МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Харківський національний університет імені В. Н. Каразіна

Факультет: **ННІ Каразінський банківський інститут**
Кафедра: **Інформаційних технологій та математичного моделювання**
Спеціальність: **122 Комп'ютерні науки**
Освітня програма: **Комп'ютерні науки та інформаційні технології в бізнесі**

Група: АК-21М **денна форма навчання**

КВАЛІФІКАЦІЙНА МАГІСТЕРСЬКА РОБОТА

на тему:

«ДОСЛІДЖЕННЯ ШВИДКОДІЇ АЛГОРИТМІВ МЕТОДІВ КЕРУВАННЯ СТАНОМ MOBX І REDUX ФРЕЙМВОРКІВ FLUTTER ТА REACT NATIVE»

ЗА НАКАЗОМ № 4601-5/3045 ВІД 25 ВЕРЕСНЯ 2024 РОКУ

здобувача вищої освіти **Кузнецов Віталій Сергійович**

Робота допущена до захисту в ЕК
протокол кафедри ІТММ №4 від 30.11.2024 р.

Завідувач кафедри
к. п. н., доцент
_____ Н. І. Стяглик

Науковий керівник
д.е.н., к.т.н., професор
_____ Б.В. Самородов

м. Харків 2024 р.

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Харківський національний університет імені В. Н. Каразіна

Факультет навчально-науковий інститут "Каразінський банківський інститут"
Кафедра інформаційних технологій та математичного моделювання
Рівень вищої освіти другий (магістерський)
Спеціальність 122 Комп'ютерні науки
Освітня програма Комп'ютерні науки

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ **Н. І. Стяглик**
Підпис ініціали, прізвище

"25" вересня 2024 року

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ (ПРОЄКТ)

Кузнецова Віталія Сергійовича
(прізвище, ім'я, по батькові студента)

1. Тема роботи «ДОСЛІДЖЕННЯ ШВИДКОДІЇ АЛГОРИТМІВ МЕТОДІВ КЕРУВАННЯ СТАНОМ MOBX І REDUX ФРЕЙМВОРКІВ FLUTTER ТА REACT NATIVE»

керівник роботи д.е.н., к.т.н., професор Самородов Б.В.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від "25" вересня 2024 року № 4601-5/3045

2. Строк подання студентом роботи 20 листопада 2024 року

3. Перелік питань, які потрібно розробити:
У розділі 1: дослідити використання фреймворків Flutter та React Native у розробці мобільних застосунків та поставити вимоги на розробку.
У розділі 2: виконати проєктування внутрішніх процесів мобільного застосунку.
У розділі 3: виконати розробку алгоритмів, порівняти їх та дослідити за обраними метриками найкращі результати.

4. План роботи

№ з/п	Назви етапів роботи
1	Вибір здобувачем теми кваліфікаційної магістерської роботи
2	Затвердження плану і завдання кваліфікаційної магістерської роботи
3	Здача кваліфікаційної магістерської роботи керівнику
4	Підпис кваліфікаційної магістерської роботи керівника
5	Підпис кваліфікаційної магістерської роботи у нормоконтролера
6	Допуск завідувачем кафедри до захисту кваліфікаційної магістерської роботи
7	Захист кваліфікаційної магістерської роботи

5. Дата видачі завдання 25 вересня 2024 року

Студент _____ В.С. Кузнецов
підпис ініціали, прізвище

Керівник роботи _____ Б.В.Самородов
підпис ініціали, прізвище

РЕФЕРАТ
НА КВАЛІФІКАЦІЙНУ МАГІСТЕРСЬКУ РОБОТУ
«ДОСЛІДЖЕННЯ ШВИДКОДІЇ АЛГОРИТМІВ МЕТОДІВ КЕРУВАННЯ СТАНОМ MOBX І REDUX ФРЕЙМВОРКІВ FLUTTER ТА REACT NATIVE»
Кузнецова Віталія Сергійовича

Кваліфікаційна магістерська робота містить: 79 сторінок, 15 таблиць, 41 рисунків, 0 формул, список літератури з 51 найменування.

Об'єкт дослідження: швидкодія методів управління станом MobX і Redux у двох різних фреймворках.

Предмет дослідження: алгоритми та методи розробки методів управління станом MobX і Redux у двох різних фреймворках Flutter та React Native.

Мета кваліфікаційної магістерської роботи: результати дослідження працездатності методів управління станом MobX і Redux у двох різних фреймворках Flutter та React Native задля підвищення ефективності розробки в IT бізнесі.

Завдання кваліфікаційної магістерської роботи:

- у першому розділі дослідити використання фреймворків Flutter та React Native у розробці мобільних застосунків та поставити вимоги на розробку;

- у другому розділі виконати проєктування внутрішніх процесів мобільного застосунку;

- у третьому розділі виконати розробку алгоритмів, порівняти їх та дослідити за обраними метриками найкращі результати.

Актуальність дослідження полягає у підвищенні ефективності розробки програмних застосунків у IT бізнесі.

За результатами дослідження сформульовані програмні алгоритми, результати порівняльного аналізу фреймворків і методів управління станом.

Практична новизна: упровадження нових результатів дослідження за метриками, які надають розуміння, який фреймворк і метод управління станом було б ефективніше використовувати в процесі розробки мобільних систем.

