки кросплатформних застосунків на основі фреймворку Flutter

Як і в будь-якому іншому інструментарію розробки, Flutter має свої проблемні аспекти в розробці. Серед всіх існуючих проблем [23-26], можна виділити наступні найвагоміші:

– продуктивність: у деяких випадках не вірно оптимізований, може споживати більше ресурсів та мати проблеми з продуктивністю на пристроях;

– складність інтеграції з нативним кодом: при інтеграції з сторонніми плагінами або бібліотеками можуть виникати складності;

– проблеми з адаптацією інтерфейсів: виникнення складності при розробці інтерфейсу, який виглядає та поводиться однаково на всіх платформах;

– складність з анімаціями та інтерфейсами: розробка може бути більш складною, ніж в інших фреймворках;

– недостача бібліотек для специфічних завдань: у деяких випадках для виконання специфічних задач може потребуватися створення власних бібліотек або модулів;

– сумісність з платформою: деякі функції або програмні інтерфейси, доступні на деяких платформах, можуть бути складними для реалізації в застосунках чи мають різні інтерфейси для взаємодії з нативними компонентами платформи тощо.

Не дивлячись на перераховані проблеми, Flutter залишається потужним інструментом для розробки кросплатформних застосунків, що підтверджують проведені дослідження.

1.4. Висновки за розділом 1

За даним розділом був проведений аналіз розробки кросплатформних застосунків з використанням фреймворків. Проведений аналіз методів управлінням станом у трьох найпопулярніших фреймворків Flutter, React Native, Xamarin. Проведена порівняльна характеристика фреймворків Flutter, React Native, Xamarin і визначено, що перший є найпопулярнішим і найкращим на сьогоднішній день. Визначені проблемні аспекти розробки кросплатформних застосунків на фреймворку Flutter.

Після виконання даного ряду завдань можна переходити до проєктування архітектури ПП, обґрунтуванню інструментарію розробки та опису методів управління станом фреймворку Flutter.

РОЗДІЛ 2
ПРОЄКТУВАННЯ СИСТЕМНИХ ЗВ'ЯЗКІВ АРХІТЕКТУРИ МЕТОДІВ УПРАВЛІННЯ СТАНОМ НА ОСНОВІ ФРЕЙМВОРКУ FLUTTER

2.1. Опис функціональних вимог мобільного застосунку

Установлення функціональних вимог є важливим етапом перед проєктуванням і розробкою програмного забезпечення. За його допомоги можна чітко визначити вимоги до ПП, зрозуміти, які екрани необхідні для застосунку, що вони будуть вміщувати [27-29] тощо.

Для виконання дослідження буде створено мобільний застосунок «TaskHub», який буде написаний на трьох методах управління станом. Він буде містити наступні три екрани:

1. Список задач: представляє список наявних задач, створених користувачем.

2. Створення задач: представляє екран створення задач користувачем.

3. Редагування задач: представляє екран редагування обраної задачі.

Екран «Список задач» буде містите наступні функціональні можливості:

– виводити наявний список задач;

– при натисненні на певну задачу відкривати повну інформацію про обрану задачу, надаючи можливість до її редагування;

– надавати можливість видаляти задачі, шляхом свайпу вліво на певній задачі;

– створювати задачі шляхом натиснення на відповідну кнопку та направляючи до екрану «Створення задач»;

– оновлювати список задач шляхом свайпу донизу;

– надавати можливість відмітити виконання чи невиконання певної задачі;

– при виникненні помилки надаватиме можливість повторити завантаження екрану шляхом натиснення на відповідну кнопку.

Екран «Створення задач» буде містити наступні функціональні можливості:

– надавати текстове поле для ведення найменування задачі;

– надавати текстове поле для введення опису задачі;

– надавати поле для вибору пріоритетності задачі;

– надавати поле для вибору категорії задачі;

– надавати доступ до календарю задля вибору дати закінчення задачі;

– збереження задачі шляхом натиснення на відповідну кнопку;

– виведення повідомлення про помилку.

Екран «Редагування задачі» буде містити наступні функціональні можливості:

– надавати доступ до редагування текстових полів найменування та опису задачі;

– надавати доступ до редагування вибору пріоритетності та категорії задачі;

– надавати доступ до редагування зміни дати закінчення задачі;

– збереження відредагованих змін шляхом натиснення на відповідну кнопку;

– виведення повідомлення про помилку.

Додатково слід зазначити такі аспекти:

а) усі дані повинні зберігатися у спеціальному сховищі даних;

б) робота з даними відноситься до двох сутностей: користувач і задачі;

в) для сутності задачі повинні бути реалізовані методи взаємодії такі як:

1) завантажити список задач;

2) створити нову задачу;

3) оновити існуючий список задач;

4) редагувати існуючу задачу;

5) видалити існуючу задачу.

Шар віджетів («Widget Layer») повинен бути незмінним для всіх трьох методів управління станом, шар бізнес логіки повинен змінюватись у

відповідності до методу управління станом, шар даних («Data Layer») повинен також бути незмінним [30].

2.2. Проєктування архітектури мобільного застосунку

Проєктування є важливим етапом перед розробкою програмного продукту. Воно надає візуальне представлення всіх необхідних елементів програми, показує зв’язки між користувачами та застосунком, повідомляє про класи, компоненти тощо. Для мобільного застосунку «TaskHub» доцільно спроєктувати такі діаграми як варіантів використання, класів, компонентів і макети користувальницького інтерфейсу [31].

2.2.1. Проєктування діаграми варіантів використання

Діаграма варіантів використання надає графічне представлення, як користувач пов’язаний із мобільним застосунком, які функції йому доступні, так як вони пов’язані між собою [32]. На рисунку 2.1 наведена діаграма варіантів використання для мобільного застосунку «TaskHub».

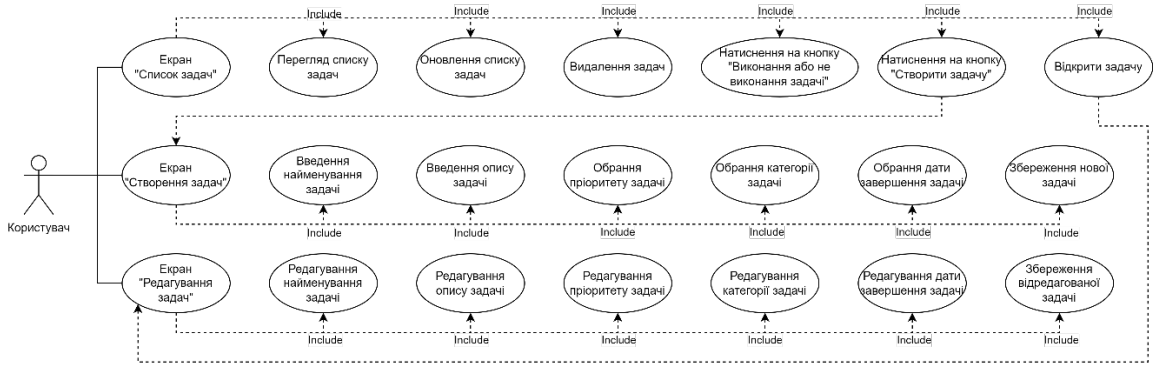

Рис. 2.1. Графічне зображення діаграми варіантів використання мобільного застосунку «TaskHub»

За наведеною діаграмою (див. рис. 2.1) можна сказати, що вона відповідає заявленим функціональним вимогам, які були описані вище.

На діаграмі важливими є наступні аспекти:

– користувач пов’язаний із наявними екранами зв’язками асоціації; у нього є можливість взаємодіяти із всіма наданими йому функціями на всіх екранах;

– прецеденти, що присутні в кожному із екранів пов’язані зв’язком включення (include); даний зв’язок показує, що на наданий функціонал міститься у відповідних екранах;

– прецеденти, які мають відношення до інших екранів пов’язані зв’язком залежності; наприклад, прецедент «Create tasks screen», що пов’язаний із «Click on the button «Create task» є залежним від нього, бо поки користувач не натисне кнопку «Створити задачу», екран «Створення задачі» не буде відкритим.

За допомогою діаграми варіантів використання (див. рис. 2.1) було з’ясовано, як саме користувач взаємодіє з мобільним застосунком, що допоможе під час розробки ПЗ.

2.2.2. Проєктування діаграми класів

Діаграма класів допомагає якісно спланувати розробку мобільного застосунку та визначити, які саме класи, атрибути та методи необхідні при виконанні реалізації [33].

Для цього необхідно виконати проєктування таких діаграм, як:

– діаграма шару даних;

– діаграми бізнес-логіки для кожного із методів управління станом окремо.

Діаграма шару даних наведена на рисунку 2.2. Вона надає розуміння, як сховище даних взаємодіє із іншими класами, які містить методи та атрибути, які зв’язки їх поєднують.

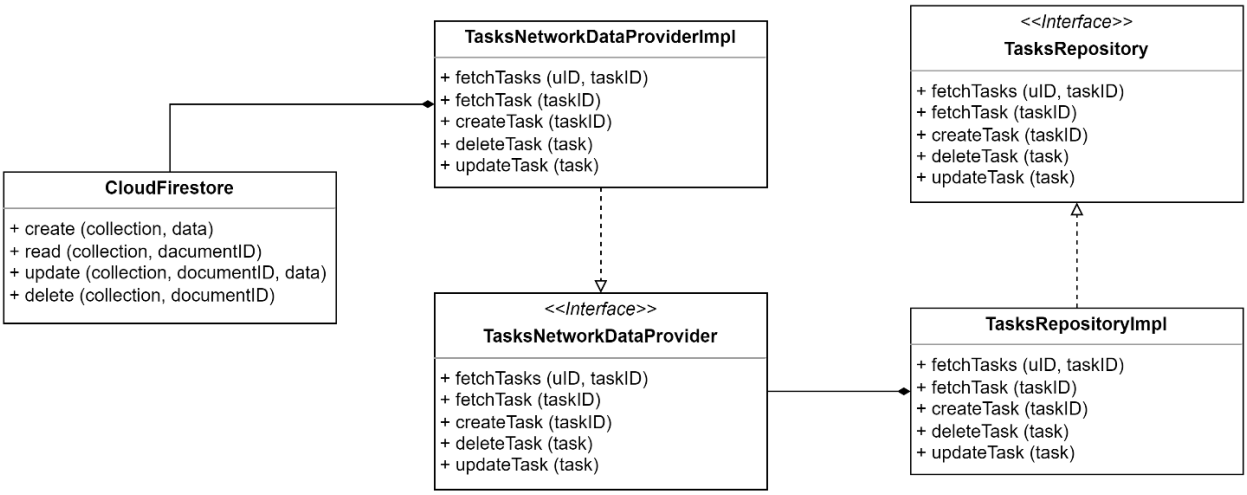

Рис. 2.2. Графічне представлення діаграми класів шару даних мобільного застосунку «TaskHub»

На діаграмі класів для шару даних (див. рис. 2.2) наведені наступні класи:

– «CloudFirebase»: хмарне сховище даних, яке зберігає задачі, які наявні в мобільному застосунку;

– «TaskNetworkDataProvider»: інтерфейсний клас, який відповідає за впровадження конструкторів класів ієрархією нижче, щоб не звертатися до конкретних реалізацій;

– «TaskNetworkDataProviderImpl»: відповідає за реалізацію інтерфейсу, який впроваджується у верхній ієрархії;

– «TasksRepository»: інтерфейсний клас, який відповідає за отримання даних із різних онлайн та офлайн джерел для отримання набору методів, а не реалізацій;

– «TasksRepositoryImpl»: відповідає за реалізацію методів, що впроваджується знизу ієрархії.

Клас «CloudFirebase» пов’язаний із «TaskNetworkDataProviderImpl» зв’язком композиції, що має строгу залежність часу існування. Клас «TaskNetworkDataProvider» пов’язаний із «TaskNetworkData ProviderImpl» зв’язком імплементації, а з «TasksRepositoryImpl» – композиції. «TasksRepository» пов’язаний із «TasksRepositoryImpl» зв’язком імплементації.

Зв’язок імплементації надає поняття відношення між двома класами, в якому перший є поставником, а другий клієнтом.

Діаграма класів методу управління станом BLoC наведена на рисунку 2.3.

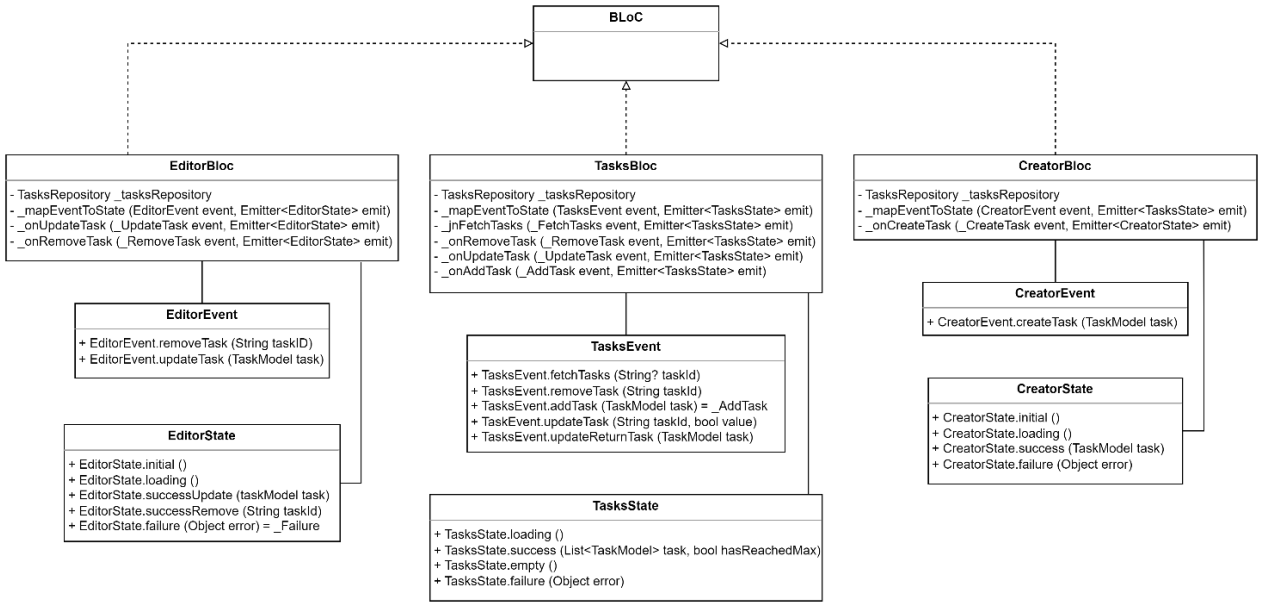

Рис. 2.3. Графічне представлення діаграми класів методу управління станом BLoC мобільного застосунку «TaskHub»

Надана діаграма класів для BLoC (див. рис. 2.3) містить наступні класи:

– «TasksState», «CreateState» та «EditorState»: абстрактні класи стану, від яких у подальшому йде наслідування та реалізація конкретних класів стану; відповідають за окремі стани бізнес-логіки;

– «TasksEvent», «CreateEvent» та «EditorEvent»: абстрактні класи подій, від яких у подальшому йде наслідування та реалізація конкретних класів подій; відповідає за окремі події задач бізнес-логіки;

– «TasksBLoC», «CreateBLoC» та «EditorBLoC»: конкретна реалізація бізнес-логіки, що має на вході події задач, а на виході стани задач;

– «BLoC»: абстрактна реалізація, що описує пристрої BLoC за замовчуванням.

У діаграмі наявні три блоки класів для кожного з екранів мобільного застосунку. «TasksState» та «TasksEvent», «CreateState» та «CreateEvent»,

«EditorState» та «EditorEvent» пов’язані зв’язками асоціації з класами «TasksBLoC», «CreateBLoC» та «EditorBLoC» відповідно, які в свою чергу пов’язані зв’язками наслідування з класом «BLoC».

Діаграма класів методу управління станом MobX наведена на рисунку 2.4.

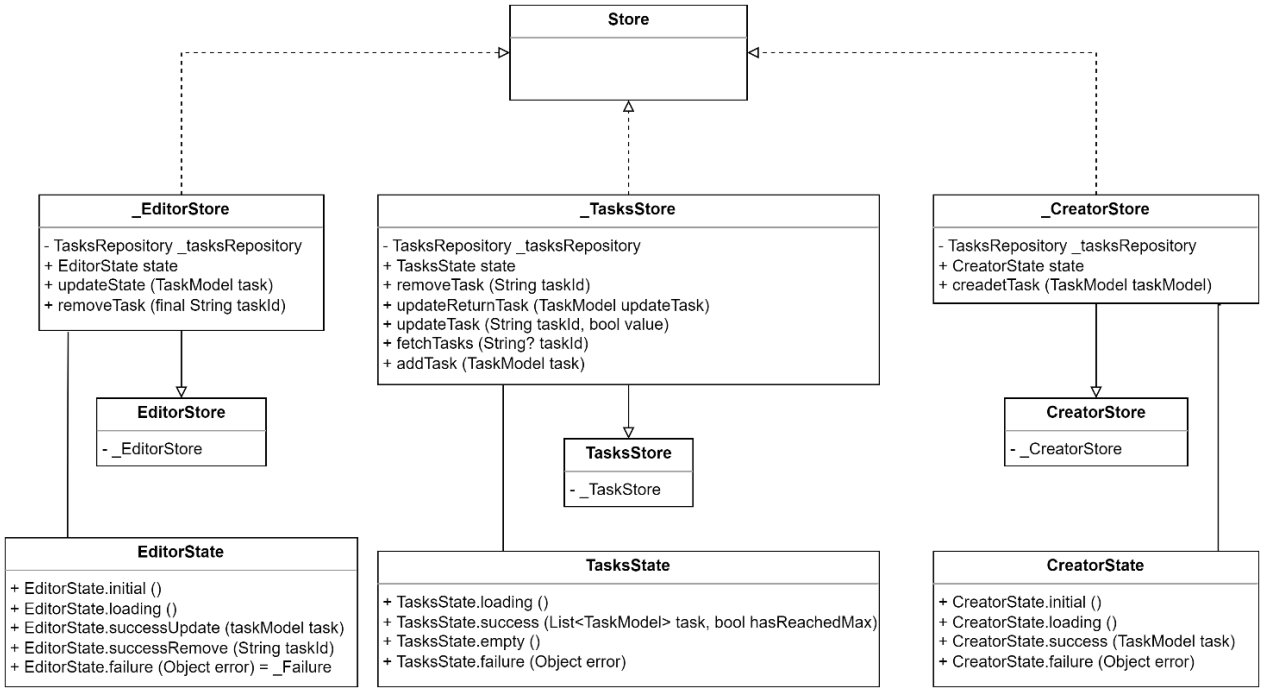

Рис. 2.4. Графічне представлення діаграми класів методу управління станом MobX мобільного застосунку «TaskHub»

На діаграмі (див. рис. 2.4) наведені наступні класи:

– «TasksStore»: сховище для керування станом задач;

– «_TasksStore», «_CreatorStore» та «_EditorStore»: сховища, що відповідають за зберігання операцій та стану;

– «CreatorStore»: сховище керування станом екраном створення задач;

– «EditorStore»: сховище керування станом екраном редагування задач;

– «TasksState», «CreateState» та «EditorState»: абстрактні класи стану, від яких у подальшому йде наслідування та реалізація конкретних класів стану; відповідають за окремі стани бізнес-логіки;

– «Store»: зберігає та керує станом застосунку.

«Store» пов’язаний зв’язками наслідування із «_TasksStore», «_CreatorStore» та «_EditorStore», які пов’язані тим ж зв’язками із «TasksStore», «CreatorStore» та «EditorStore» відповідно. «_TasksStore», «_CreatorStore» та «_EditorStore» пов’язані зв’язками асоціації із класами «TasksState», «CreateState» та «EditorState» відповідно.

Діаграма класів методу управління станом Redux наведена на рисунку 2.5.

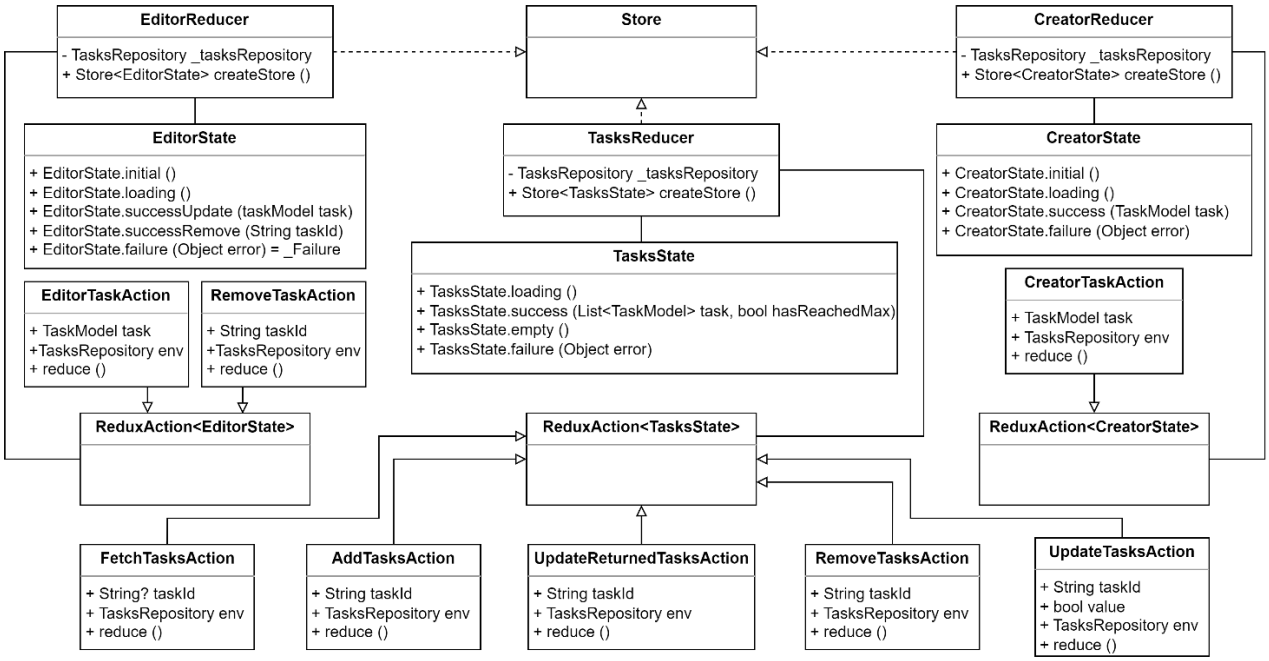

Рис. 2.5. Графічне представлення діаграми класів методу управління станом Redux мобільного застосунку «TaskHub»

На даній діаграмі (див. рис. 2.5) наведені наступні класи:

– «TasksReducer», «EditorReducer» та «CreatorReducer»: відповідальні за обробку подій та створення класу «Store»;

– «TasksState», «EditorState» та «CreatorState»: приватні класи стану, від яких йде успадкування та реалізація конкретних станів;

– «FetchTasksAction», «AddTasksAction», «UpdateReturnedTasksAction», «RemoveTasksAction» та «UpdateTasksAction»: події отримання, додання, оновлення зміненої задачі, видалення та оновлення задач відповідно;

– «EditorTaskAction», «RemoveTaskAction» та «CreatorTaskAction»: події зберігання, видалення та створення задач відповідно;

– «ReduxAction<EditorState>», «ReduxAction<CreatorAction>» та «ReduxAction<TasksState>»: формування відмітки класів як подій;

– «Store»: створення сховища Redux, в якому зберігається дерево станів застосунку.

За наданими діаграмами класів буде значно краще зорієнтуватися при розробці мобільного застосунку.

2.2.3 Проєктування діаграми компонентів

Діаграма компонентів допомагає структурувати компоненти, які будуть відноситися до системи. Діаграма компонентів є важливим інструментом для архітектурного проєктування та документування системи, а також для забезпечення чіткості взаємозв'язків між компонентами [34].

Для мобільного застосунку «TaskHub» було спроєктовано чотири діаграми. Перша є загальна для всіх методів управління станом, наведена на рисунку 2.6.

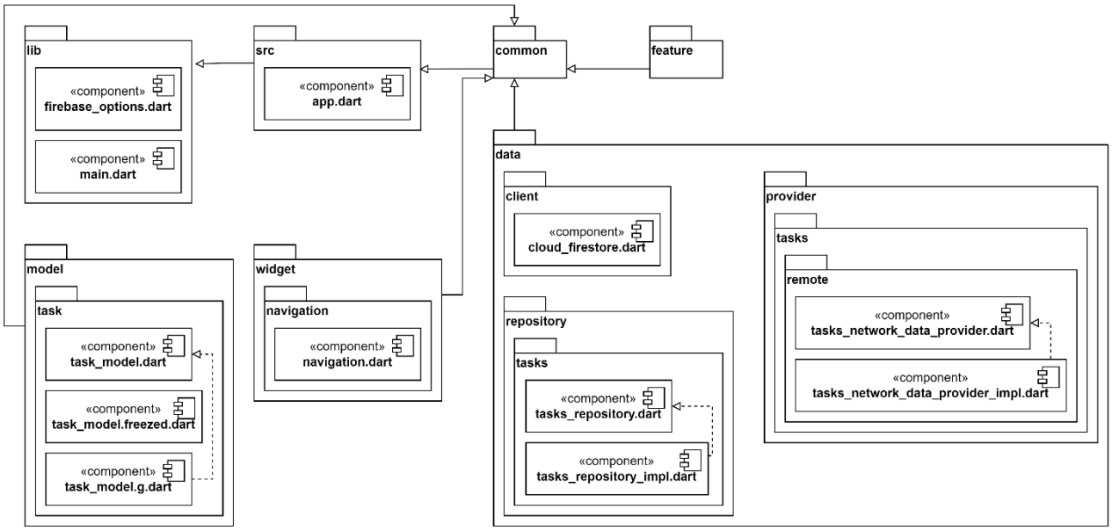

Рис. 2.6. Графічне представлення діаграми загальних компонентів мобільного застосунку «TaskHub»

За наданою діаграмою (див. рис. 2.6) наведені наступні репозиторії:

– «lib»: головний репозиторій, який вміщує в себе всі компоненти програми, головний файл та налаштування сховища даних;

– «src»: містить репозиторії «common» та «feature», а також файл первинного налаштування застосунку;

– «common»: містить репозиторії «model», «widget» та «data»;

– «model»: репозиторій, що містить файли зберігання про кожну сутність задачу;

– «widget»: репозиторій, що містить у собі елементи, пов’язані із інтерфейсом;

– «data»: репозиторій, який містить у собі отримання та відправлення даних;

– «feature»: репозиторій, який містить у собі окремі елементи логіки та візуалізації.

Друга діаграма компонентів направлена на репозиторій «feature», що відноситься до методу управління станом BLoC. Вона наведена на рисунку 2.7.

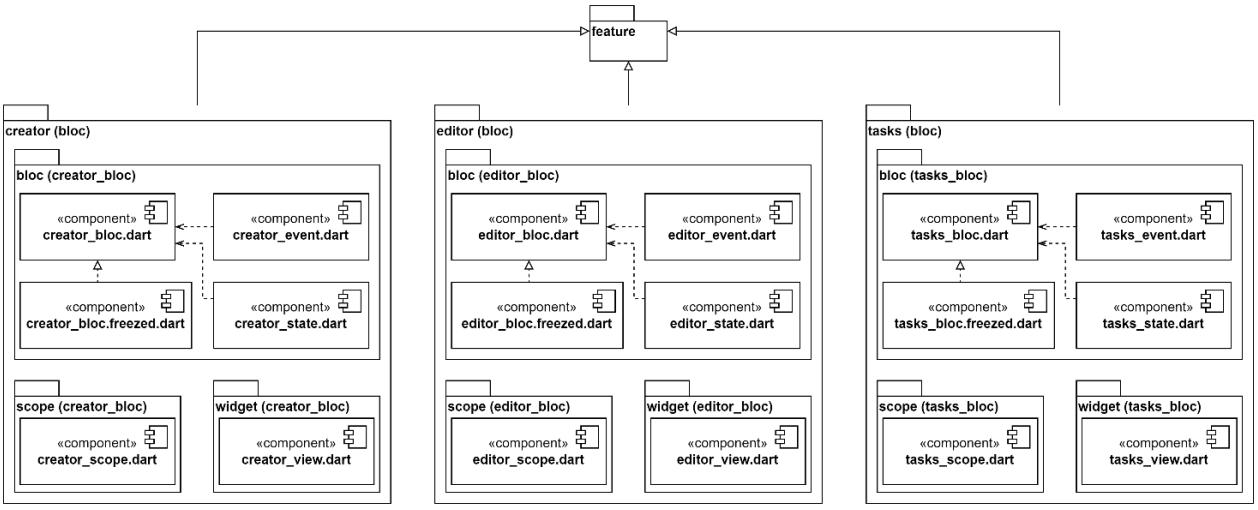

Рис. 2.7. Графічне представлення діаграми компонентів репозиторію для методу управління станом BLoC мобільного застосунку «TaskHub»

Даний репозиторій (див. рис. 2.7) для методу управління станом BLoC містить такі репозиторії:

– «creator (bloc)»: репозиторій, необхідний для реалізації екрану створення задач;

– «editor (bloc)»: репозиторій, необхідний для реалізації екрану редагування задач;

– «tasks (bloc)»: репозиторій, необхідний для реалізації екрану списку задач.

Третьою діаграмою є діаграма компонентів направлена на репозиторій «feature», що відноситься до методу управління станом MobX. Вона наведена на рисунку 2.8.

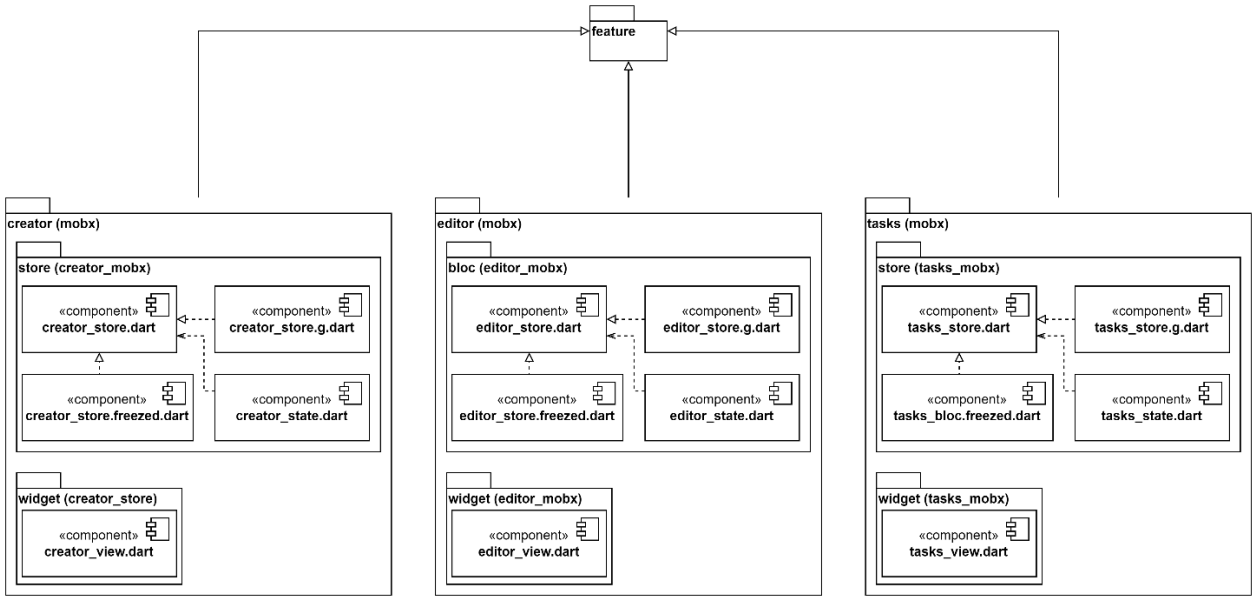

Рис. 2.8. Графічне представлення діаграми компонентів репозиторію для методу управління станом MobX мобільного застосунку «TaskHub»

Четвертою діаграмою є діаграма компонентів направлена на репозиторій «feature», що відноситься до методу управління станом Redux. Вона наведена на рисунку 2.9.

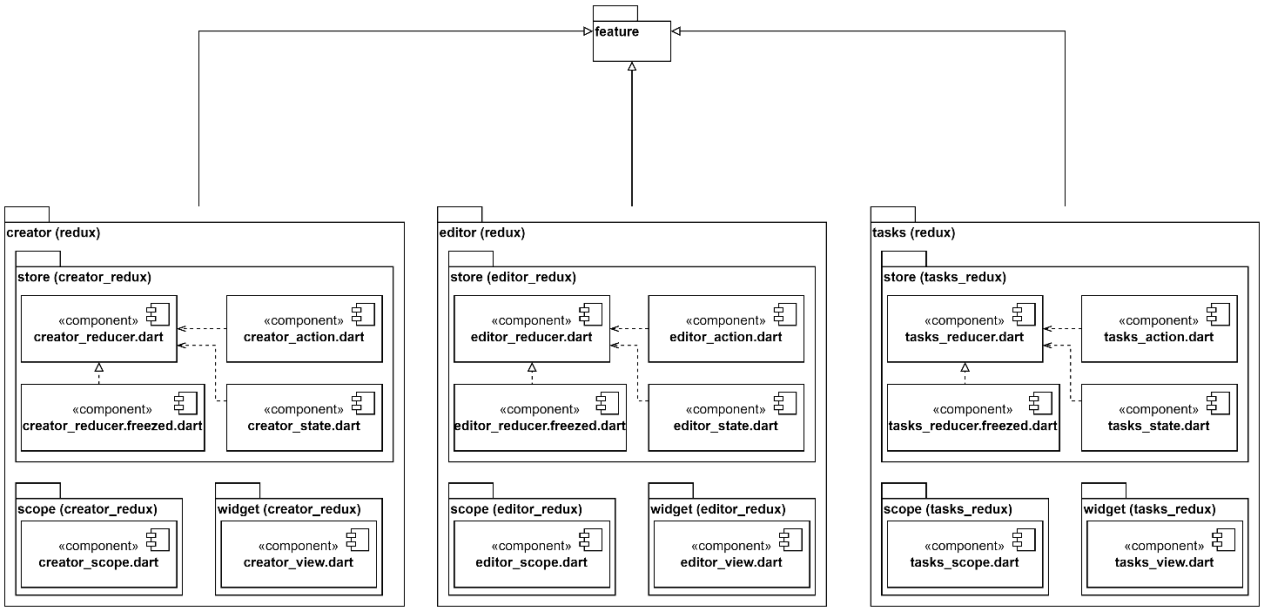

Рис. 2.9. Графічне представлення діаграми компонентів репозиторію для методу управління станом Redux мобільного застосунку «TaskHub»

На діаграмах (див. рис. 2.8-2.9) наведені ті ж репозиторії, що й на попередній діаграмі (див. рис. 2.7). Відмінність їх у тому, що для кожного із них використовуються інші файли іншими програмними алгоритмами. Таким чином, на наданих діаграмах компонентів, як загальній, так і для кожного методу управління станом наведені структуровані репозиторії, файли та компоненти, які буде вміщувати в себе програма.