Одержані результати можуть бути використані в сфері IT та бізнесі подібного роду для підвищення ефективності розробки програмних систем.

КЛЮЧОВІ СЛОВА: FLUTTER, MOBX, REACT NATIVE, REDUX, АЛГОРИТМИ МОБІЛЬНОЇ РОЗРОБКИ, МОБІЛЬНА РОЗРОБКА.

ABSTRACT
AT QUALIFICATION MAGISTER WORK
«RESEARCH OF THE SPEED OF ALGORITHMS OF CONTROL METHODS MOBX AND REDUX FLUTTER AND REACT NATIVE FRAMES»
Vitalii Kuznetsov

The master's thesis contains 79 pages, 15 table, 41 drawings, 0 formulas, a list of references of 51 titles.

The object of the study is performance of MobX and Redux state management methods in two different frameworks.

The subject of the study is algorithms and methods of developing MobX and Redux state management methods in two different frameworks, Flutter and React Native.

The purpose of the master's qualification work: to investigate and compare the performance results of MobX and Redux state management methods in two different frameworks, Flutter and React Native, in order to increase the efficiency of development in IT business.

The tasks of the master's qualification work are:

- in the first section to investigate the use of Flutter and React Native frameworks in the development of mobile applications and set requirements for development;

- in the second section to design the internal processes of the mobile application;

- in the third section to perform the development of algorithms, compare them and investigate the best results based on the selected metrics.

The relevance of the study in increasing the efficiency of the development of software applications in the IT business.

According to the results of the research, software algorithms, the results of a comparative analysis of frameworks and state management methods were formulated.

Main theoretical provisions on the topic of the implementation of new research results on metrics that provide insight into which framework and state management method would be more effective to use in the development process of mobile systems.

The results obtained can be used in the field of IT and business of this kind to improve the efficiency of the development of software systems.

KEYWORDS: FLUTTER, MOBX, REACT NATIVE, REDUX, MOBILE DEVELOPMENT ALGORITHMS, MOBILE DEVELOPMENT.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧОК, СИМВОЛІВ І ТЕРМІНІВ 9
ВСТУП 10
РОЗДІЛ 1. ТЕОРЕТИКО МЕТОДИЧНИЙ АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ 12
1.1. Використання фреймворків Flutter та React Native у розробці мобільних застосунків 12
1.1.1. Використання фреймворку Flutter у розробці мобільних застосунків 12
1.1.2. Використання фреймворку React Native у розробці мобільних застосунків 14
1.1.3. Порівняння фреймворків у розробці мобільних застосунків 15
1.2. Використання методів управління станом MobX і Redux у розробці мобільних застосунків 17
1.2.1. Методи управління станом MobX і Redux фреймворку Flutter у розробці мобільних застосунків 17
1.2.2. Методи управління станом MobX і Redux фреймворку React Native у розробці мобільних застосунків 20
1.2.3. Порівняння методів управління станом MobX і Redux у фреймворках Flutter і React Native у розробці мобільних застосунків 22
1.3. Методика дослідження використання фреймворків Flutter та React Native 26
1.4. Визначення вимог до розробки мобільного застосунку 28
1.4.1. Вимоги до мобільного застосунку 28
1.4.2. Фреймворки та мови програмування 29
1.4.3. Методи порівняння фреймворків і методів управління станом 29
1.5. Висновки до розділу 1 30

РОЗДІЛ 2. ПРОЄКТУВАННЯ ВНУТРІШНІХ ПРОЦЕСІВ РОЗРОБКИ МОБІЛЬНОГО ЗАСТОСУНКУ 31
2.1. Проєктування діаграми класів 31
2.1.1. Проєктування діаграми класів для фреймворку Flutter 31
2.1.2. Проєктування діаграми класів для фреймворку React Native 34
2.2. Проєктування діаграми станів 35
2.3. Проєктування діаграми діяльності 38
2.4. Проєктування діаграми розгортання 41
2.5. Висновки до розділу 2 43
РОЗДІЛ 3. ПРОГРАМНА РОЗРОБКА МОБІЛЬНОГО ЗАСТОСУНКУ ТА ПОРІВНЯННЯ РЕЗУЛЬТАТІВ ПРАЦЕЗДАТНОСТІ МЕТОДІВ КЕРУВАННЯ СТАНОМ MOBX І REDUX ФРЕЙМВОРКІВ FLUTTER ТА REACT NATIVE 44
3.1. Програмна реалізація мобільного застосунку з використанням фреймворку Flutter 44
3.1.1. Програмна реалізація користувальницького інтерфейсу 44
3.1.2. Програмна реалізація логіки з використанням методу керування станом MobX 48
3.1.3. Програмна реалізація логіки з використанням методу керування станом Redux 51
3.1.4. Програмна реалізація зберігання даних 53
3.2. Програмна реалізація мобільного застосунку з використанням фреймворку React Native 55
3.2.1. Програмна реалізація користувальницького інтерфейсу 55
3.2.2. Програмна реалізація логіки з використанням методу керування станом MobX 57
3.2.3. Програмна реалізація логіки з використанням методу керування станом Redux 61
3.2.4. Програмна реалізація зберігання даних 63