2.2.4. Проєктування макетів екрану застосунку

Створення макетів екрану є ще одним із найважливіших етапів перед безпосередньою розробкою мобільного застосунку. На них зображується схематичний користувальницький інтерфейс. Дане проєктування необхідно для того, щоб під час розробки зрозуміти, які саме кнопки та їх назви будуть і де вони розташовані, які поля необхідні для створення та редагування задач, як виглядатиме список задач тощо.

Макети екранів «Список задач», «Створення задач» та «Редагування задач» наведений на рисунку 2.10.

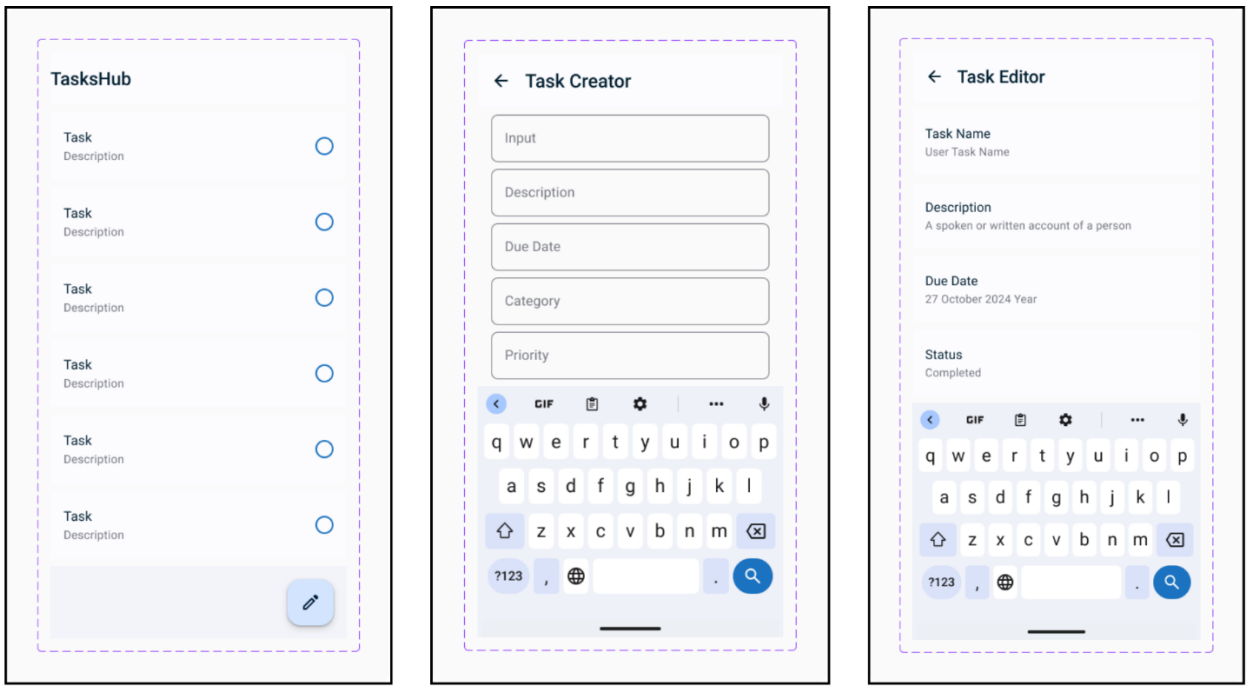

Рис. 2.10. Графічне представлення спроєктованих макетів екранів мобільного застосунку «TaskHub»

На макеті екрану списку задач (див. рис. 2.10) наведені такі елементи інтерфейсу:

– найменування списку задач;

– список задач;

– кнопки для відмітки виконання чи невиконання задач;

– кнопка для створення задач;

– кнопка редагування задач шляхом натиснення на самі задачі.

На макеті екрану створення задач (див. рис. 2.10) наведені наступні елементи інтерфейсу:

– найменування екрану;

– кнопка повернення на попередній екран;

– текстові поля найменування та опису задачі;

– поле вибору дати закінчення задачі, категорії та пріоритету.

На макеті екрану редагування задач (див. рис. 2.10) наведені наступні елементи інтерфейсу:

– найменування екрану;

– кнопка повернення на попередній екран;

– редагування текстових полів найменування та опису задачі;

– редагування полів дати закінчення задачі, пріоритету та категорії.

За наданими макетами користувальницького інтерфейсу буде створений мобільний застосунок «TaskHub». У реалізованих екранах можуть змінитися кольори, та місце розташування полів або кнопок.

2.3. Методи управління станом фреймворку Flutter

Управління станом Flutter включає в себе багато речей, від виклику мережевої функції застосунку асинхронно, додавання мережевих бібліотек, парсингу даних, їх оновлення в головному потоці інтерфейсу тощо. Кожен розробник використовує різні підходи для виконання однієї і тієї ж задачі, оскільки не може бути одного ідеального рішення для всіх випадків використання. Однак ефективність виконаної роботи залежить від вибору розробника, його знань про цей підхід і від того, наскільки йому комфортно з ним працювати [35-36].

Коли відбувається створення інтерфейсу ПЗ, дуже важливо керувати реакцією на кожну подію, ініційовану діями, що виконуються в ньому. Тобто, потрібно керувати станом елементів інтерфейсу, які змінюються через взаємодію з кінцевим користувачем. Керуючи станом інтерфейсу, можна зіткнутися з необхідністю розбити великий інтерфейс на кілька менших компонентів, забезпечуючи при цьому ефективний зв'язок між ними. Крім того, усі ці компоненти інтерфейсу повинні бути в курсі стану програми в будь-який час. Найпопулярнішими підходами управління станом є BLoC, MobX та Redux [36].

2.3.1. Метод управління станом BLoC

BLoC був створений, щоб відокремити бізнес-логіку від рівня представлення, таким чином полегшуючи розробникам більш ефективне використання коду. BLoC – це бібліотека управління станом, яка допомагає розробникам реалізувати дизайн у своєму Flutter застосунку. Це означає, що розробники повинні знати про стан програми щоразу, коли відбувається взаємодія з інтерфейсом користувача. Наприклад, під час операції збору даних ПП повинен відображати на екрані анімацію завантаження даних. Тому дизайнери застосунків повинні враховувати кожен варіант використання, який має стан, що виникає з нього, щоб легко відокремити рівень презентації від бізнес-логіки і, таким чином, спростити управління станами в ПЗ [37-38].

Перевагами використання методу управління станом BLoC [38] є:

– відокремлення презентаційного шару від бізнес-шару;
– спрощення тестування застосунків;
– краща продуктивність для великих обсягів даних;
– базування на реактивному програмуванні;
– забезпечення безпечного виконання операцій на різних потоках.

Недоліками використання методу управління станом BLoC [38] є:

– призначений для невеликих або середніх за розміром проєктах;
– надмірна кількість коду.

BLoC – це просунутий клас. Він покладається на подію для запуску стану, а не на функцію [38]. На рисунку 2.11 наведено схему роботу BLoC.

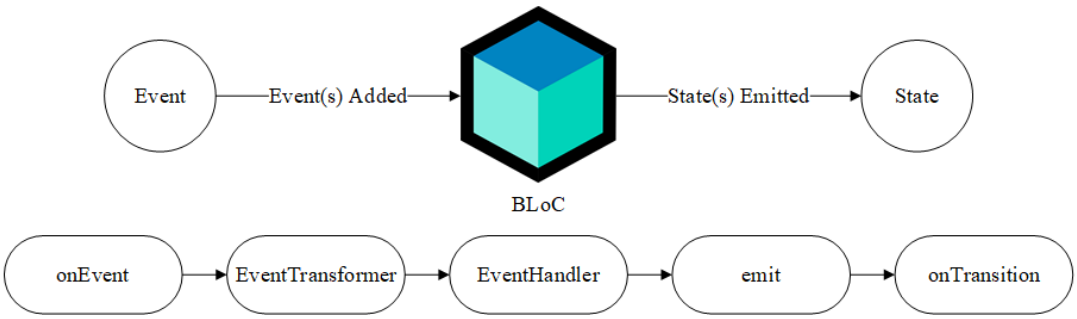

Рис. 2.11. Графічне представлення схеми роботи BLoC

Необхідно зазначити, що BLoC не завжди є оптимальним вибором для кожного проєкту, і його використання вимагає розуміння концепцій реактивного програмування та архітектурних принципів. Однак, за правильного використання, він може значно спростити та покращити процес розробки застосунків на Flutter.

2.3.2. Метод управління станом MobX

MobX – це бібліотека управління станами, яка використовує реактивне програмування для управління станами у Flutter застосунках. Вона добре підходить для складного ПЗ з великою кількістю станів і надає простий, легкий у використанні API. Вона використовує спостережувані дані та реакції для автоматичного оновлення користувацького інтерфейсу у відповідь на зміни стану програми. У Flutter MobX може допомогти спростити управління станами, зменшивши кількість шаблонного коду, необхідного для обробки змін стану. Це досягається завдяки використанню реактивної моделі програмування, яка автоматично оновлює користувацький інтерфейс щоразу, коли змінюється стан [39-41].

Перевагами методу управління станом MobX [41] є:

– заснований на реактивному програмуванні, що дозволяє легко відстежувати зміни стану та автоматично оновлювати користувальницький інтерфейс;

– потреба меншої кількості коду в порівнянні з іншими станами.

Недоліками методу управління станом MobX [41] є:

– обмежена підтримка для різних платформ;

– недостатньо широка адоптація.

Схема роботи MobX наведена на рисунку 2.12.

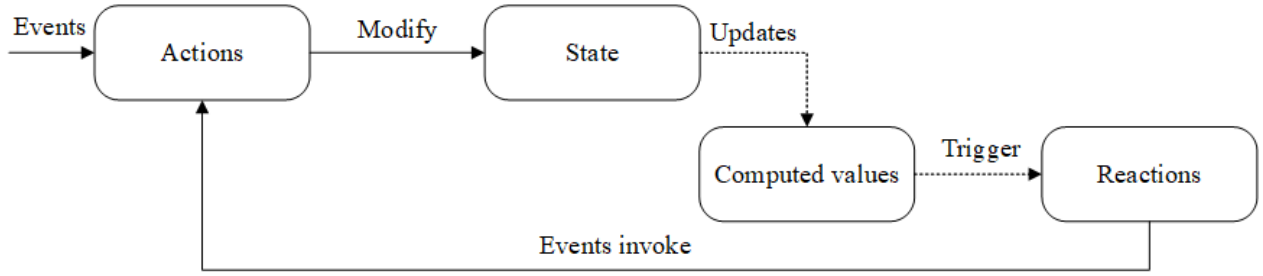

Рис. 2.12. Графічне представлення схеми роботи MobX

Загалом, пакет MobX може допомогти зробити управління станами у Flutter більш ефективним і простим у підтримці, автоматизуючи більшу частину ручної роботи, пов'язаної з синхронізацією інтерфейсу зі станом програми.

2.3.3. Метод управління станом Redux

Redux – це підхід до управління станами за допомогою контейнерів станів. Він допомагає успішно розподіляти дані між віджетами в повторюваній манері, дозволяє керувати станом застосунку за допомогою односпрямованого потоку. Redux дозволяє структурувати застосунок таким чином, що стан витягується з центрального сховища. Він піднімає вгору стан, дію та редуктор, які складають програму. Тут дані з централізованого сховища можуть бути доступні будь-якому віджету. Отже, немає необхідності передавати дані через ланцюжок віджетів у дереві. Redux покликаний запобігти цій помилці, роблячи стан незмінним і забезпечуючи односпрямований потік даних. Цей підхід добре зарекомендував себе в синхронній ситуації, але може викликати проблеми, коли ви виконуєте завдання асинхронно [42-44].

Перевагами методу управління станом Redux [44] є:

– забезпечення суворого та передбачуваного потоку управління станом;

– гарна масштабованість для крупних і складних проєктів.

Недоліками методу управління станом Redux [44] є:

– використання централізованого сховища, в якому зберігається весь стан програмного продукту;

– легка інтеграція з інструментами для збереження та відновлення історії стану застосунку;

– не надання вбудованого механізму для управління асинхронними операціями.

Схема роботи Redux наведена на рисунку 2.13.

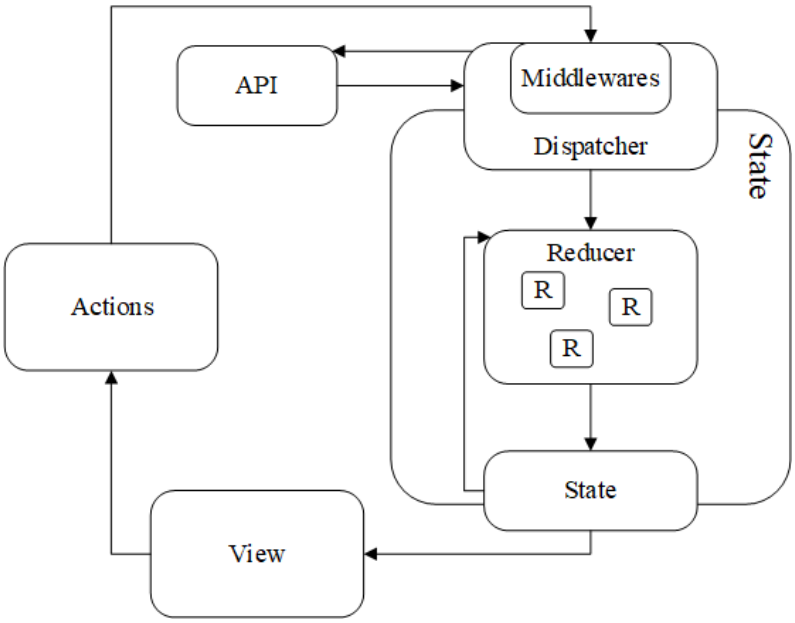

Рис. 2.13. Графічне представлення схеми роботи Redux

Вибір Redux залежить від вимог проєкту і досвіду розробників. Він підходить для великих і складних застосунків, де передбачуваність і масштабованість відіграють важливу роль, але може бути надлишковим для менших проєктів.

2.4. Засоби реалізації мобільного застосунку

Вибір інструментарію для розробки мобільного застосунку є важливим етапом, адже він надає спрощення розробки проєкту, кращу продуктивність, швидкість реалізації ПП тощо. Вибір засобів також залежить від таких факторів, як платформа розробки, навички та досвід розробників, вимоги до мобільного

застосунку, строки виконання, тестування та відладки, прототипування тощо [45].

2.4.1. Мова програмування та фреймворк

В якості мови програмування було обрано Dart. Його перевагами [46-47] є:

– компіляція в нативний код: забезпечення високої продуктивності, особливо у парі із фреймворком Flutter;

– сильна типізація: попередження множини помилок на етапі компіляції та покращення надійності коду;

– гаряче перезавантаження: прискорення процесу розробки та відладки;

– сучасний синтаксис;

– збір сміття;

– асинхронне програмування.

В якості фреймворку було обрано Flutter. Його перевагами є:

– підтримка кросплатформності;

– висока продуктивність: використання компіляції в нативний код, що забезпечує високу продуктивність та швидке виконання застосунків;

– гаряче перезавантаження;

– відкритий вихідний код;

– розширення та адаптація.

Зазвичай, Dart і Flutter йдуть у парі, тому вони й були обрані разом. Вони є потужним інструментарієм для розробки кросплатформних мобільних застосунків. У якості методів управління станом були обрані BLoC, MobX і Redux, переваги та недоліки яких було описано вище.

2.4.2. Середовище розробки

В якості середовища розробки було обрано Android Studio. Вона має такі переваги [48-49]:

– потужні інструменти розробки, відладки та тестування застосунків, включаючи інтегрований емулятор, графічний дизайнер інтерфейсу тощо;

– інструменти аналізу продуктивності застосунку;

– підтримка різних пристроїв;

– використання вбудованого емулятора для розробки кросплатформних застосунків;

– інтеграція зі специфічними сервісами або фунціоналом і їх тестування;

– інтуїтивно зрозумілий інтерфейс.

Дане середовище розробки має й інші переваги, але саме вищевказані є найпотужнішими та найвагомішими. Вони роблять Android Studio ефективним інструментом у розробці застосунків.

2.4.3. Метрики порівняння методів управління станом

Метриками, обраними для порівняння методів управління станом BLoC, MobX, Redux є:

– вимір швидкодії часу виконання операції: у різних станах застосунку відбувається вимір часу виконання різних операцій, так як запити до серверу, обробка даних або перемальовування віджетів;

– вимір розмірів коду в залежності від методу управління станом, з та без урахування загальної функції;

– проведення аналізу графіків розподілення пам’яті в застосунку для кожного із методів управління станом.

Обрані метрики допоможуть провести аналіз порівняння методів управління станом та визначити їх показники.

2.5. Висновки за розділом 2

За другим розділом були виконані наступні завдання:

– описані функціональні вимоги мобільного застосунку;

– спроєктована діаграма варіантів використання;

– спроєктовані діаграми класів для кожного методу управління станом;

– спроєктовані макети екранів мобільного застосунку;

– описані методи управління станом;

– виконаний обґрунтований вибір інструментів розробки мобільного застосунку;

– описані метрики порівняння методів управління станом.

Після виконаних вищеописаних завдань можна переходити до розробки програмного забезпечення, а також порівнянню результатів дослідження.