3.3. Порівняння методів керування станом MobX і Redux фреймворків Flutter та React Native 64
3.3.1. Порівняння результатів швидкодії 64
3.3.2. Порівняння результатів часу відгуку 67
3.3.3. Порівняння результатів використання пам’яті 69
3.3.4. Порівняння результатів використання енергії 70
3.4. Висновки до розділу 3 71
ВИСНОВКИ 72
ПЕРЕЛІК ПОСИЛАНЬ 74
ДОДАТКИ 80

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧОК, СИМВОЛІВ І ТЕРМІНІВ

API – Application Programming Interface

CRUD – Create, Read, Update, Delete

UI – User Interface, користувальницький інтерфейс

БД – база даних

ІТ – інформаційні технології

ВСТУП

За останні роки мобільні системи охопили весь світ і стали невід'ємною частиною майже кожного. Вони надають користувачам досить широкий спектр можливостей – від звичайного спілкування та розваг до керування бізнесом і фінансами. Дуже важливим аспектом є вибір інструментарію для розробки мобільних застосунків, що будуть забезпечувати високу продуктивність і зручність у використанні. Серед таких особливе місце займають фреймворки Flutter і React Native. Вони дозволяють створювати багатоплатформні системи з єдиною базою програмного коду. Управління керування станом є ключовим компонентом архітектури мобільних застосунків від якого залежить їх швидкодія та чуйність. Існують різні бібліотеки та методи керування станом, серед яких можна виділити MobX та Redux. Перший дозволяє реактивний підхід до керуванню станом, а другий – заснований на концепції передбачуваності та незмінності стану [14, 37].

Актуальністю роботи є підвищення ефективності розробки мобільних застосунків у сфері бізнесу в інформаційних технологіях (IT). Підвищення вимог до результативності процесу дасть розуміння до різних підходів к управління станом. Аналіз продуктивності методів управління станом у різних фреймворках дозволить створити універсальні рекомендації для розробки систем під різноманітні платформи. Ефективне керування станом дозволить економити ресурси, що важливо для мобільних пристроїв. Швидкий та плавний застосунок напряму впливає на задоволеність користувачів та їх лояльність.

Метою роботи є результати дослідження працездатності методів управління станом MobX і Redux у двох різних фреймворках Flutter та React Native задля підвищення ефективності розробки в IT бізнесі.

Виходячи з актуальності та мети роботи, завдання на дослідження наступні:

1) дослідити використання фреймворків Flutter і React Native в розробці

мобільних систем;

2) дослідити використання методів управління станом MobX і Redux в розробці мобільних систем;

3) визначити методи дослідження для порівняння методів управління станом у фреймворках;

4) визначити вимоги до розробки мобільного застосунку;

5) спроєктувати внутрішні процеси мобільної системи;

6) виконати програмну реалізацію на чотирьох мобільних застосунків, використовуючи методи управління станом і фреймворки;

7) виконати порівняльний аналіз, використовуючи обрані методи порівняння;

8) зробити відповідні висновки.

Об’єктом дослідження є швидкодія методів управління станом MobX і Redux у двох різних фреймворках.

Предметом дослідження є алгоритми та методи розробки методів управління станом MobX і Redux у двох різних фреймворках Flutter та React Native.

Робота складається із вступу, трьох частин та висновку.

У вступі наведена актуальність, мета дослідження, представлені завдання на роботу, наведені об’єкт і предмет дослідження.

У першому розділі проведено теоретичне дослідження предметної області, визначено використання фреймворків і методів управління станом у розробці багатоплатформних застосунків.

У другому розділі наведено проєктування внутрішніх процесів системи.

У третьому розділі описані програмні алгоритми розробки системи та проведено порівняльний аналіз з результатами обчислень швидкодії.

Висновки підкреслюють виконані задачі за кожним розділом і більш точно узагальнюють результати проведеної роботи.

РОЗДІЛ 1.
ТЕОРЕТИКО МЕТОДИЧНИЙ АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Використання фреймворків Flutter та React Native у розробці мобільних застосунків

1.1.1. Використання фреймворку Flutter у розробці мобільних застосунків

Flutter являє собою фреймворк із відкритим вихідним кодом для створення нативних застосунків, як для мобільних пристроїв, так і для веб і для настільних із однією кодовою базою, що є однієї з ключових переваг. Така можливість значно скорочує час та витрати на розробку, бо розробникам не потрібно писати та підтримувати окремий код для кожної платформи. Більш того, застосунки на Flutter компілюються а нативний код, що забезпечує високу продуктивність та плавність роботи. Функція «гарячого перезавантаження» робить процес розробки на Flutter дуже зручною. Вона дозволяє одразу ж побачити результати змін у коді без необхідності перезавантажувати застосунок. Така можливість прискорює процес програмної реалізації та відладки, роблячи його більш інтерактивним і ефективним [31, 40, 45].

Фреймворк надає великий набір готових віджетів, які можна налаштовувати та комбінувати для створення унікальних користувальницьких інтерфейсів. Усі віджети відповідають рекомендаціям від розробників операційних систем Android та iOS, тому він чудово підходить для розробки багатоплатформних систем. Також фреймворк зручний для прототипування, дозволяючи швидко розроблювати та тестувати нові ідеї та інтерфейси [7, 22, 40].

Одним із прикладів успішного використання Flutter є розробка застосунків для електронної комерції. Платформи комерції часто потребують

високої продуктивності та привабливого користувальницького інтерфейсу для забезпечення кращого досвіду. Фреймворк дозволяє розробникам створювати інтерактивні інтерфейси, які працюють швидко та надійно, як на Android, так і на iOS [3, 48].