РОЗДІЛ 3
ПРОГРАМНА РЕАЛІЗАЦІЯ МОБІЛЬНОГО ЗАСТОСУНКУ НА ОСНОВІ МЕТОДІВ УПРАВЛІННЯ СТАНОМ ТА АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ

3.1 Програмна реалізація шару «Widget Layer»

«Widget Layer» у мобільному застосунку відіграє роль організації користувальницького інтерфейсу. Flutter використовує віджети як основні будівельні блоки для побудови користувальницького інтерфейсу. Шар віджетів являє собою структуру, в якій віджети компонуються та взаємодіють один з одним, утворюючи ієрархію віджетів – дерево елементів. Воно представляє собою ієрархію віджетів, де кожен з них є частиною користувальницького інтерфейсу, починаючи з простих елементів, таких як кнопки або текстові поля, закінчуючи більш складними, такими як списки. Для кожного з мобільних застосунків на трьох методах управління станом було створено по три дерева елементів для кожного з екранів (див. рис. А.1-А.9).

При побудові застосунків Flutter кожен віджет будується у відповідності з його описом, починаючи з кореневого віджету та рекурсивно оброблюючи дочірні.

На початку створення шару віджетів були імпортовані необхідні бібліотеки для роботи. Реалізація екрану «Список задач» організовано навколо класу «CreatorView», який є віджетом із збереженням стану та класу «_CreatorViewState». Основна логіка віджету включає створення текстових контролерів для полів введення, звернення до поточного стану для відстеження змін і побудову інтерфейсу з використанням віджетів, таких як «AppBar» і «Stack». Також для кожного методу управління станом використовується обробка станів, що дозволяє визначити дії у відповідь на його зміни (див. рис. 3.1).

```
  @override
  State<CreatorView> createState() => _CreatorViewState();
}

class _CreatorViewState extends State<CreatorView> {
  @override
  Widget build(BuildContext context) {
    final nameController = TextEditingController();
    final descriptionController = TextEditingController();
    final dueDateController = TextEditingController();
    final categoryController = TextEditingController();
    final priorityController = TextEditingController();
```

Рис. 3.1. Графічне представлення програмної реалізації екрану «Список задач»

Візуальна реалізація екрану «Створення задач» побудована на основі класу «TasksView». Основні моменти включають використання віджетів «RefreshIndicator» для оновлення списку завдань, «CustomScrollView», «SliverList.builder» для відображення списку завдань і «Dismissible» для можливості видалення завдання свайпом. Також присутні «GestureDetector» для навігації до редактора задач і «FloatingActionButton» для додавання нових завдань. Також реалізовано оновлення стану залежно від поточного стану застосунку, що впливає на відображення відповідного інтерфейсу (див. рис. 3.2-3.4).

```
child: GestureDetector(
  onTap: () async {
    HapticFeedback.vibrate();
    Navigator.pushNamed(
      context,
      RouteNames.editor,
      arguments: state.tasks[index],
    ).then(
      (value) {
        if (value == null) return;
        if (value is String) {
          TasksScope.removeTask(context, value);
        } else if (value is TaskModel) {
          TasksScope.updateReturnTask(context, value);
        }
```

Рис. 3.2. Графічне представлення програмної реалізації екрану «Створення задач» віджету «GestureDetector»

```
return RefreshIndicator(
  onRefresh: () async => TasksScope.fetchTasks(context, null),
  child: Scaffold(
    appBar: AppBar(
      title: const Text('TaskHub'),
    ),
    body: Center(
      child: state.map(
        loading: (state) => const CircularProgressIndicator(),
        success: (state) => CustomScrollView(
          slivers: [
            SliverList.builder(
              itemCount: state.tasks.length,
              itemBuilder: (BuildContext context, int index) {
                if (index >= state.tasks.length - 1) {
                  TasksScope.fetchTasks(context, state.tasks[index].taskId);
                }
                return Padding(
                  padding: const EdgeInsets.symmetric(horizontal: 8.0),
                  child: Dismissible(
                    key: Key(state.tasks[index].taskId),
                    onDismissed: (direction) {
                      HapticFeedback.vibrate();
                      TasksScope.removeTask(
                          context, state.tasks[index].taskId);
                    },
```

Рис. 3.3. Графічне представлення програмної реалізації екрану «Створення задач» віджетів «RefreshIndicator», «CustomScrollView» і «Dismissible»

```
floatingActionButtonLocation: FloatingActionButtonLocation.endContained,
floatingActionButton: FloatingActionButton(
  onPressed: () async {
    HapticFeedback.vibrate();
    Navigator.pushNamed(context, RouteNames.creator).then((task) {
      if (task == null) return;
      TasksScope.addTask(context, task as TaskModel);
    });
  },
  child: const Icon(Icons.create),
),
```

Рис. 3.4. Графічне представлення програмної реалізації екрану «Створення задач» віджету «FloatingActionButton»

Візуальна реалізація екрану «Редагування задач» побудована на основі класу «EditorView». Він відображає форми редагування задач, ініціалізує стан віджету з даними задачі та керує їх змінами, має контролери текстових полів для введення різних параметрів задачі таких як назва, опис, дата, категорія, пріоритет (див. рис. 3.5). Віджет також містить метод для перевірки змін у параметрах завдання та метод для побудови UI, який реагує на стан керування завданнями

та відображає відповідні елементи інтерфейсу. Крім того, було визначено підвіджети для відображення статусу завдання та введення інформації про нього, як-от опис, пріоритет, назва, категорія та термін виконання.

```
class EditorView extends StatefulWidget {
  final TaskModel task;

  const EditorView({
    Key? key,
    required this.task,
  }) : super(key: key);

  @override
  State<EditorView> createState() => _EditorViewState();
}

class _EditorViewState extends State<EditorView> {
  late TextEditingController nameController;
  late TextEditingController descriptionController;
  late TextEditingController dueDateController;
  late TextEditingController categoryController;
  late TextEditingController priorityController;
  late bool status;
```

Рис. 3.5. Графічне представлення програмної реалізації екрану «Редагування задач»

Для впровадження методів управління станом у візуальні компоненти необхідний клас «ScreenFactory» та його функції в окремому файлі «navigation.dart» (див. рис. 3.6).

```
class ScreenFactory {
  Widget makeTasks() {
    final firestore = FirebaseFirestore.instance;
    return BlocProvider<TasksBloc>(
      create: (_) => TasksBloc(
        tasksRepository: TasksRepositoryImpl(
          tasksNetworkDataProviderImpl: TasksNetworkDataProviderImpl(
            cloudFirestore: CloudFirestore(firestore: firestore),
          ),
        ),
      )..add(const TasksEvent.fetchTasks()),
      child: const TasksView(),
    );
  }
```

Рис. 3.6. Графічне представлення програмного коду класу «ScreenFactory»

3.2. Програмна реалізація «Business Logic Layer»

Шар бізнес-логіки мобільного застосунку було відтворено трьома методами управління станом, кожен із яких є унікальним. Програмна реалізація відбувалася створенням власних алгоритмів, із малим відсотком запозиченого коду, такого як поширені методи та функції для відтворення застосунку [50].

3.2.1. Програмна реалізація методу управління BLoC

Для обробки станів та подій методу управління станом BLoC на екрану «Список задач» було використано пакет «flutter_bloc». Клас «TasksBloc» виконує обробку різних подій, таких як завантаження, видалення, оновлення та додавання задач (див. рис. 3.7).

```
class TasksBloc extends Bloc<TasksEvent, TasksState> {
  final TasksRepository _tasksRepository;

  TasksBloc({
    required TasksRepository tasksRepository,
  })  : _tasksRepository = tasksRepository,
        super(const TasksState.loading()) {
    on<TasksEvent>(
      (event, emit) async => event.map<Future<void>>(
        fetchTasks: (event) => _onFetchTasks(event, emit),
        removeTask: (event) => _onRemoveTask(event, emit),
        updateTask: (event) => _onUpdateTask(event, emit),
        addTask: (event) => _onAddTask(event, emit),
        updateReturnTask: (event) => _onUpdateReturnTask(event, emit),
      ),
      transformer: droppable(),
    );
  }
```

Рис. 3.7. Графічне представлення програмної реалізації екрану «Список задач» методу управління станом BLoC класу «TasksBloc»

При створенні екземпляру BLoC, він ініціалізується репозиторієм задач та початковим станом «loading()». Далі встановлюються обробники кожного типу події, використовуючи метод «on», і визначений метод «_onFetchTasks» для обробки події завантаження задач. У методі «_onFetchTasks» здійснюється

виклик репозиторію для отримання завдань за вказаними параметрами. Якщо поточний стан: «_Success», то виконується перевірка, чи досягнуто ліміт задач, і список задач оновлюється. Інакше створюється новий стан «_Empty()» чи «_Success» (див. рис. 3.8).

```
Future<void> _onFetchTasks(
  _FetchTasks event,
  Emitter<TasksState> emit,
) async {
  try {
    final uid = FirebaseAuth.instance.currentUser!.uid;
    final tasks = await _tasksRepository.fetchTasks(
      uid: uid,
      taskId: event.taskId,
    );
    if (state is _Success) {
      if ((state as _Success).hasReachedMax) return;
      emit(
        tasks.isEmpty
            ? (state as _Success).copyWith(hasReachedMax: true)
            : (state as _Success).copyWith(
                tasks: (state as _Success).tasks + tasks.toList(),
                hasReachedMax: false,
              ),
      );
    } else {
      final newState = tasks.isEmpty
          ? const _Empty()
          : _Success(tasks: tasks.toList(), hasReachedMax: false);
      emit(newState);
    }
  } on Object catch (error) {
    emit(_Failure(error: error));
    rethrow;
  }
}
```

Рис. 3.8. Графічне представлення програмної реалізації екрану «Список задач» методу управління станом BLoC методу «_onFetchTasks»

Аналогічно, у методах «_onRemoveTask», «_onUpdateTask», «_onAddTask» та «_onUpdateReturnTask» виконуються операції видалення, оновлення, додавання та оновлення повернутої задачі відповідно.

Головним завданням BLoC на екрані «Створення задач» є обробка події «createTask». BLoC ініціалізується екземпляром «TasksRepository», який служить для взаємодії з даними (див. рис. 3.9).

```
class CreatorBloc extends Bloc<CreatorEvent, CreatorState> {
  final TasksRepository _tasksRepository;

  CreatorBloc({
    required TasksRepository tasksRepository,
  })  : _tasksRepository = tasksRepository,
        super(const CreatorState.initial()) {
    on<CreatorEvent>(
      (event, emit) => event.map<Future<void>>(
        createTask: (event) => _onCreateTask(event, emit),
      ),
    );
  }

  Future<void> _onCreateTask(
    _CreateTask event,
    Emitter<CreatorState> emit,
  ) async {
    try {
      emit(const _Loading());
      final task = await _tasksRepository.createTask(task: event.task);
      emit(_Success(task: task));
    } on Object catch (error) {
      emit(_Failure(error: error));
      emit(const _Initial());
      rethrow;
    }
  }
}
```

Рис. 3.9. Графічне представлення програмної реалізації екрану «Створення задач» методу управління станом BLoC

Коли відбувається подія «createTask», викликається приватний метод «_onCreateTask», у якому обробляється логіка створення задачі. Усередині «_onCreateTask» відправляється стан «_Loading()», щоб показати розпочатий процес створення задачі. Далі викликається метод «_tasksRepository.createTask», який повертає створену задачу. Якщо створення пройшло успішно, відправляється стан «_Success». У разі помилки надсилається стан «_Failure», а потім повертається початковий стан «_Initial()»та повторно викидається виняток (див. рис. 3.9).

Для екрану «Редагування задач» клас «EditorBloc» обробляє події, пов’язані з оновленням і видаленням. Під час створення екземпляра BLoC він ініціалізується репозиторієм задач і початковим станом «initial()». Потім встановлюються обробники для подій оновлення та видалення завдань (див. рис. 3.10).

```
class EditorBloc extends Bloc<EditorEvent, EditorState> {
  final TasksRepository _tasksRepository;

  EditorBloc({
    required TasksRepository tasksRepository,
  })  : _tasksRepository = tasksRepository,
        super(const EditorState.initial()) {
    on<EditorEvent>(
      (event, emit) => event.map<Future<void>>(
        removeTask: (event) => _onRemoveTask(event, emit),
        updateTask: (event) => _onUpdateTask(event, emit),
      ),
    );
  }

  Future<void> _onUpdateTask(
    _UpdateTask event,
    Emitter<EditorState> emit,
  ) async {
    try {
      emit(const _Loading());
      await _tasksRepository.updateTask(task: event.task);
      emit(_SuccessUpdate(task: event.task));
    } on Object catch (error) {
      emit(_Failure(error: error));
      emit(const _Initial());
      rethrow;
    }
  }

  Future<void> _onRemoveTask(
    _RemoveTask event,
    Emitter<EditorState> emit,
  ) async {
    try {
      emit(const _Loading());
      await _tasksRepository.deleteTask(taskId: event.taskId);
      emit(_SuccessRemove(taskId: event.taskId));
    } on Object catch (error) {
      emit(_Failure(error: error));
      emit(const _Initial());
      rethrow;
    }
  }
}
```

Рис. 3.10. Графічне представлення програмної реалізації екрану «Редагування задач» методу управління станом BLoC

Метод «_onUpdateTask» обробляє подію оновлення завдання. Усередині блоку «try-catch» відбувається емітинг станів, пов'язаних із завантаженням, успішним оновленням або помилкою. Якщо оновлення пройшло успішно, створюється стан «_SuccessUpdate» з оновленим завданням. Метод «_onRemoveTask» аналогічно обробляє подію видалення завдання. У блоці «try-catch» також здійснюється емітинг станів, пов’язаних із завантаженням, успішним видаленням або помилкою. Якщо видалення пройшло успішно,

створюється стан «_SuccessRemove» з ідентифікатором видаленого завдання. Обидва методи забезпечують оброблення помилок та емітинг відповідних станів, що забезпечує надійне опрацювання сценаріїв оновлення та видалення завдань у застосунку (див. рис. 3.10).

3.2.2. Програмна реалізація методу управління MobX

Реалізація методу управління станом MobX представляє собою розробку сховища «TaskStore» (див. рис. 3.11) для екрану «Список задач». «_tasksRepository» є екземпляром репозиторію задач, який надає методи для роботи з даними.

```
class TasksStore = _TasksStore with _$TasksStore;

abstract class _TasksStore with Store {
  final TasksRepository _tasksRepository;

  _TasksStore({
    required TasksRepository tasksRepository,
  }) : _tasksRepository = tasksRepository;

  @observable
  TasksState state = const TasksState.loading();

  @action
  Future<void> removeTask(final String taskId) async {
    try {
      await _tasksRepository.deleteTask(taskId: taskId);
      final updatedTasks = (state as _Success)
          .tasks
          .where((task) => task.taskId != taskId)
          .toList();

      if (updatedTasks.isEmpty) {
        state = const _Empty();
      } else {
        state = (state as _Success).copyWith(tasks: updatedTasks);
      }
    } on Object catch (error) {
      state = _Failure(error: error);
      rethrow;
    }
  }
```

Рис. 3.11. Графічне представлення програмної реалізації екрану «Список задач» методу управління станом MobX класу «TaskStore» та дії «@action removeTask»

Дії «@action removeTask», «@action updateReturnTask», «@action updateTask», «@action fetchTasks» та «@action addTask» необхідні для видалення задачі, оновлюючи стан у відповідності з результатами операції, оновлення задачі та стану, статусу виконання задачі та стану, завантаження задачі та оновлення стану та додавання нової задачі та оновлення стану відповідно. Для кожної дії була передбачена обробка помилок з використанням «try-catch», де при виникненні помилок стану оновлюється на «_Failure», виняток повторно реалізується. При додаванні нової задачі в стан відбувається сортування списку задач за датою завершення (див. рис. 3.11).

Взагалі «TasksStore» забезпечує управління станом задач, а MobX автоматично обробляє реакцію на зміни змінних і оновлення інтерфейсу користувача.

Реалізація екрану «Створення задач» представляє собою сховище «CreatorStore» (див. рис. 3.12).

```
class CreatorStore = _CreatorStore with _$CreatorStore;

abstract class _CreatorStore with Store {
  final TasksRepository _tasksRepository;

  _CreatorStore({
    required TasksRepository tasksRepository,
  }) : _tasksRepository = tasksRepository;

  @observable
  CreatorState state = const CreatorState.initial();

  @action
  Future<void> createdTask(final TaskModel taskModel) async {
    try {
      state = const _Loading();
      final task = await _tasksRepository.createTask(task: taskModel);
      state = _Success(task: task);
    } on Object catch (error) {
      state = _Failure(error: error);
      state = const _Initial();
      rethrow;
    }
  }
}
```

Рис. 3.12. Графічне представлення програмної реалізації екрану «Створення задач» методу управління станом MobX

Клас «CreatorStore» є основним класом сховища, що об’єднує анотації Freezed и MobX. Декоратор «@observable» означає, що змінна «state» може бути спостережуваною, а зміни в ній будуть відстежуватися автоматично. Дія «@action createdTask», яка необхідна для створення нової задачі та оновлює стан у відповідності з результатами операції. Для обробки помилок використовується «try-catch». У разі виникнення помилки стан оновлюється на «_Failure», потім на «_Initial», і виняток повторно порушується (див. рис. 3.12).

Для реалізації екрану «Редагування задач» було створено відповідне сховище та однойменний клас «EditorStore» (див. рис. 3.13).

```
class EditorStore = _EditorStore with _$EditorStore;

abstract class _EditorStore with Store {
  final TasksRepository _tasksRepository;

  _EditorStore({
    required TasksRepository tasksRepository,
  }) : _tasksRepository = tasksRepository;

  @observable
  EditorState state = const EditorState.initial();

  @action
  Future<void> updateTask(final TaskModel task) async {
    try {
      state = const _Loading();
      await _tasksRepository.updateTask(task: task);
      state = _SuccessUpdate(task: task);
    } on Object catch (error) {
      state = _Failure(error: error);
      state = const _Initial();
      rethrow;
    }
  }

  @action
  Future<void> removeTask(final String taskId) async {
    try {
      state = const _Loading();
      await _tasksRepository.deleteTask(taskId: taskId);
      state = _SuccessRemove(taskId: taskId);
    } on Object catch (error) {
      state = _Failure(error: error);
      state = const _Initial();
      rethrow;
    }
  }
}
```

Рис. 3.13. Графічне представлення програмної реалізації екрану «Редагування задач» методу управління станом MobX

Дія «@action updateTask» оновлює задачу і стан у відповідності з результатами операції. Дія «@action removeTask» видаляє задачу та оновлює стан у відповідності з результатами операції. Логіка обробки помилок відповідна до попередній (див. рис. 3.13).

3.2.3. Програмна реалізація методу управління Redux

Програмна реалізація методу управління станом Redux для екрану «Список задач» представляє собою реалізацію класу «TasksReducer», що використовується для створення сховища.

Клас «TasksReducer» має конструктор, що приймає сховище задач і метод «createStore», що створює сховище з початковим станом «loading» і передачею сховища як оточення. Конструктор TasksReducer приймає екземпляр сховища задач «TasksRepository», а метод «createStore()» створює екземпляр сховища «Store<TasksState>». У «initialState» встановлюється початковий стан сховища як «TasksState.loading()». У «environment» передається екземпляр «_tasksRepository» як оточення для доступу до даних (див. рис. 3.14).

```
class TasksReducer {
  final TasksRepository _tasksRepository;

  TasksReducer({
    required TasksRepository tasksRepository,
  }) : _tasksRepository = tasksRepository;

  Store<TasksState> createStore() => Store<TasksState>(
        initialState: const TasksState.loading(),
        environment: _tasksRepository,
      );
}
```

Рис. 3.14. Графічне представлення програмної реалізації екрану «Список задач» методу управління станом Redux

Реалізація дій визначена в наступних класах:

– «FetchTasksAction»: виконує запит задач за допомогою сховища та оновлює стан у ньому;

– «RemoveTaskAction»: виконує видалення задач за його ідентифікатором і оновлює стан, видаляючи їх зі списку задач в успішному стані «_Success»;

– «AddTaskAction»: додає нову задачу до списку і оновлює стан; якщо стан успішний «_Success», то задача додається в поточний список і сортується за датою завершення.

– «UpdateTaskAction»: оновлює статус виконання задач й оновлює стан, зберігаючи зміни в списку задач.

– «UpdateReturnedTaskAction»: оновлює інформацію про задачу, повернуту з репозиторію, і оновлює відповідну в списку задач в успішному стані «_Success».

Кожна дія реалізує метод «reduce()», який виконує фактичну логіку оновлення стану. Обробка помилок також передбачена, і в разі помилки стан оновлюється на «_Failure».

Екран «Створення задач» супроводжується класом «CreatorReducer», який містить логіку створення сховища для управління станом створення задач. У даному випадку, він створює «Store<CreatorState>» з початковим станом «CreatorState.initial()» та використовує «TasksRepository» в якості оточення.

Дія «CreateTaskAction» розширюється «ReduxAction<CreatorState>» та представляє собою створення задачі та містить інформацію про неї. «TasksRepository get env => super.env as TasksRepository» перерозподіляє метод «env», який надає доступ до оточення дій (див. рис. 3.15).

Метод «Future<CreatorState> reduce() async { ... }» виконує логіку обробки дії. Всередині «try» виконується спроба створення нової задачі з використанням методу «createTask» з «TasksRepository». У випадку успішного створення, повертається новий стан «_Success» з інформацією про неї, у випадку помилки повертається стан «_Initial» (див. рис. 3.15).

```
class CreateTaskAction extends ReduxAction<CreatorState> {
  final TaskModel task;

  CreateTaskAction({required this.task});

  @override
  TasksRepository get env => super.env as TasksRepository;

  @override
  Future<CreatorState> reduce() async {
    try {
      // return const _Loading();
      final currentTask = await env.createTask(task: task);
      return _Success(task: currentTask);
    } on Object catch (error) {
      // return _Failure(error: error);
      return const _Initial();
    }
  }
}
```

Рис. 3.15. Графічне представлення програмної реалізації екрану «Створення задач» методу управління станом Redux класу «CreateTaskAction»

Для реалізації екрану «Редагування задач» було створено клас редуктора «EditorReducer», який містить екземпляр «TasksRepository». Було розроблено метод, який створює та повертає сховище для керування станом: «Store<EditorState> createStore() => Store<EditorState>(...)». В якості початкового стану встановлюється «EditorState.initial()», а оточення задається як екземпляр (див. рис. 3.16).

```
class EditorReducer {
  final TasksRepository _tasksRepository;

  EditorReducer({
    required TasksRepository tasksRepository,
  }) : _tasksRepository = tasksRepository;

  Store<EditorState> createStore() => Store<EditorState>(
    initialState: const EditorState.initial(),
    environment: _tasksRepository,
  );
}
```

Рис. 3.16. Графічне представлення програмної реалізації екрану «Редагування задач» методу управління станом Redux

У реалізації екрану редагування задач також присутні дії. «RemoveTaskAction» необхідний для видалення задачі, містить параметр «taskId», який надає ідентифікатор задачі. При виконанні дії викликається метод «env.deleteTask» для видалення задачі за ідентифікатором. При успішно виконаній дії повертається «_SuccessRemove», у зворотному випадку «_Initial».

Для редагування задач була реалізована дія «EditorTaskAction». Вона містить параметр «task», що представляє модель оновленого завдання. При виконанні цієї дії, викликається метод «env.updateTask». Якщо операція оновлення проходить успішно, повертається новий стан «_SuccessUpdate», що містить інформацію про відредаговану задачу, у зворотному випадку повертається стан «_Initial».

3.3. Програмна реалізація шару «Data Layer»

Шар даних у мобільному застосунку керує доступом до даних, таким як база даних, віддалені API або сховища. Для програмної реалізації даних було створено абстрактний клас «TasksRepository» (див. рис. 3.17), який слугує інтерфейсом для взаємодії з задачами у застосунку та реалізовується класом «TasksRepositoryImpl».

```
abstract interface class TasksRepository {
  Future<Iterable<TaskModel>> fetchTasks({
    required final String uid,
    final String? taskId,
  });

  Future<TaskModel> fetchTask({required final String taskId});

  Future<TaskModel> createTask({required final TaskModel task});

  Future<void> deleteTask({required final String taskId});

  Future<void> updateTask({required final TaskModel task});
}
```

Рис. 3.17. Графічне представлення програмної реалізації «Data Layer» класу-інтерфейсу «TasksRepository»

Клас «TasksRepositoryImpl» оголошений та відмічений як імутабельний, що вказує на те, що його екземпляри незмінні. Також у ньому визначений конструктор, який приймає обов'язковий параметр типу «TasksNetworkDataProviderImpl» та присвоює його приватному полю «_tasksNetworkDataProviderImpl». Реалізація методу інтерфейсу «TasksRepository» надає функціональність для роботи із задачами, такими як створення, видалення, отримання та оновлення (див. рис. 3.17).

«TasksNetworkDataProvider» (див. рис. 3.18) є також абстрактним класом, який необхідний для надання інформації про задачі за мережею, реалізовується класом «TasksNetworkDataProviderImpl».

```
abstract interface class TasksNetworkDataProvider {
  Future<Iterable<TaskModel>> fetchTasks({required String uid, String? taskId});

  Future<TaskModel> fetchTask({required String taskId});

  Future<String> createTask({required TaskModel task});

  Future<void> deleteTask({required String taskId});

  Future<void> updateTask({required TaskModel task});
}
```

Рис. 3.18. Графічне представлення програмної реалізації «Data Layer» класу-інтерфейсу «TasksNetworkDataProvider»

У класі «TasksNetworkDataProviderImpl» є конструктор, який приймає екземпляр «CloudFirestore» в якості параметру та ініціалізує приватну змінну «_cloudFirestore». У класі присутні методи видалення, витяг, створення та оновлення задач (див. рис. 3.18). Також був розроблений метод витягнення списку задач із сховища на основі ідентифікатора «uid». Для отримання поточного «uid» використовується екземпляр «FirebaseAut».

Клас «CloudFirestore» (див. рис. 3.19) забезпечує простий інтерфейс для виконання базових операцій взаємодії із сховищем Firestore сервісу Firebase.

```
class CloudFirestore {
  const CloudFirestore({
    required FirebaseFirestore firestore,
  }) : _firestore = firestore;

  final FirebaseFirestore _firestore;
```

Рис. 3.19. Графічне представлення програмної реалізації «Data Layer» класу «CloudFirestore»

У класі присутні наступні методи (див. рис. 3.20):

– «create»: додає новий документ у вказану колекцію із заданими даними;

– «read»: отримує документ із вказаної колекції за його ідентифікатором;

– «update»: оновлює дані існуючого документу в указаній колекції за його ідентифікатором;

– «delete»: видаляє документ за вказаною колекцією за його ідентифікатором.

```
Future<void> create({
  required final String collection,
  required final Map<String, dynamic> data,
}) async {
  try {
    await _firestore.collection(collection).add(data);
  } on Object catch (error, stackTrace) {
    Error.throwWithStackTrace(error, stackTrace);
  }
}

Future<DocumentSnapshot> read({
  required final String collection,
  required final String documentId,
}) async {
  try {
    return await _firestore.collection(collection).doc(documentId).get();
  } on Object catch (error, stackTrace) {
    Error.throwWithStackTrace(error, stackTrace);
  }
}
```

Рис. 3.20. Графічне представлення програмної реалізації «Data Layer» методів «create» та «read» класу «CloudFirestore»

Кожен метод використовує екземпляр «FirebaseFirestore» для виконання відповідної операції. У разі виникнення помилки, код перехоплює виняток і

використовує функцію «Error.throwWithStackTrace» для виводу інформації про помилку та стек викликів.

3.4. Результати програмної реалізації мобільного застосунку

Результатами програмної розробки є мобільний застосунок «TaskHub» із трьома екранами. На екрані «Список задач» користувач може переглянути наявні задачі (див. рис. 3.21), відмітити із виконання або не виконання (див. рис. А.10), перейти до створення або редагування задач.

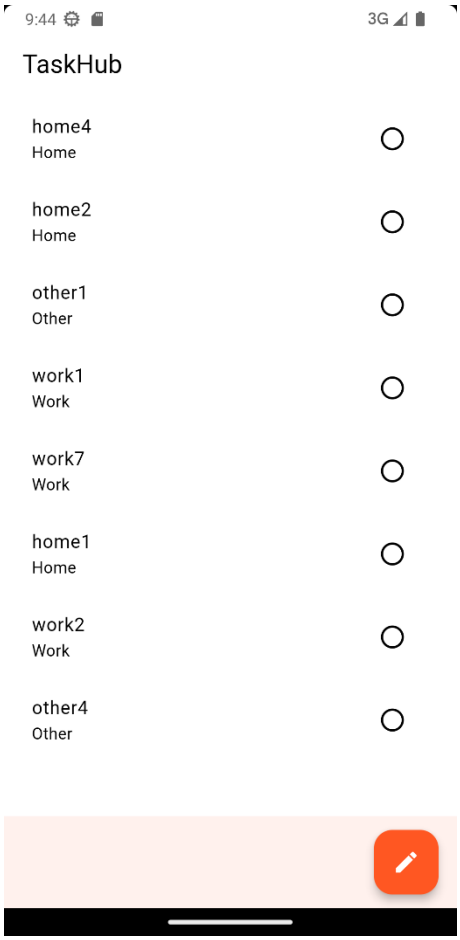

Рис. 3.21. Графічне представлення екрану «Список задач»

При натисненні на кнопку «Створити» відкриється екран «Створення задач» (див. рис. 3.22). При натисненні на будь-яке з полів з’явиться клавіатура для заповнення або обрання необхідних даних (див. рис. А.11).

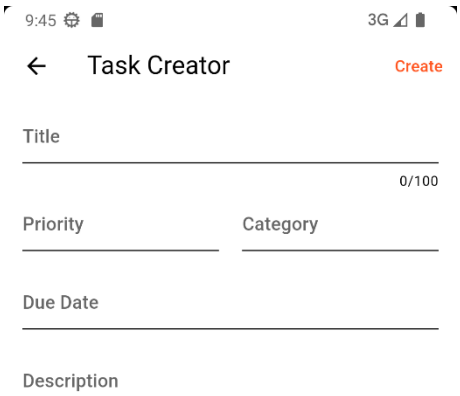

Рис. 3.22. Графічне представлення екрану «Створення задач»

Після заповнення та збереження задачі, її можна відкрити (див. рис. 3.23), видалити або відредагувати (див. рис. 3.24).

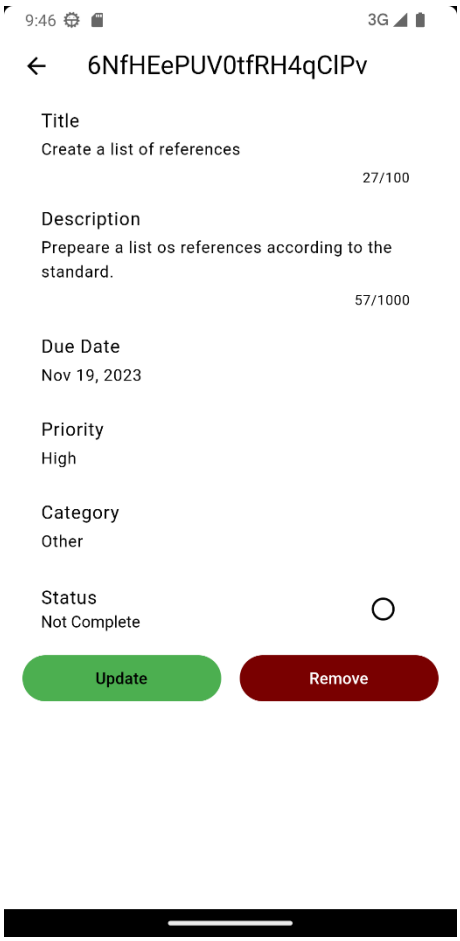

Рис. 3.23. Графічне представлення створеної задачі

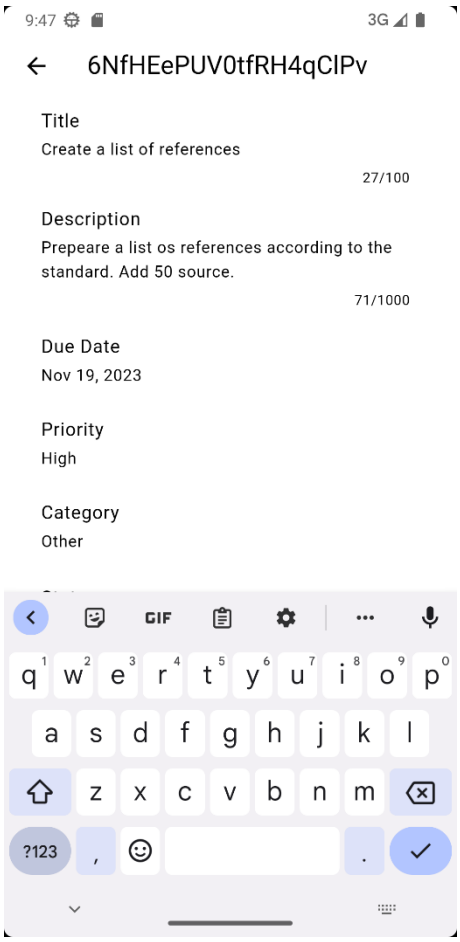

Рис. 3.24. Графічне представлення екрану «Редагування задач»

Також зі списку задач (див. рис. 3.21) можна видалити задачу свайпом ліворуч (див. рис. А.12).

3.5. Дослідження отриманих результатів за метриками

Для фіксації часу початку та завершення операції було використано функцію «DateTime.now()» (див. рис. 3.25).

```
dartCopy codefinal startTime = DateTime.now();
// Виконайте операцію тут
final endTime = DateTime.now();
final executionTime = endTime.difference(startTime);
print('Час виконання операції: $executionTime');
```

Рис. 3.25. Графічне представлення програмної реалізації виміру часу

Результати виміру за трьома методами управління станом наведені в таблиці 3.1.

Таблиця 3.1

Таблиця результатів виміру часу виконання операцій

Найменування операції	Результати виміру за методами управління станом, мс		
	BLoC	MobX	Redux
Оновлення	14,79	15,82	12,67
Видалення	18,59	15,88	11,98
Додавання	0,08	0,03	0,26
Завантаження	54,23	82,13	173,25

За таблицею 3.1 був побудований графік швидкодії даних методів управління станом (див. рис. 3.26).