Приклад такої системи наведено на рисунку 1.1.

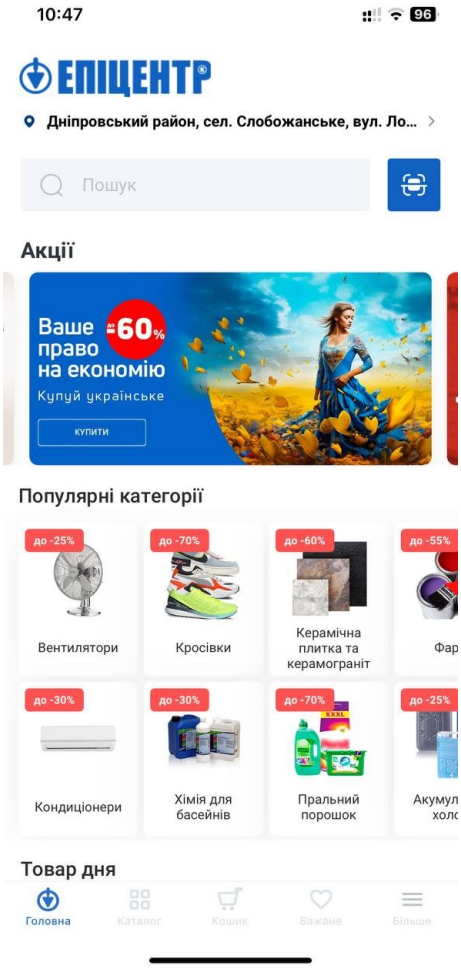


Рис. 1.1. Мобільний застосунок електронної комерції «EpicentrK» [2]

Додатковою перевагою у фреймворку Flutter є інтеграція з різними системами backend та Application Programming Interface (API). Наприклад, він може взаємодіяти з Representational State Transfer ful (RESTful) та GraphQL API, що дозволяє легко взаємодіяти з серверними компонентами та забезпечити надійну та безпечну передачу даних. Дані факти роблять фреймворк гарним вибором для створення складних застосунків, що

потребують постійного обміну даними із сервером. Крім того, розробники можуть додавати власні функції та масштабувати застосунки завдяки модульному підходу у фреймворку. Це особливо важливо для стартапів та компаній, які планують розширювати свої системи та додавати нові можливості за зростом свого бізнесу [39, 42].

Отож, як можна побачити, Flutter являє собою потужний і гнучкий інструмент для розробки мобільних застосунків. Його переваги надають можливість швидко та ефективно реалізовувати програмні системи, надаючи подальший розвиток для них.

1.1.2. Використання фреймворку React Native у розробці мобільних застосунків

Фреймворк React Native, як і попередній, дозволяє створювати кросплатформні застосунки на базі єдиного коду. Серед переваг можна також виділити наступні фактори [1, 25]:

- швидке оновлення коду без необхідно перезавантаження застосунку;
- підтримка нативних модулів, що покращує продуктивність;
- надання доступу до специфічних функцій пристрою тощо.

Фреймворк використовується в різних галузях. Наприклад, соціальні мережі, електронна комерція, навчальні платформи, бізнес-застосунки тощо. Також, одним із значних переваг React Native є можливість створювати високоякісні користувальницькі інтерфейси. У фреймворку є можливість для стилізації компонентів, що дозволяє створювати складні та привабливі інтерфейси, що спираються на натив. Така можливість надає користувачам спиратися на власний досвід роботи з програмами [11, 17].

Фреймворк інтегрується з різними сервісами та API, що дозволяє легко додавати функціональність реального часу тощо. React Native має можливість співпрацювати з різними інструментами розробки та налагодження. Також він

має різні інструменти для спостерігання стану застосунку, продуктивності та налагодженню помилок. Це значно прискорює процес програмної реалізації та виявлення і усунення помилок [47, 12].

React Native є потужним інструментом для розробки багатоплатформних застосунків, який дозволяє значно скоротити час реалізації. Він надає високу продуктивність та можливість використання нативних компонентів. Однак, у більш складних та вимогливих системах він може потребувати додаткової оптимізації.

1.1.3 Порівняння фреймворків у розробці мобільних застосунків

Для порівняння обох фреймворків були використані характеристики та їх опис. Порівняльний аналіз наведений у таблиці 1.1.

Таблиця 1.1

Порівняльна характеристика фреймворків Flutter і React Native

Характеристика	Flutter	React Native
1	2	3
Мова програмування	Dart	JavaScript
Архітектура	Власний двигун рендерінгу	Використання нативних компонентів
Підтримка платформ	iOS, Android, вебсистеми, десктопні	iOS, Android, вебсистеми з додатковими фреймворками
Компоненти користувальницького інтерфейсу	Кастомні від Material Design та Cupertino	Нативні
Продуктивність	Висока, власний рендерінг	Середня, використовує нативні мости