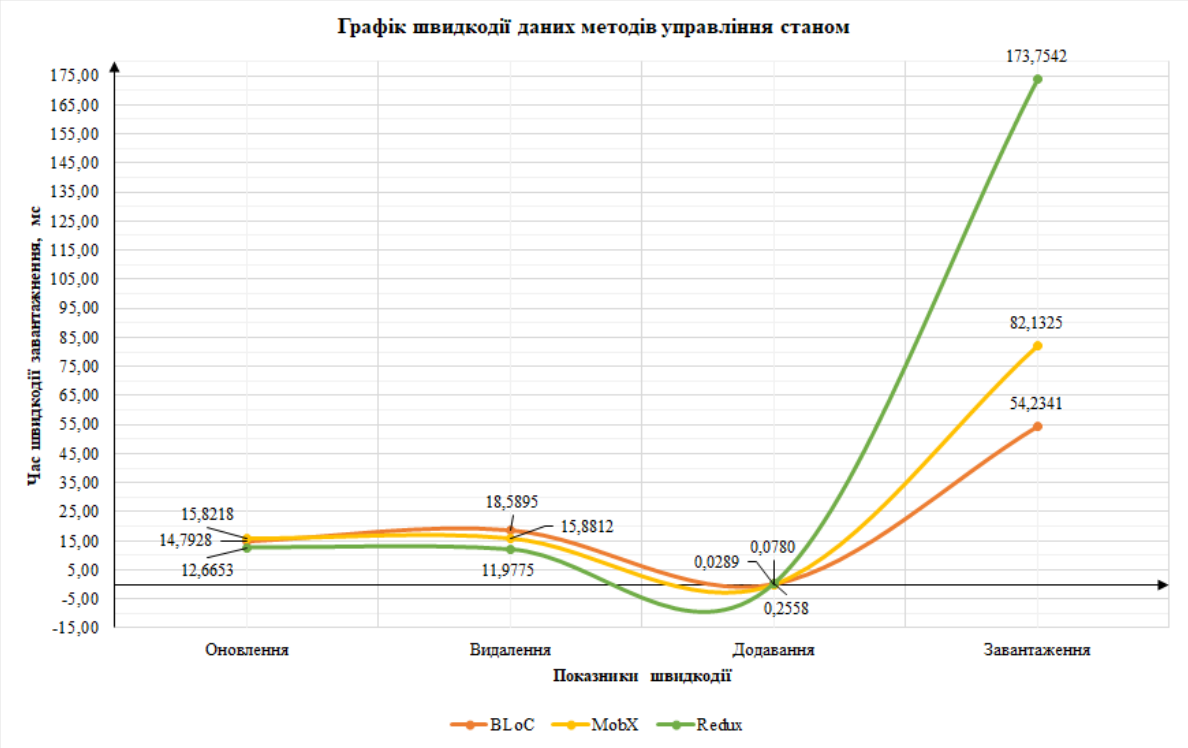

Рис. 3.26. Графік швидкодії операцій методів управління станом мобільного застосунку «TaskHub»

За часом оновленням, видаленням та додаванням даних можна стверджувати, що Redux справився краще свої опонентів, а на операції завантаженні швидше був BLoC.

Вимір розмірів коду репозиторію «feature» було виконано за допомогою Flutter DevTools [50]. За даними найважчим проєктом виявився з методом управління станом BLoC, а найлегшим Redux (див. рис. А.13-А.15).

За кожним екраном для кожного методу управління станом було розраховано мінімальний відсоток коду для роботи та спільний відсоток коду. Для розрахунку використовувалась наступна формула:

$$K_{\Delta} = \frac{K_1 - K_2}{K_1} * 100\%, \tag{2.1}$$

де

K_1 – загальний розмір коду методу управління станом із врахуванням спільних функцій;

K_2 – розмір чистого коду методу управління станом без спільних функцій.

Результати розрахунків за формулою (2.1) наведені в таблиці 3.2-3.3.

Таблиця 3.2

Таблиця результатів мінімального коду для роботи методів управління станом

Методи управління станом	Відсоток мінімального коду для роботи за екранами, %		
	Список задач	Створення задач	Редагування задач
BLoC	94,96%	98,34%	98,55%
MobX	98,35%	97,44%	98,11%
Redux	97,85%	97,83%	98,04%

Таблиця 3.3

Таблиця результатів спільного коду методів управління станом

Методи управління станом	Відсоток мінімального коду для роботи за екранами, %		
	Список задач	Створення задач	Редагування задач
BLoC	5,04%	1,66%	1,45%
MobX	1,65%	2,06%	1,89%
Redux	2,15%	2,17%	1,96%

Також було розраховано середнє значення мінімального коду для роботи та спільного коду і побудовано кругову діаграму (див. рис. 3.27).

Рис. 3.27. Графік мінімального коду для роботи та спільного коду методів управління станом

Додатково за допомогою Flutter DevTools було виміряно витрачену пам’ять в процесі роботи мобільного застосунку «TaskHub» під трьома методами управління станом. Були виміряні наступні показники:

– «Resident Set Size» («RSS»): об’єм пам’яті, який містить завантажені бібліотеки, пам’ять стеку та кучі;

– «Allocated»: поточний об’єм кучі;

– «Dart / Flutter»: об’єкти Dart та Flutter, що розташовані в кучі;

– «Dart / Flutter Native»: пам’ять, яка не розташована в кучі «Dart / Flutter», але є частиною загального об’єму пам’яті мобільного застосунку.

Результати виміру були занесені в таблицю 3.4.

Таблиця 3.4

Таблиця результатів виміру витраченої пам’яті за методами управління станом

Найменування показників	Результати виміру за методами управління станом, MB		
	BLoC	MobX	Redux
«RSS»	316,02	363,23	337,41
«Allocated»	76,75	79,63	84,74
«Dart / Flutter»	61,18	61,40	71,56
«Dart / Flutter Native»	0,0008	0,0008	0,0011

За результатами (див. табл. 3.4) можна стверджувати, що BLoC витрачає найменше мегабайт пам’яті, аніж MobX і Redux.

3.6. Висновки за розділом 3

За даним розділом було описано програмну реалізацію шару віджетів, програмну реалізацію бізнес-логіки під кожен метод управління станом, реалізацію шару даних, наведено результати розробленого мобільного застосунку та результати дослідження за метриками.

ВИСНОВКИ

Мобільні застосунку будуть супроводжувати нас ще довгий час. Тому так важливо зосередитися на дослідженні методів управління станом, тому що вони є важливим аспектом при розробці програмних продуктів на мобільні пристрої.

Результатами проведеного дослідження є розроблені та порівняні мобільні застосунки з використанням методів управління станом BLoC, MobX і Redux.

За час виконання кваліфікаційної випускної магістерської роботи були виконані наступні поставлені завдання:

– проаналізовано розробку кросплатформних застосунків з використанням фреймворків, визначені переваги та недоліки;

– проаналізовані методи управління станом на основі кросплатформних фреймворків Flutter, React Native та Xamarin;

– визначені переваги та недоліки методів управління станом на основі кросплатформних фреймворків і визначено, що найкращий із описаних є Flutter;

– проаналізована проблематика розробки кросплатформних застосунків на основі фреймворку Flutter;

– описані функціональні вимоги до мобільного застосунку «TaskHub»;

– спроєктована архітектура мобільного застосунку у вигляді діаграм варіантів використання, класів і компонентів;

– спроєктовані макети екранів мобільного застосунку;

– описані методи управління станом у Flutter, визначені їх переваги та недоліки;

– описані засоби розробки програмного забезпечення, встановлені їх переваги;

– визначені метрики порівняння методів управління станом;

– виконана програмна реалізація мобільних застосунків з використанням методів управління станом;

– описана програмна реалізація шару віджетів, бізнес-логіки та шару даних;

– приведені результати розроблених мобільних застосунків;

– проведене дослідження та порівняні результати метрик.

За результатами виміру часу виконання операцій (оновлення, видалення, додавання та завантаження) найкращі результати розділили між собою BLoC та Redux, а MobX надав середні результати.

За розрахунками мінімального коду для роботи та спільного коду результати у всіх методах управління станом є високими, тому що методи відрізняються один від одного та мали різні підходи до розробки поставленого завдання.

За результатами виміру витраченої пам’яті найкращі результати надав BLoC, середні показники у MobX, а найгірші у Redux.

За отриманими результатами можна стверджувати, що найзручнішим методом управління станом є BLoC. Він не надає на всіх метриках найкращих результатів, але має стабільнішу роботу та меншу кількість витраченої пам’яті. BLoC має більшу спільноту серед розробників, легший синтаксис та швидший процес розробки.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Мамон О. Кросплатформна розробка мобільних додатків. *Наукові праці викладачів, аспірантів, магістрантів і студентів фізико-математичного факультету*. 2023. № 41. С. 107-109.

2. Кучменко І. А. Розробка кросплатформного застосунку візуалізації тривимірних числових даних : магістерська робота. Запоріжжя, 2020. 42 с.

3. Xanthopoulos S., Xinogalos S. A comparative analysis of cross-platform development approaches for mobile applications. *Association for Computing Machinery* : 6th balkan conference in informatics, Thessaloniki, 19 September 2013. P. 213-220.

4. Лаврека Д. М. Розробка мобільного додатку для пошуку роботи : магістерська робота. Одеса, 2020. 60 с.

5. Порєв Г. В. Дослідження методів розробки кросплатформного програмного забезпечення. *Методи та системи оптико-електронної і цифрової обробки зображень та сигналів*. 2013. Т. 1, № 19. С. 64-68.

6. Taxonomy of cross-platform mobile applications development approaches / W. S. El-Kassas et al. *Ain shams engineering journal*. 2017. Vol. 8, no. 2. P. 163-190.

7. Ciman M., Gaggi O., Gonzo N. Cross-platform mobile development: a study on apps with animations. *Association for computing machinery* : SAC 2014: Symposium on Applied Computing, Gyeongju Republic, 24-28 March 2014.

8. Савельєв О. С., Розломій І. О. Сучасні фреймворки для розробки Android-додатків. *Інформаційна безпека та комп'ютерні технології* : V Міжнар. науково-практ. конф., м. Кропивницький, 19-20 трав. 2022 р. С. 40-41.

9. Філімончук Т. В., Чепелєв Є. О. Модель мобільного застосунку з використанням фреймворку Flutter. *Проблеми інформатизації* : IX Міжнар. науково-техн. конф., м. Харків. С. 99.

10. Коба Ю. Ю., Афанасьєва І. В., Онищенко К. Г. Використання технології Flutter для розробки кросплатформлених застосунків. *Current issues of*

science and integrated technologies : The 1th International scientific and practical conference, м. St. Louis, 29 жовт. – 1 листоп. 2023 р. С. 799.

11. Чепелєв Є. О. Модель мобільного застосунку з використанням фреймворку Flutter : магістерська робота. Харків, 2021. 52 с.

12. Kuitunen M. Cross-Platform mobile application development with React Native : extended abstract of Bachelor`s Thesis. 2019.

13. Wu W. React Native vs Flutter, cross-platform mobile application frameworks : extended abstract of Bachelor`s Thesis. 2018.

14. Сахарова А. Використання React та React Native у розробці користувальницького проекту для зручного користування зарядними станціями для електромобілів. *Technical sciences multidisciplinary academic research, innovation and results* : The XIII International Scientific And Practical Conference, м. Prague, 5-8 квіт. 2022 р. С. 771-772.

15. Chernysh A. Як ми інтегрували React Native у наявний Android застосунок. Розглядаємо реальний кейс. *18 серпня 2021*. URL: https://dou.ua/forums/topic/34405/.

16. Манойло О. О., Владімірова В. Б. Порівняння розробки кросплатформних мобільних додатків з використанням Xamarin та Ionic2. *Стан, досягнення і перспективи інформаційних систем і технологій* : Матеріали XVII Всеукр. науково-техн. конф. молодих вчен., аспірантів та студентів, м. Одеса, 19 квіт. 2017 р. С. 54-55.

17. Люлік А. Ю. Засоби розробки мобільних застосунків. *Кібербезпека в сучасному світі* : Матеріали III Всеукр. науково-практ. конф., м. Одеса, 19 листоп. 2021 р. / ред. О. В. Дикого ; уклад.: С. А. Горбаченко, Н. І. Логінова. С. 127-133.

18. Подопригора А. А., Міхайлуца О. М. Розробка крос-платформних застосунків з використанням технології Xamarin. *Eurasian Scientific Discussions* : Матеріали IV International Scientific and Practical Conference, м. Barcelona, 8-10 трав. 2022 р. С. 206-208.

19. Hatfield S. React Native vs. Flutter vs. Xamarin. URL: https://medium.com/@Toglefritz/flutter-vs-react-native-vs-xamarin-e43307963537.

20. Сілагін О. В., Цимбалістий В. В. Вибір інформаційної технології для розробки навчальних мобільних додатків. *Internet-Education-Science* : Матеріали XII International scientific-practical conference, м. Vinnytsia, 26-29 трав. 2020 р. С. 265-267.

21. Vailshery L. S. Cross-platform mobile frameworks used by software developers worldwide from 2019 to 2022, 2023. URL: https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/.

22. Синельніков О. Дослідження кросплатформенного фреймворку Flutter. чи означатиме розквіт цієї технології зникнення нативної розробки на Android та iOS?. *Комп'ютерні науки та кібербезпека*. 2019. Т. 2, № 2019. С. 19-25.

23. Пасічник В. В. Розробка мобільного додатку для тестування з використанням фреймворку Flutter : магістерська робота. Запоріжжя, 2020. 56 с.

24. Application development using Flutter / A. Tashildar et al. *International research journal of modernization in engineering technology and science*. 2020. Vol. 2, no. 8. P. 1262-1266.

25. Осташко Є. В. Дослідження методів створення сервісно-орієнтованих та крос-платформних додатків за допомогою Flutter з сервесною частиною на Firebase : магістерська робота. Харків, 2023. 63 с.

26. Що таке функціональні вимоги: приклади, визначення, повний посібник. URL: https://visuresolutions.com/uk/блог/функціональні-вимоги/.

27. A study of non-functional requirements in apps for mobile devices / L. Corbalán et al. *Conference on cloud computing and big data*. 2019. Vol. 1050. P. 125-136.

28. Rieger C., Majchrzak T. A. Towards the definitive evaluation framework for cross-platform app development approaches. *Journal of systems and software*. 2019. Vol. 153, no. 2019. P. 175-199.

29. Головін Е. В. Розробка мобільного застосунку для створення зображень за їхнім текстовим описом : бакалаврська робота. Харків, 2023. 78 с.

30. Кислий О. І. Порівняльний аналіз інструментальних засобів для розробки мобільних додатків. *Наукові записки молодих учених*. 2021. Т. 8, № 2021.

31. Ohst D., Welle M., Kelter U. Differences between versions of UML diagrams. *Association for computing machinery* : Матеріали 9th European Software Engineering Conference 2003, Helsinki, 1-5 September 2003. 2003. P. 227-236.

32. Maßen T. V., Lichter H. Modeling variability by UML use case diagrams. *International workshop on requirements engineering for product lines* : IEEE Joint International Requirements Engineering Conference, Essen, 9-13 September 2002. P. 19-25.

33. Herchi H., Abdessalem W. B. From user requirements to UML class diagram. 4 November 2012.

34. Supporting the composition of UML component diagrams / G. Ermel et al. *Association for computing machinery* : Матеріали XIV Brazilian Symposium on Information Systems, Caxias do Sul, 4-8 June 2018. 2018. P. 1-9.

35. Slepnev D. State management approaches in Flutter : extended abstract of Bachelor’s thesis. 2020. 98 p.

36. Nayak S. K. Flutter bloc state management: a comprehensive step-by-step guide. URL: https://shivamkumarnayak.medium.com/flutter-bloc-state-management-a-comprehensive-step-by-step-guide-213b38d7041b.

37. Designing a flutter application in a “clean” way using bloc. URL: https://habr.com/en/articles/733960/.

38. Matrenin A. Огляд архітектур управління станом на flutter. URL: https://dou.ua/lenta/articles/flutter-architecture/.

39. MobX: Flutter state management like a boss. URL: https://codereis.com/posts/mobx-state-management/.

40. Bui M. Managing UI state in Flutter with MobX and provider – Dissecting a Hacker News app. URL: https://dexterx.dev/managing-ui-state-in-flutter-with-mobx-and-provider/.

41. We are writing an application in Flutter in conjunction with Redux. URL: https://habr.com/en/articles/481624/.

42. Rawas P. Redux state management in Flutter. URL: https://medium.flutterdevs.com/redux-state-management-in-flutter-a64873dec1d0.

43. Wong J. Flutter Redux Thunk, an example finally. URL: https://medium.com/codechai/flutter-redux-thunk-27c2f2b80a3b.

44. Mahendra M., Anggorojati B. Evaluating the performance of Android based Cross-Platform App Development Frameworks. Tokyo, 27-29 November 2020. New York, 2021. P. 32-37.

45. Evolution of mobile software development from platform-specific to web-based multiplatform paradigm / L. Corral et al. Portland Oregon, 22-27 October 2011. New York.

46. Матюнін М. Anti-patterns of error handling in dart. URL: https://plugfox.dev/error-handling-and-anti-patterns/.

47. Fayzullaev J. Native-like cross-platform mobile development : multi-os engine & Kotlin Native vs Flutter : extended abstract of Bachelor's thesis. 2018. 62 p.

48. Tyagi P. Pragmatic flutter: building cross-platform mobile apps for android, ios, web & desktop. CRC Press, 2021. 337 p.

49. Performance and stability comparison of react and flutter: cross-platform application development / K. Kishore et al. Dubai, 6-7 October 2022. P. 1-4.

50. Матюнін М. Business logic component. URL: https://plugfox.dev/business-logic-component-2/.

ДОДАТОК А

КОД ПРОГРАМИ

main.dart //файл запуску та ініціалізації проєкту методу управління станом BLoC

```
import 'dart:async';
import 'package:bloc_concurrency/bloc_concurrency.dart';
import 'package:flutter/material.dart';
import 'package:firebase_core/firebase_core.dart';
import 'package:firebase_crashlytics/firebase_crashlytics.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
import 'package:taskhub/src/app.dart';
import 'package:taskhub/firebase_options.dart';
import 'package:taskhub/src/common/widget/navigation/navigation.dart';
import 'package:taskhub/src/common/bloc/bloc_observer.dart';
Future<void> main() async => runZonedGuarded(() async {
    final navigation = Navigation();
    Bloc.observer = BLoCObserver();
    Bloc.transformer = sequential();
    WidgetsFlutterBinding.ensureInitialized();
    await Firebase.initializeApp(
      options: DefaultFirebaseOptions.currentPlatform,
    );
    runApp(App(
      navigation: navigation,
    ));
  },
      (error, stack) =>
          FirebaseCrashlytics.instance.recordError(error, stack));
```

app.dart //файл налаштування та поведінку мобільного застосунку методу управління станом BLoC

```
import 'package:flutter/material.dart';
import 'package:taskhub/src/common/widget/navigation/navigation.dart';
import 'package:taskhub/src/common/widget/theme/dark_theme.dart';
import 'package:taskhub/src/common/widget/theme/light_theme.dart';
class App extends StatelessWidget {
  final Navigation navigation;
  const App({
    Key? key,
    required this.navigation,
  }) : super(key: key);
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      themeMode: ThemeMode.system,
      theme: LightTheme.themeData,
      darkTheme: DarkTheme.themeData,
      routes: navigation.routes,
      onGenerateRoute: navigation.onGenerateRoute,
      initialRoute: RouteNames.tasks,
    );
  }
}
```

creator_bloc.dart //файл логіки екрану створення задач методу управління станом BLoC

```
import 'package:flutter_bloc/flutter_bloc.dart';
```

```
import 'package:freezed_annotation/freezed_annotation.dart';
import 'package:taskhub/src/common/model/task/task_model.dart';
import 'package:taskhub/src/common/data/repository/tasks/tasks_repository.dart';
part 'creator_event.dart';
part 'creator_state.dart';
part 'creator_bloc.freezed.dart';
class CreatorBloc extends Bloc<CreatorEvent, CreatorState> {
  final TasksRepository _tasksRepository;
  CreatorBloc({
    required TasksRepository tasksRepository,
  })  : _tasksRepository = tasksRepository,
        super(const CreatorState.initial()) {
    on<CreatorEvent>(
      (event, emit) => event.map<Future<void>>(
        createTask: (event) => _onCreateTask(event, emit),
      ),
    );
  }
  Future<void> _onCreateTask(
    _CreateTask event,
    Emitter<CreatorState> emit,
  ) async {
    try {
      emit(const _Loading());
      final task = await _tasksRepository.createTask(task: event.task);
      emit(_Success(task: task));
    } on Object catch (error) {
      emit(_Failure(error: error));
      emit(const _Initial());
      rethrow;
    }
  }
}
```

creator_event.dart //файл опису класів дій методу управління станом BLoC

```
part of 'creator_bloc.dart';
@freezed
class CreatorEvent with _$CreatorEvent {
  const CreatorEvent._();
  const factory CreatorEvent.createTask({
    required final TaskModel task,
  }) = _CreateTask;
}
```

creator_scope.dart //файл взаємодії з діями методу управління станом BLoC

```
import 'package:flutter/material.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
import 'package:taskhub/src/common/model/task/task_model.dart';
import 'package:taskhub/src/feature/creator/bloc/creator_bloc.dart';
abstract class CreatorScope {
  static CreatorBloc of(BuildContext context) {
    return context.read<CreatorBloc>();
  }
  static void createTask(BuildContext context, TaskModel task) {
    of(context).add(CreatorEvent.createTask(task: task));
  }
}
```

creator_view.dart //файл опису UI та логікою методу управління станом BLoC

```
 import 'dart:async';
```

```
import 'package:flutter/services.dart';
import 'package:intl/intl.dart';
import 'package:flutter/material.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
import 'package:taskhub/src/common/model/task/task_model.dart';
import 'package:taskhub/src/feature/creator/bloc/creator_bloc.dart';
import 'package:taskhub/src/feature/creator/scope/creator_scope.dart';
class CreatorView extends StatefulWidget {
  const CreatorView({super.key});
  @override
  State<CreatorView> createState() => _CreatorViewState();
}
class _CreatorViewState extends State<CreatorView> {
  @override
  Widget build(BuildContext context) {
    final nameController = TextEditingController();
    final descriptionController = TextEditingController();
    final dueDateController = TextEditingController();
    final categoryController = TextEditingController();
    final priorityController = TextEditingController();
    final state = context.watch<CreatorBloc>().state;
    final formKey = GlobalKey<FormState>();
    return BlocListener<CreatorBloc, CreatorState>(
      listener: (context, state) {
        state.mapOrNull(
          success: (state) => Navigator.pop(context, state.task),
          failure: (state) => ScaffoldMessenger.of(context).showSnackBar(
            SnackBar(content: Text(state.error.toString())),
          ),
        );
      },
      child: Scaffold(
        appBar: AppBar(
          title: const Text('Task Creator'),
          actions: [
            TextButton(
              onPressed: state.mapOrNull(
                initial: (state) => () {
                  if (formKey.currentState!.validate()) {
                    HapticFeedback.vibrate();
                    final newTask = TaskModel(
                      category: categoryController.text,
                      completed: false,
                      description: descriptionController.text,
                      dueDate: dueDateController.text != ''
                          ? DateFormat.yMMMd().parse(dueDateController.text)
                          : null,
                      priority: priorityController.text,
                      taskId: '',
                      title: nameController.text,
                      uid: '',
                    );
                    CreatorScope.createTask(context, newTask);
                  }
                },
              ),
              child: const Text('Create'),
            ),
          ],
        ),
        body: Stack(
          alignment: Alignment.bottomCenter,
          children: [
            Padding(
              padding: const EdgeInsets.only(
                  left: 16.0, right: 16.0, top: 8.0, bottom: 32.0),
              child: Form(
                key: formKey,
                child: Column(
                  children: [
```

```
                TitleTextFormField(nameController: nameController),
                const SizedBox(height: 4.0),
                Row(
                  children: [
                    Expanded(
                      child: PriorityTextFormField(
                          priorityController: priorityController),
                    ),
                    const SizedBox(width: 20.0),
                    Expanded(
                      child: CategoryTextFormField(
                          categoryController: categoryController),
                    ),
                  ],
                ),
                const SizedBox(height: 20.0),
                DueDateTextFormField(dueDateController: dueDateController),
                const SizedBox(height: 20.0),
                Expanded(
                  child: DescriptionTextFormField(
                      descriptionController: descriptionController),
                ),
              ],
            ),
          ),
        ),
      ],
    ),
  ),
 );
 }
}
class DescriptionTextFormField extends StatelessWidget {
 const DescriptionTextFormField({
  super.key,
  required this.descriptionController,
 });
 final TextEditingController descriptionController;
 @override
 Widget build(BuildContext context) {
  return TextFormField(
   controller: descriptionController,
   keyboardType: TextInputType.text,
   decoration: const InputDecoration(
    hintText: 'Description',
   ),
   expands: true,
   maxLines: null,
   textAlignVertical: TextAlignVertical.top,
   maxLength: 1000,
   validator: (String? value) {
    if (value!.isNotEmpty && value.length > 1000) {
     return 'Description should not exceed 1000 characters.';
    }
    return null;
   },
  );
 }
}
class PriorityTextFormField extends StatelessWidget {
 const PriorityTextFormField({
  super.key,
  required this.priorityController,
 });
 final TextEditingController priorityController;
 @override
 Widget build(BuildContext context) {
  final priority = <String>['High', 'Medium', 'Low'];
  Future<void> openPriorityDialog() async {
   await showDialog<String?>(
```

```
        context: context,
        builder: (BuildContext context) {
          return AlertDialog(
            title: const Text('Select a Priority'),
            content: Column(
              mainAxisSize: MainAxisSize.min,
              children: priority
                  .map(
                    (e) => ListTile(
                      title: Text(e),
                      onTap: () {
                        Navigator.of(context).pop(e);
                      },
                    ),
                  )
                  .toList(),
            ),
          );
        },
      ).then((value) {
        if (value != null) {
          priorityController.text = value;
        }
      });
    }
    return TextFormField(
      controller: priorityController,
      showCursor: false,
      readOnly: true,
      onTap: () async => await openPriorityDialog(),
      decoration: const InputDecoration(
        hintText: 'Priority',
      ),
      validator: (String? value) {
        if (value == null || value == '') {
          return 'Priority is required.';
        }
        return null;
      },
    );
  }
}
class TitleTextFormField extends StatelessWidget {
  const TitleTextFormField({
    super.key,
    required this.nameController,
  });
  final TextEditingController nameController;
  @override
  Widget build(BuildContext context) {
    return TextFormField(
      controller: nameController,
      decoration: const InputDecoration(
        hintText: 'Title',
      ),
      keyboardType: TextInputType.text,
      maxLength: 100,
      validator: (String? value) {
        if (value == null || value == '') {
          return 'Title is required.';
        } else if (value.length > 100) {
          return 'Title should not exceed 100 characters.';
        }
        return null;
      },
    );
  }
}
class CategoryTextFormField extends StatelessWidget {
  const CategoryTextFormField({
```

```
    super.key,
    required this.categoryController,
  });
  final TextEditingController categoryController;
  @override
  Widget build(BuildContext context) {
    final categories = <String>['Work', 'Home', 'Other'];
    Future<void> openCategoryDialog() async {
      await showDialog<String?>(
        context: context,
        builder: (BuildContext context) {
          return AlertDialog(
            title: const Text('Select a Category'),
            content: Column(
              mainAxisSize: MainAxisSize.min,
              children: categories
                  .map((e) => ListTile(
                      title: Text(e),
                      onTap: () => Navigator.of(context).pop(e)))
                  .toList(),
            ),
          );
        },
      ).then((value) {
        if (value != null) {
          categoryController.text = value;
        }
      });
    }
    return TextFormField(
      controller: categoryController,
      showCursor: false,
      readOnly: true,
      onTap: () async => await openCategoryDialog(),
      decoration: const InputDecoration(
        hintText: 'Category',
      ),
      validator: (String? value) {
        if (value == null || value == '') {
          return 'Category is required.';
        }
        return null;
      },
    );
  }
}
class DueDateTextFormField extends StatelessWidget {
  const DueDateTextFormField({
    super.key,
    required this.dueDateController,
  });
  final TextEditingController dueDateController;
  @override
  Widget build(BuildContext context) {
    return TextFormField(
      controller: dueDateController,
      decoration: const InputDecoration(
        hintText: 'Due Date',
      ),
      showCursor: false,
      readOnly: true,
      validator: (String? value) {
        if (value != null && value != '') {
          final now = DateTime.now();
          final selectedDate = DateFormat.yMMMd().parse(value);
          if (selectedDate.isBefore(now)) {
            return 'Due date must be in the future.';
          }
        }
        return null;
```

```
      },
      onTap: () {
        final date = DateTime.now();
        showDatePicker(
          context: context,
          initialDate: DateTime.now(),
          firstDate: DateTime.now(),
          lastDate: DateTime(date.year + 10, date.month, date.day),
        ).then(
          (date) {
            if (date != null) {
              dueDateController.text = DateFormat.yMMMd().format(date);
            }
          },
        );
      },
    );
  }
}
```

editor_bloc.dart //файл логіки екрану редагування задач методу управління станом BLoC

```
import 'package:flutter_bloc/flutter_bloc.dart';
import 'package:freezed_annotation/freezed_annotation.dart';
import 'package:taskhub/src/common/data/repository/tasks/tasks_repository.dart';
import 'package:taskhub/src/common/model/task/task_model.dart';
part 'editor_event.dart';
part 'editor_state.dart';
part 'editor_bloc.freezed.dart';
class EditorBloc extends Bloc<EditorEvent, EditorState> {
  final TasksRepository _tasksRepository;
  EditorBloc({
    required TasksRepository tasksRepository,
  })  : _tasksRepository = tasksRepository,
        super(const EditorState.initial()) {
    on<EditorEvent>(
      (event, emit) => event.map<Future<void>>(
        removeTask: (event) => _onRemoveTask(event, emit),
        updateTask: (event) => _onUpdateTask(event, emit),
      ),
    );
  }
  Future<void> _onUpdateTask(
    _UpdateTask event,
    Emitter<EditorState> emit,
  ) async {
    try {
      emit(const _Loading());
      await _tasksRepository.updateTask(task: event.task);
      emit(_SuccessUpdate(task: event.task));
    } on Object catch (error) {
      emit(_Failure(error: error));
      emit(const _Initial());
      rethrow;
    }
  }
  Future<void> _onRemoveTask(
    _RemoveTask event,
    Emitter<EditorState> emit,
  ) async {
    try {
      emit(const _Loading());
      await _tasksRepository.deleteTask(taskId: event.taskId);
      emit(_SuccessRemove(taskId: event.taskId));
    } on Object catch (error) {
      emit(_Failure(error: error));
```

```
      emit(const _Initial());
      rethrow;
    }
  }
}
```

editor_event.dart //файл опису класів дій методу управління станом BLoC

```
part of 'editor_bloc.dart';
@freezed
class EditorEvent with _$EditorEvent {
  const EditorEvent._();
  const factory EditorEvent.removeTask({
    required final String taskId,
  }) = _RemoveTask;
  const factory EditorEvent.updateTask({
    required final TaskModel task,
  }) = _UpdateTask;
}
```

editor_scope.dart //файл взаємодії з діями методу управління станом BLoC

```
import 'package:flutter/material.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
import 'package:taskhub/src/feature/editor/bloc/editor_bloc.dart';
import 'package:taskhub/src/common/model/task/task_model.dart';
abstract class EditorScope {
  static EditorBloc of(BuildContext context) {
    return context.read<EditorBloc>();
  }
  static void removeTask(BuildContext context, String taskId) {
    of(context).add(EditorEvent.removeTask(taskId: taskId));
  }
  static void updateTask(BuildContext context, TaskModel task) {
    of(context).add(EditorEvent.updateTask(task: task));
  }
}
```