Продовження таблиці 1.1

1	2	3
Кросплатформність	Повна, під усі системи	Повна, але необхідна адаптація під кожну платформу
Час збірки	Довга через повну збірку	Швидка, але залежить від коду
Розмір застосунків	Від малих до великих	Від малих до середніх
Плагіни та бібліотеки	Середня підтримка	Велика підтримка
Екосистема інструментів	Повна інтеграція з Google інструментами	Велика підтримка сторонніх інструментів
Підтримка анімацій	Власні засоби для анімації	Може потребувати додаткових бібліотек
Тестування	Підтримка вбудованих засобів	Багато сторонніх рішень
Структура проєкту	Проста та зрозуміла	Залежить від підходу розробки
Швидкість розробки	Швидка, завдяки потужним інструментам	Швидка, завдяки наявності множини бібліотек
Стабільність	Висока	Висока
Інтеграція автоматизації процесів	Підтримується, але необхідне налаштування	Підтримується
Підтримка доповненої та віртуальної реальностей	Обмежена	Обмежена, але можлива через сторонні бібліотеки

Джерело: розробка автора

Таблиця надає більш точно оцінити всі переваги та недоліки фреймворків та обрати оптимальну платформу для розробки застосунків.

1.2. Використання методів управління станом MobX і Redux у розробці мобільних застосунків

1.2.1. Методи управління станом MobX і Redux фреймворку Flutter у розробці мобільних застосунків

Методи управління станом MobX і Redux фреймворку Flutter при розробці мобільних застосунків мають свої унікальні підходи та особливості.

Основні концепції MobX у Flutter наведені в таблиці 1.2.

Таблиця 1.2

Основні концепції методу управління станом MobX у Flutter

Концепція	Опис
Реактивність	Заснований на концепції реактивного програмування. Автоматично відстежує зміни в станах та оновлює відповідні представлення, коли дані змінюються
Спостережувальні стану	Використання спостережувальних станів (observable), які можуть бути простими змінними, списками або картами. Вони можуть бути змінені в будь-якій частині застосунку
Дії (actions)	Зміни в стані повинні бути виконані через дії, що робить зміни в стані явними та передбачуваними
Реакції (reactions)	Реакції (спостерігачі, реакції, обчислювальні властивості) автоматично оновлюються, коли спостерігаються зміни в стані

Джерело: розробка автора

Перевагами MobX є наступні [38, 41]:

- простий і зручний API;

- висока продуктивність завдяки автоматичному керуванню оновленнями користувальницького інтерфейсу;

- підходить для розробників, які знайомі з концепцією реактивного програмування.

Недоліками MobX є наступні [38, 41]:

- надає велику гнучкість, що може призвести до непослідовності у великих командах або проєктах;

- має невелику популярність та підтримку в спільноті.

Основні концепції Redux у Flutter наведені в таблиці 1.3.

Таблиця 1.3

Основні концепції методу управління станом Redux у Flutter

Концепція	Опис
Єдине сховище (store)	Усі стани застосунку зберігаються в одному глобальному сховищі
Reducers	Визначають, як стан застосунку змінюється у відповідь на дії. Вони є чистими функціями, які приймають поточні стани та дії та повертають новий стан
Дії (actions)	Дії є об’єктами, які описують, що відбулося в застосунку. Вони відправляються для оновлення стану
Milddlewares	Дозволяють додавати додаткову логіку при відправці дій (наприклад, асинхронні запити)

Джерело: розробка автора

Перевагами Redux є наступні [19, 51]:

- через використання чистих функцій та єдиного сховища, стан застосунку стає передбачуваним;

- надає строгу архітектуру, що допомагає в підтримці великих проєктів та командної роботи;

- має велику спільноту та множину додаткових бібліотек та інструментів.

Недоліками Redux є наступні [19, 51]:

- може бути складний у засвоєнні для тих, хто починає через необхідність писати шаблонній код;

- потребує більше коду для налаштування та управління станом.

У таблиці 1.4 наведено застосування даних методів управління станом у фреймворку Flutter.

Таблиця 1.4

Застосування MobX і Redux у Flutter

Застосування	MobX Flutter	Redux Flutter
Основна бібліотека	mobx	redux
Пакети	flutter_mobx, який надає віджети, що оновлюють автоматично при змінах станів, за якими спостерігається	flutter_redux, який надає віджети для роботи зі сховищем та конектори для зв’язку станів з віджетами
Розмір проєктів	Підходить для невеликих і середніх проєктів, де важлива простота та гнучкість і динамічне оновлення інтерфейсу	Підходить для крупних проєктів із великою кількістю станів та складною логікою, а також чітка структура та передбачуваність станів

Джерело: розробка автора

1.2.2. Методи управління станом MobX і Redux фреймворку React Native у розробці мобільних застосунків

Методи управління станом MobX і Redux фреймворку React Native є популярними рішеннями, що забезпечують ефективне керування станом.

Основні концепції MobX у React Native наведені в таблиці 1.5.

Таблиця 1.5

Основні концепції методу управління станом MobX у React Native

Концепція	Опис
Observable State	Стан застосунку робиться спостережуваним за допомогою декораторів або функцій
Реакції (reactions)	Реакції автоматично оновлюються при зміні залежностей
Дії (actions)	Зміни стану відбуваються через дії, які змінюють стан та автоматично повідомляють спостережувача
Computed Values	Обчислювальні значення автоматично перераховуються тільки при зміні використовуваних даних у них

Джерело: розробка автора

Перевагами MobX React Native є [15, 23]:

1) не потребує шаблонного коду, має більш просте API, що надає більш швидше засвоїтися в навчанні;

2) потребує менше додаткових абстракцій (дій, ред'юсерів) та код виглядає більш природньо;

3) зміни в стані автоматично відслідковуються, що спрощує реактивне програмування;

4) інтегрується з компонентами фреймворку, зменшуючи необхідність додаткових бібліотек для зв'язку.

Недоліки MobX React Native є [15, 23]:

1) надає менший контроль над змінами стану застосунку, що може призвести до складностей у великих проєктах;

2) має меншу кількість розширень і інструментів підтримки, що може ускладнювати пошуки рішень складних завдань.

Основні концепції Redux у React Native наведені в таблиці 1.6.

Таблиця 1.6

Основні концепції методу управління станом Redux у React Native

Концепція	Опис
Сховище	Центральне сховище, що містить сан всього застосунку
Дії (actions)	Дії є об’єктами, які направляють дані від застосунку до сховища
Reduces	Функції, які оброблюють дії та змінюють стан у сховищі
Selectors	Функції, які обирають частини стану зі сховища для використання в компонентах

Джерело: розробка автора

Перевагами Redux React Native є [20, 35]:

1) строгий контроль над станом, який забезпечує однонаправлений потік даних та чіткі правила зміну стану, що робить його переважним для більшості складних і великих проєктів;

2) завдяки однозначному потоку даних та використання чистих функцій reducers, відладка та тестування простіше;

3) має велику кількість розширень, middleware, інструментів для розробки, що спрощує інтеграцію та розширення функціоналу застосунку.

Недоліками Redux React Native є [20, 35]:

1) для реалізації простого функціоналу може виникнути необхідність написати більше програмного коду та використовувати більше шаблонів;

2) через архітектурні концепції починаючим розробникам може біти складно засвоїти фреймворк.

1.2.3. Порівняння методів управління станом MobX і Redux у фреймворках Flutter і React Native у розробці мобільних застосунків

Для більшого розуміння відмінностей методів управління станом у фреймворках у таблиці 1.7 наведена порівняльна характеристика.

Таблиця 1.7

Порівняльна характеристика MobX і Redux у Flutter і React Native

Характеристика	Flutter MobX	Flutter Redux	React Native MobX	React Native Redux
1	2	3	4	5
Керування станом	Реактивне програмування, використання анотацій @observable, @computed для керування змінами стану	Централізоване сховище стану, одно направлений потік даних через reduces та actions	Реактивне програмування, використання анотацій @observable, @computed	Централізоване сховище стану, одно направлений потік даних через reduces і actions
Використання	Популярний у спільноті за простоту та гнучкість	Використовується для керування складним станом застосунку, що потребує чітких оновлень стану	Популярний, особливо з React компонентами для спрощення керування станом	Передбачений для складних застосунків, де необхідне чітке керування станом
Складність	Може складно засвоюватися та використовуватися завдяки	Має більш високий поріг входу через необхідність	Засвоєння залежить від знайомства з реактивним	Потребує розуміння одно направленого потоку даних та

	більшій кількості концепцій та абстракцій	розуміння одно направленого потоку даних та middleware	програмування м	використання middleware, що може бути складним для новачків

Продовження таблиці 1.7

1	2	3	4	5
Продуктивність	Забезпечує високу продуктивність завдяки реактивним оновленням стану	Може мати невеликі затримки на продуктивність через необхідності проходу через багаторівневі редуктори для оновлення стану	Забезпечує високу продуктивність завдяки ефективному управлінню станом та реактивними оновленнями	Може мати затримки продуктивності через необхідність проходу через багаторівневі редуктори
Спільнота та підтримка	Активна спільнота з різними пакетами та інструментами підтримки	Широка спільнота з множиною плагінів та інтеграцій для спрощення розробки	Широка спільнота з багатьма ресурсами та бібліотеками, що допомагають в інтеграції та розробці	Широко використовується, з активною спільнотою та ресурсами для підтримки
Використання в інтерфейсі	Інтеграція з віджетами через анотації та реактивні стріми для автоматичного оновлення інтерфейсу при зміні стану	Потребує явного зв’язку з інтерфейсом через конектори, на що може знадобитися додатковий код для оновлення компонентів	Інтеграція з React компонентами, підтримка автоматичного оновлення через реактивні стріми та анотації	Потребує зв’язок з компонентами інтерфейсу, але є можливість автоматично відписатися на зміни
Гнучкість і розширюв	Надає високу гнучкість завдяки можливості	Заснований на чіткій структурі з одно	Надає гнучкість у виборі підходів до	Структурований для чіткого управління

а ність	використання реактивних та імперативних підходів у залежності від вимог	направленим потоком даних, що може зробити його менш гнучким для певних сценаріїв	управління станом та інтеграції з іншими бібліотеками	станом, що може обмежити гнучкість у деяких сценаріях
Інтеграція з асинхронними операціями	Спрощена завдяки підтримці реактивних стримів та можливості автоматичного оновлення стану	Необхідне використання middleware або додаткових бібліотек для керування асинхронними операціями	Підтримка операцій через реактивні стріми та анотації, що полегшує інтеграцію з мережевими запитами тощо	Потребує використання middleware для керування асинхронними операціями та оновлення стану після завершення

Джерело: розробка автора

Дана характеристика (табл. 1.7) надає зрозуміти, що використання методів управління станом у різних фреймворках у деяких моментах є однаковим, а у деяких – різняться зовсім. Графічно можна побачити схеми роботи методів управління станів, що наведені на рисунках 1.2-1.3.