editor_view.dart //файл опису UI та логікою методу управління станом BLoC

```
import 'package:intl/intl.dart';
import 'package:flutter/material.dart';
import 'package:flutter/services.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
import 'package:taskhub/src/common/model/task/task_model.dart';
import 'package:taskhub/src/feature/editor/bloc/editor_bloc.dart';
import 'package:taskhub/src/feature/editor/scope/editor_scope.dart';
class EditorView extends StatefulWidget {
  final TaskModel task;
  const EditorView({
    Key? key,
    required this.task,
  }) : super(key: key);
  @override
  State<EditorView> createState() => _EditorViewState();
}
class _EditorViewState extends State<EditorView> {
  late TextEditingController nameController;
  late TextEditingController descriptionController;
  late TextEditingController dueDateController;
  late TextEditingController categoryController;
  late TextEditingController priorityController;
  late bool status;
```

```
@override
void initState() {
  super.initState();
  nameController = TextEditingController(text: widget.task.title);
  descriptionController =
      TextEditingController(text: widget.task.description);
  dueDateController = TextEditingController(
      text: widget.task.dueDate != null
          ? DateFormat.yMMMd().format(widget.task.dueDate!)
          : '');
  categoryController = TextEditingController(text: widget.task.category);
  priorityController = TextEditingController(text: widget.task.priority);
  status = widget.task.completed;
}
@override
void dispose() {
  nameController.dispose();
  descriptionController.dispose();
  dueDateController.dispose();
  categoryController.dispose();
  priorityController.dispose();
  super.dispose();
}
bool hasTaskChanged() {
  return nameController.text != widget.task.title ||
      descriptionController.text != widget.task.description ||
      dueDateController.text !=
          (widget.task.dueDate != null
              ? DateFormat.yMMMd().format(widget.task.dueDate!)
              : '') ||
      categoryController.text != widget.task.category ||
      priorityController.text != widget.task.priority ||
      status != widget.task.completed;
}
@override
Widget build(BuildContext context) {
  final formKey = GlobalKey<FormState>();
  return BlocListener<EditorBloc, EditorState>(
    listener: (context, state) {
      state.mapOrNull(
        successRemove: (state) {
          Navigator.pop(context, state.taskId);
        },
        successUpdate: (state) {
          Navigator.pop(context, state.task);
        },
        failure: (state) {
          ScaffoldMessenger.of(context).showSnackBar(
            SnackBar(content: Text(state.error.toString())),
          );
        },
      );
    },
    child: Scaffold(
      appBar: AppBar(
        title: Text(widget.task.taskId),
      ),
      body: Padding(
        padding: const EdgeInsets.symmetric(horizontal: 16.0),
        child: Form(
          key: formKey,
          child: CustomScrollView(
            slivers: [
              SliverList(
                delegate: SliverChildListDelegate([
                  TitleTextFormField(nameController: nameController),
                  DescriptionTextFormField(
                      descriptionController: descriptionController),
                  DueDateTextFormField(dueDateController: dueDateController),
                  PriorityTextFormField(
```

```
    priorityController: priorityController),
CategoryTextFormField(
    categoryController: categoryController),
StatusCard(
  status: status,
  onStatusChanged: (newStatus) {
    setState(() {
      status = newStatus;
    });
  },
),
BlocBuilder<EditorBloc, EditorState>(
  builder: (context, state) {
    return Row(
      children: [
        Expanded(
          child: FilledButton.tonal(
            onPressed: state.mapOrNull(
              initial: (state) => () {
                HapticFeedback.vibrate();
                if (formKey.currentState!.validate()) {
                  if (hasTaskChanged()) {
                    EditorScope.updateTask(
                      context,
                      widget.task.copyWith(
                        category: categoryController.text,
                        completed: status,
                        description:
                            descriptionController.text,
                        dueDate:
                            dueDateController.text != ''
                                ? DateFormat.yMMMd().parse(
                                    dueDateController.text)
                                : null,
                        priority: priorityController.text,
                        title: nameController.text,
                      ),
                    );
                  } else {
                    Navigator.pop(context);
                  }
                }
              },
            ),
            child: const Text('Update'),
          ),
        ),
        const SizedBox(width: 16.0),
        Expanded(
          child: FilledButton.tonal(
            style: FilledButton.styleFrom(
              backgroundColor:
                  Theme.of(context).colorScheme.error,
            ),
            onPressed: state.mapOrNull(
              initial: (state) => () {
                HapticFeedback.vibrate();
                EditorScope.removeTask(
                    context, widget.task.taskId);
              },
            ),
            child: Text(
              'Remove',
              style: TextStyle(
                color:
                    Theme.of(context).colorScheme.onError,
              ),
            ),
          ),
        ),
```

```
                    ],
                  );
                },
              ),
            ]),
          ),
        ],
      ),
    ),
  ),
 );
 }
}

class StatusCard extends StatelessWidget {
 final bool status;
 final ValueChanged<bool> onStatusChanged;
 const StatusCard({
  Key? key,
  required this.status,
  required this.onStatusChanged,
 }) : super(key: key);
 @override
 Widget build(BuildContext context) {
  return ListTile(
   title: const Text('Status'),
   subtitle: Text(status ? 'Complete' : 'Not Complete'),
   trailing: IconButton(
    icon: Icon(status ? Icons.check_circle_outline : Icons.circle_outlined),
    onPressed: () {
     HapticFeedback.vibrate();
     onStatusChanged(!status);
    },
   ),
   subtitleTextStyle: TextStyle(
    fontSize: 14.0,
    color: Theme.of(context).colorScheme.onBackground,
   ),
  );
 }
}
class DescriptionTextFormField extends StatelessWidget {
 const DescriptionTextFormField({
  Key? key,
  required this.descriptionController,
 }) : super(key: key);
 final TextEditingController descriptionController;
 @override
 Widget build(BuildContext context) {
  return ListTile(
   title: const Text('Description'),
   subtitle: TextFormField(
    controller: descriptionController,
    keyboardType: TextInputType.text,
    style: TextStyle(
     fontSize: 14.0,
     color: Theme.of(context).colorScheme.onBackground,
    ),
    decoration: const InputDecoration(
     hintText: 'Description',
     isDense: true,
     filled: false,
     contentPadding: EdgeInsets.zero,
     errorBorder: InputBorder.none,
     focusedBorder: InputBorder.none,
     focusedErrorBorder: InputBorder.none,
     disabledBorder: InputBorder.none,
     enabledBorder: InputBorder.none,
     border: InputBorder.none,
```

```
        ),
        maxLines: null,
        textAlignVertical: TextAlignVertical.top,
        maxLength: 1000,
        validator: (String? value) {
          if (value != null && value.isNotEmpty && value.length > 1000) {
            return 'Description should not exceed 1000 characters.';
          }
          return null;
        },
      ),
    );
  }
}
class PriorityTextFormField extends StatelessWidget {
  const PriorityTextFormField({
    Key? key,
    required this.priorityController,
  }) : super(key: key);
  final TextEditingController priorityController;
  @override
  Widget build(BuildContext context) {
    final priority = <String>['High', 'Medium', 'Low'];
    Future<void> openPriorityDialog() async {
      final value = await showDialog<String?>(
        context: context,
        builder: (BuildContext context) {
          return AlertDialog(
            title: const Text('Select a Priority'),
            content: Column(
              mainAxisSize: MainAxisSize.min,
              children: priority
                  .map(
                    (e) => ListTile(
                      title: Text(e),
                      onTap: () {
                        Navigator.of(context).pop(e);
                      },
                    ),
                  )
                  .toList(),
            ),
          );
        },
      );
      if (value != null) {
        priorityController.text = value;
      }
    }
    return ListTile(
      title: const Text('Priority'),
      subtitle: TextFormField(
        controller: priorityController,
        showCursor: false,
        readOnly: true,
        onTap: () => openPriorityDialog(),
        style: TextStyle(
          fontSize: 14.0,
          color: Theme.of(context).colorScheme.onBackground,
        ),
        decoration: const InputDecoration(
          hintText: 'Priority',
          isDense: true,
          filled: false,
          contentPadding: EdgeInsets.zero,
          errorBorder: InputBorder.none,
          focusedBorder: InputBorder.none,
          focusedErrorBorder: InputBorder.none,
          disabledBorder: InputBorder.none,
          enabledBorder: InputBorder.none,
```

```
          border: InputBorder.none,
        ),
        validator: (String? value) {
          if (value == null || value.isEmpty) {
            return 'Priority is required.';
          }
          return null;
        },
      ),
    );
  }
}
class TitleTextFormField extends StatelessWidget {
  const TitleTextFormField({
    Key? key,
    required this.nameController,
  }) : super(key: key);
  final TextEditingController nameController;
  @override
  Widget build(BuildContext context) {
    return ListTile(
      title: const Text('Title'),
      subtitle: TextFormField(
        controller: nameController,
        style: TextStyle(
          fontSize: 14.0,
          color: Theme.of(context).colorScheme.onBackground,
        ),
        decoration: const InputDecoration(
          hintText: 'Title',
          isDense: true,
          filled: false,
          contentPadding: EdgeInsets.zero,
          errorBorder: InputBorder.none,
          focusedBorder: InputBorder.none,
          focusedErrorBorder: InputBorder.none,
          disabledBorder: InputBorder.none,
          enabledBorder: InputBorder.none,
          border: InputBorder.none,
        ),
        keyboardType: TextInputType.text,
        maxLength: 100,
        validator: (String? value) {
          if (value == null || value.isEmpty) {
            return 'Title is required.';
          } else if (value.length > 100) {
            return 'Title should not exceed 100 characters.';
          }
          return null;
        },
      ),
    );
  }
}
class CategoryTextFormField extends StatelessWidget {
  const CategoryTextFormField({
    Key? key,
    required this.categoryController,
  }) : super(key: key);
  final TextEditingController categoryController;
  @override
  Widget build(BuildContext context) {
    final categories = <String>['Work', 'Home', 'Other'];
    Future<void> openCategoryDialog() async {
      final value = await showDialog<String?>(
        context: context,
        builder: (BuildContext context) {
          return AlertDialog(
            title: const Text('Select a Category'),
            content: Column(
```

```
                mainAxisSize: MainAxisSize.min,
                children: categories
                    .map((e) => ListTile(
                          title: Text(e),
                          onTap: () {
                            Navigator.of(context).pop(e);
                          },
                        ))
                    .toList(),
              ),
            );
          },
        );
        if (value != null) {
          categoryController.text = value;
        }
      }
      return ListTile(
        title: const Text('Category'),
        subtitle: TextFormField(
          controller: categoryController,
          showCursor: false,
          readOnly: true,
          onTap: () => openCategoryDialog(),
          style: TextStyle(
            fontSize: 14.0,
            color: Theme.of(context).colorScheme.onBackground,
          ),
          decoration: const InputDecoration(
            hintText: 'Category',
            isDense: true,
            filled: false,
            contentPadding: EdgeInsets.zero,
            errorBorder: InputBorder.none,
            focusedBorder: InputBorder.none,
            focusedErrorBorder: InputBorder.none,
            disabledBorder: InputBorder.none,
            enabledBorder: InputBorder.none,
            border: InputBorder.none,
          ),
          validator: (String? value) {
            if (value == null || value.isEmpty) {
              return 'Category is required.';
            }
            return null;
          },
        ),
      );
    }
}
class DueDateTextFormField extends StatelessWidget {
  const DueDateTextFormField({
    Key? key,
    required this.dueDateController,
  }) : super(key: key);
  final TextEditingController dueDateController;
  @override
  Widget build(BuildContext context) {
    return ListTile(
      title: const Text('Due Date'),
      subtitle: TextFormField(
        style: TextStyle(
          fontSize: 14.0,
          color: Theme.of(context).colorScheme.onBackground,
        ),
        controller: dueDateController,
        decoration: const InputDecoration(
          hintText: 'Due Date',
          isDense: true,
          filled: false,
```

```
          contentPadding: EdgeInsets.zero,
          errorBorder: InputBorder.none,
          focusedBorder: InputBorder.none,
          focusedErrorBorder: InputBorder.none,
          disabledBorder: InputBorder.none,
          enabledBorder: InputBorder.none,
          border: InputBorder.none,
        ),
        showCursor: false,
        readOnly: true,
        validator: (String? value) {
          if (value != null && value != '') {
            final now = DateTime.now();
            final selectedDate = DateFormat.yMMMd().parse(value);
            if (selectedDate.isBefore(now)) {
              return 'Due date must be in the future.';
            }
          }
          return null;
        },
        onTap: () {
          final date = DateTime.now();
          showDatePicker(
            context: context,
            initialDate: DateTime.now(),
            firstDate: DateTime.now(),
            lastDate: DateTime(date.year + 10, date.month, date.day),
          ).then((date) {
            if (date != null) {
              dueDateController.text = DateFormat.yMMMd().format(date);
            }
          });
        },
      ),
    );
  }
}
```

tasks_bloc.dart //файл логіки екрану списку задач методу управління станом BLoC

```
import 'package:firebase_auth/firebase_auth.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
import 'package:bloc_concurrency/bloc_concurrency.dart';
import 'package:freezed_annotation/freezed_annotation.dart';
import 'package:taskhub/src/common/model/task/task_model.dart';
import 'package:taskhub/src/common/data/repository/tasks/tasks_repository.dart';
part 'tasks_event.dart';
part 'tasks_state.dart';
part 'tasks_bloc.freezed.dart';
class TasksBloc extends Bloc<TasksEvent, TasksState> {
  final TasksRepository _tasksRepository;
  TasksBloc({
    required TasksRepository tasksRepository,
  })  : _tasksRepository = tasksRepository,
        super(const TasksState.loading()) {
    on<TasksEvent>(
      (event, emit) async => event.map<Future<void>>(
        fetchTasks: (event) => _onFetchTasks(event, emit),
        removeTask: (event) => _onRemoveTask(event, emit),
        updateTask: (event) => _onUpdateTask(event, emit),
        addTask: (event) => _onAddTask(event, emit),
        updateReturnTask: (event) => _onUpdateReturnTask(event, emit),
      ),
      transformer: droppable(),
    );
  }
```

```
Future<void> _onFetchTasks(
  _FetchTasks event,
  Emitter<TasksState> emit,
) async {
  try {
    final uid = FirebaseAuth.instance.currentUser!.uid;
    final tasks = await _tasksRepository.fetchTasks(
      uid: uid,
      taskId: event.taskId,
    );
    if (state is _Success) {
      if ((state as _Success).hasReachedMax) return;
      emit(
        tasks.isEmpty
            ? (state as _Success).copyWith(hasReachedMax: true)
            : (state as _Success).copyWith(
                tasks: (state as _Success).tasks + tasks.toList(),
                hasReachedMax: false,
              ),
      );
    } else {
      final newState = tasks.isEmpty
          ? const _Empty()
          : _Success(tasks: tasks.toList(), hasReachedMax: false);
      emit(newState);
    }
  } on Object catch (error) {
    emit(_Failure(error: error));
    rethrow;
  }
}
Future<void> _onRemoveTask(
  _RemoveTask event,
  Emitter<TasksState> emit,
) async {
  try {
    final taskId = event.taskId;
    await _tasksRepository.deleteTask(taskId: taskId);
    final updatedTasks = (state as _Success)
        .tasks
        .where((task) => task.taskId != taskId)
        .toList();
    if (updatedTasks.isEmpty) {
      emit(const _Empty());
    } else {
      emit((state as _Success).copyWith(tasks: updatedTasks));
    }
  } on Object catch (error) {
    emit(_Failure(error: error));
    rethrow;
  }
}
Future<void> _onUpdateTask(
  _UpdateTask event,
  Emitter<TasksState> emit,
) async {
  try {
    final updatedTasks = (state as _Success).tasks.map((task) {
      if (task.taskId == event.taskId) {
        return task.copyWith(completed: event.value);
      }
      return task;
    }).toList();
    await _tasksRepository.updateTask(
        task:
            updatedTasks.singleWhere((task) => task.taskId == event.taskId));
    emit((state as _Success).copyWith(tasks: updatedTasks));
  } on Object catch (error) {
    emit(_Failure(error: error));
    rethrow;
```

```
    }
  }
  Future<void> _onAddTask(
    _AddTask event,
    Emitter<TasksState> emit,
  ) async {
    final start = DateTime.now();
    try {
      if (state is _Success) {
        List<TaskModel> list = List.from((state as _Success).tasks)
          ..add(event.task)
          ..sort((a, b) {
            if (a.dueDate == null && b.dueDate == null) {
              return 0;
            } else if (a.dueDate == null) {
              return 1;
            } else if (b.dueDate == null) {
              return -1;
            } else {
              return a.dueDate!.compareTo(b.dueDate!);
            }
          });
        emit((state as _Success).copyWith(tasks: list));
      } else {
        emit(_Success(tasks: [event.task], hasReachedMax: true));
      }
    } on Object catch (error) {
      emit(_Failure(error: error));
      rethrow;
    }
    final stop = DateTime.now();
    print(stop.difference(start));
  }
  Future<void> _onUpdateReturnTask(
    _UpdateReturnedTask event,
    Emitter<TasksState> emit,
  ) async {
    try {
      final updatedTasks = (state as _Success).tasks.map((task) {
        if (task.taskId == event.task.taskId) {
          return task = event.task;
        }
        return task;
      }).toList();
      await _tasksRepository.updateTask(
          task: updatedTasks
              .singleWhere((task) => task.taskId == event.task.taskId));
      emit((state as _Success).copyWith(tasks: updatedTasks));
    } on Object catch (error) {
      emit(_Failure(error: error));
      rethrow;
    }
  }
}
```

ДОДАТОК Б

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ДНІПРОВСЬКА ПОЛІТЕХНІКА»

Факультет інформаційних технологій
Кафедра програмного забезпечення комп'ютерних систем

ВІДГУК

Наукового керівника Мещерякова Леоніда Івановича, доктора технічних наук, професора, професора кафедри ПЗКС
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання, посада, місце роботи)

На кваліфікаційну роботу

студента Петрушенка Артема В'ячеславовича
(прізвище, ім'я, по батькові)

курсу II групи 121м-22-3

спеціальності 121 Інженерія програмного забезпечення

на тему Дослідження швидкодії мобільних застосунків у залежності від різних методів управління станом на прикладі фреймворку Flutter

Актуальність теми обґрунтована розвитком наукової діяльності, дослідженням методів управління станом на основі фреймворку Flutter для розвитку програмної реалізації мобільних застосунків.

Мета досліджень полягає у підвищенні та вдосконаленні процесів розробки мобільних застосунків шляхом дослідження швидкодії використання методів управління станом BLoC, MobX і Redux фреймворку Flutter.

Коротка характеристика розділів роботи. Перший розділ роботи присвячений методичній частині, у якій розглядаються основні теоретичні аспекти розробки мобільних застосунків з використанням фреймворків. Другий розділ роботи полягає у проєктуванні архітектури мобільної системи «TaskHub». Третій розділ роботи описує основні алгоритми та методи розробки мобільного застосунку, надає результати дослідження швидкодії методів управління станом.

Практичне значення роботи полягає у використанні результатів дослідження в майбутніх проєктах у розробці мобільних застосунках при виборі методу управління станом, а також у подальшому вдосконаленні розробки мобільних систем.

Зауваження та недоліки. У роботі порівнюються методи управління станом тільки одного фреймворку.

Висновки та оцінка. Кваліфікаційна робота магістра заслуговує оцінки «відмінно», а Петрушенко Артем В'ячеславович присвоєння кваліфікації «магістра».

Науковий керівник Мещеряков Леонід Іванович, професор кафедри ПЗКС
(прізвище, ім'я, по батькові, посада, місце роботи)

«___» _____ 2023 р. _____
(підпис)

ДОДАТОК В

РЕЦЕНЗІЯ
на кваліфікаційну роботу

студента Петрушенка Артема В'ячеславовича
(прізвище, ім'я, по батькові)

курсу II групи 121м-22-3

кафедри програмного забезпечення комп'ютерних систем

спеціальності 121 Інженерія програмного забезпечення

Тема роботи Дослідження швидкодії мобільних застосунків у залежності від різних методів управління станом на прикладі фреймворку Flutter

Стисла характеристика розділів роботи. Перший розділ присвячений аналізу теоретичних і методологічних аспектів досліджуваної задачі та проблематиці розробки кросплатформних мобільних застосунків. У другому розділі студент надає роз'яснення щодо проєктування системних зв'язків архітектури методів управління станом фреймворку Flutter. У третьому розділі розглядається розробка алгоритмів мобільного застосунку, результати роботи та порівняння методів управління станом за обраними метриками.

Пропозиції, внесені студентом, рівень їх наукового обґрунтування впливають на пришвидшення розробки кросплатформних мобільних застосунків.

Практичне значення роботи полягає в оптимізації мобільної розробки, використовуючи методи управління станом фреймворку Flutter.

Якість оформлення роботи виконана на високому рівні, у відповідності з вимогами та повністю відповідає поставленим задачам.

Недоліки в роботі. Відсутність дослідження методів управління станом сторонніх фреймворків.

Загальний висновок отримані результати є закінченою науково-дослідною роботою, викликають науковий інтерес і демонструють здатність Петрушенка Артема В'ячеславовича до самостійного аналізу, проведення наукової роботи та

вміння розробки програмного забезпечення.

Оцінка магістерської роботи «відмінно»

Рецензент Коротенко Григорій Михайлович, доцент, професор кафедри ІТКІ НТУ «Дніпровська політехніка»

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання, посада, місце роботи)

«____» _____ 2023 р. _____

(підпис)

ДОДАТОК Г

Перелік файлів на диску

Ім’я файла	**Опис**
Пояснювальні документи	
121м-22-3_ПетрушенкоАВ.docx	Пояснювальна записка роботи. Документ Word
121м-22-3_ПетрушенкоАВ.pdf	Пояснювальна записка роботи. Документ PDF
Програма	
taskhub.zip	Архів. Містить коди програми і откомпільовану програму
Презентація	
121м-22-3_ПетрушенкоАВ.pptx	Презентація роботи