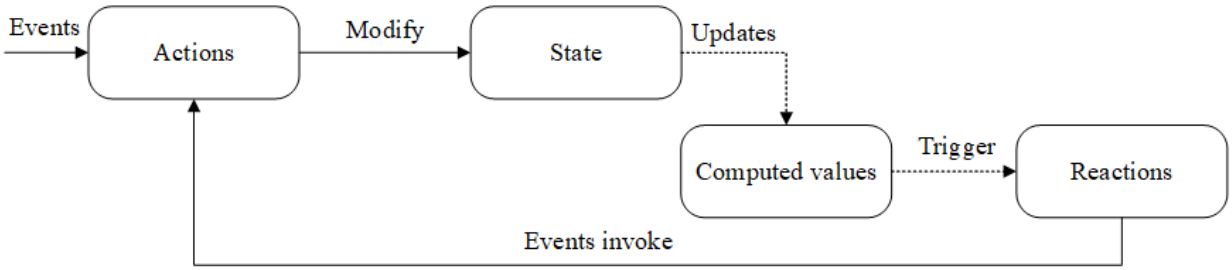


Рис. 1.2. Схема роботи методу управління станом MobX

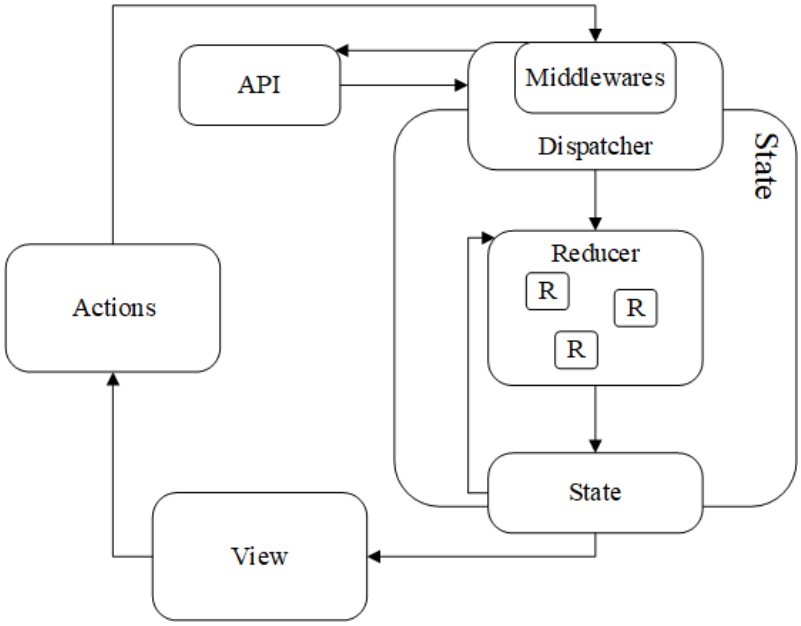


Рис. 1.3. Схема роботи методу управління станом Redux

Як можна побачити із наведених схем (рис. 1.2-1.3) методи управління станом мають однакові властивості, але вони є різні в розробці та її аспектах.

1.3. Методика дослідження використання фреймворків Flutter та React Native

Для об’єктивного порівняння важливо обрати критерії, які відображають різні аспекти розробки та експлуатації застосунку. У таблиці 1.8 наведені критерії та підкритерії методів порівняння фреймворків.

Таблиця 1.8

Методика порівняння фреймворків Flutter та React Native

Підкритерій	Опис	Інструменти та методи
1	2	3
Швидкодія		
Швидкість запуску застосунку	Час, необхідний для повного завантаження застосунку з моменту його запуску до появи головного екрану	Android Studio Profiler, Xcode Instruments, Flutter DevTools, React Native Debugger, запуск та вимір часу виконання тестових операцій
Швидкість виконання операцій	Час, витрачений на виконання різних операцій	Використання профілювання для аналізу вузьких місць у продуктивності
Рендерінг інтерфейсу	Час, необхідний для відмальовування інтерфейсу, включаючи відгук на взаємодію користувача	
Час відгуку		
Затримка натиснень	Час від моменту натиснення кнопки користувачем до виконання відповідної дії та відображення результату	Chrome DevTools, Flutter DevTools, тестиування відгуку на користувальницькі дії, вимір затримки натиснень
Обробка тестів	Час відгуку на жести (свайпи, щипкі, довгі натиснення)	
Час відгуку		

Перехід між екранами	Час, необхідний для переходу від одного екрану до іншого, включаючи анімації та завантаження даних	
Використання пам’яті		
Використання оперативної пам’яті	Об’єм пам’яті, що використовується застосунком під час його роботи в різних станах	Android Studio Memory Profiler, Xcode Instruments, Flutter DevTools, React Native Debugger; моніторинг використання пам’яті
Витік пам’яті	Наявність та кількість витоків пам’яті, що призводять до збільшенню використовуваної	Використання інструментів для виявлення та усунення витоків пам’яті

Продовження таблиці 1.8

1	2	3
	пам’яті та зниженню продуктивності застосунку з часом	
Використання енергії		
Споживання енергії в активному режимі	Кількість енергії, використовуваної при активному використанні	Android Studio Energy Profiler, Xcode Energy Log, інструменти для моніторингу батареї на реальних пристроях, вимір енергоспоживання при різних сценаріях використання
Споживання енергії в фоновому режимі	Енергоспоживання при роботі застосунків у фоновому режимі	
Ефективність використання апаратних ресурсів	Оцінка, наскільки оптимально застосунок використовує апаратні ресурси для зниження навантаження на батарею	

Джерело: розробка автора

Дана таблиця (табл. 1.8) дозволить структуровано представити критерії оцінки продуктивності, часу відгуку, використання пам’яті та енергії для порівняння Flutter та React Native.

1.4. Визначення вимог до розробки мобільного застосунку

Визначення вимог є важливим етапом при розробці будь-якої системи. Перед розробкою необхідно визначити вимоги до застосунку, інструменти для розробки та методи порівняльного аналізу [16, 49].

1.4.1. Вимоги до мобільного застосунку

Вимогами до мобільних застосунків є наступні:

- мобільних застосунків повинно бути чотири з такими комбінаціями: MobX Flutter, Redux Flutter, MobX React Native, Redux React Native;
- усі застосунки повинні мати однаковий користувальницький інтерфейс (UI) задля забезпечення вірності результатів дослідження;
- в якості застосунку можна обрати легкий варіант для створення задач;
- користувальницький інтерфейс повинен бути інтуїтивно зрозумілим;
- застосунок повинен надавати можливість для створення, редагування та видалення задач;
- застосунок повинен надавати можливість для перегляду створеної задачі;
- усі застосунки повинні мати однаковий шар віджетів для кожного з фреймворків та шар даних;

- усі застосунки повинні мати різний шар логіки для кожного з методів управління станом і фреймворків.

1.4.2. Фреймворки та мови програмування

Для розробки були обрані фреймворки Flutter та React Native. До них у пару стануть мови програмування Dart та JavaScript. Вони є ідеальними один для одного [10, 28].

Перевагами Dart є такі аспекти [44]:

- висока продуктивність, яка надає компіляцію в машинний код або попередню компіляцію, що дозволяє швидше завантажувати застосунок;
- екосистема з Flutter, зо дозволяє створювати багатоплатформні застосунки з єдиною кодовою базою;
- типізація, яка допомагає уникнути множини помилок на етапі компіляції;
- проста робота з асинхронністю;
- інтеграція з різними інструментами розробки та відладки.

Перевагами JavaScript є наступні аспекти [43]:

- повсюдне використання як в розробці користувальницького інтерфейсу, так і в серверній частині;
- обширна екосистема з різноманітними бібліотеками, фреймворками і пакетними менеджерами;
- проста робота з асинхронністю.

Dart часто обирають для розробки кросплатформних застосунків на Flutter завдяки своїй продуктивності та інтеграції. JavaScript є більш універсальним інструментом для веброзробки та інших областей, завдяки своїй посюдності та обширній екосистемі [24, 36].

1.4.3. Методи порівняння фреймворків і методів управління станом

В якості методів для порівняльного аналізу будуть обрані вже описані критерії (табл. 1.8). Такі методи дадуть розуміння, який фреймворк і метод управління станом є найкращими для кросплатформної мобільної розробки. Для визначення результатів того, чи іншого критерію будуть використовуватися:

- критерій порівняння швидкодії застосунку;
- критерій порівняння часу відгуку застосунку;
- критерій порівняння використання пам’яті;
- критерій порівняння використання енергії.

1.5. Висновки до розділу 1

За даним розділом було виконано дослідження розробки кросплатформних застосунків методами управління станом MobX і Redux у двох різних фреймворках Flutter та React Native. Був проведений порівняльний аналіз як методів управління станом, так і фреймворків. Кожен із має свої значні переваги та недоліки. Були визначені методи порівняльного аналізу мобільного застосунку, за якими й буде визначено найкращі результати.

РОЗДІЛ 2.
ПРОЄКТУВАННЯ ВНУТРІШНІХ ПРОЦЕСІВ РОЗРОБКИ МОБІЛЬНОГО ЗАСТОСУНКУ

2.1. Проєктування діаграми класів

Для проєктування діаграми класів необхідно визначитися з класами та їх наповненням (атрибутами, методами, функціями тощо). Також необхідно розуміти, які класи як пов’язані [5-6].

Є наступні зв’язки [21]:

- залежності, який визначає вплив одного класу на інший;
- асоціації, який визначає переміщення об’єктів одного класу до іншого;
- агрегації, який визначає асоціацію між цілим класом та його частиною;
- композиції (чітка агрегація), який визначає чітку залежність часу;
- узагальнення, який визначає підтипи класів до іншого – надтипу;
- реалізації (імплементації), який визначає відношення одного класу «клієнту» до іншого (постачальника).

Такі відношення класі дозволяють при розробці зрозуміти, які класи підпорядковуються яким, що в них міститься тощо.

2.1.1. Проєктування діаграми класів для фреймворку Flutter

Для розробки застосунку необхідні такі шари, як віджети, логіка та дані. Перший орієнтований на користувальницький інтерфейс, а два останніх – на серверну частину. Шари віджетів та даних є загальними для логіки методів управління станом MobX та Redux [4, 13, 21, 46].

Для шару даних була спроєктована діаграма класів, що наведена на рисунку 2.1.

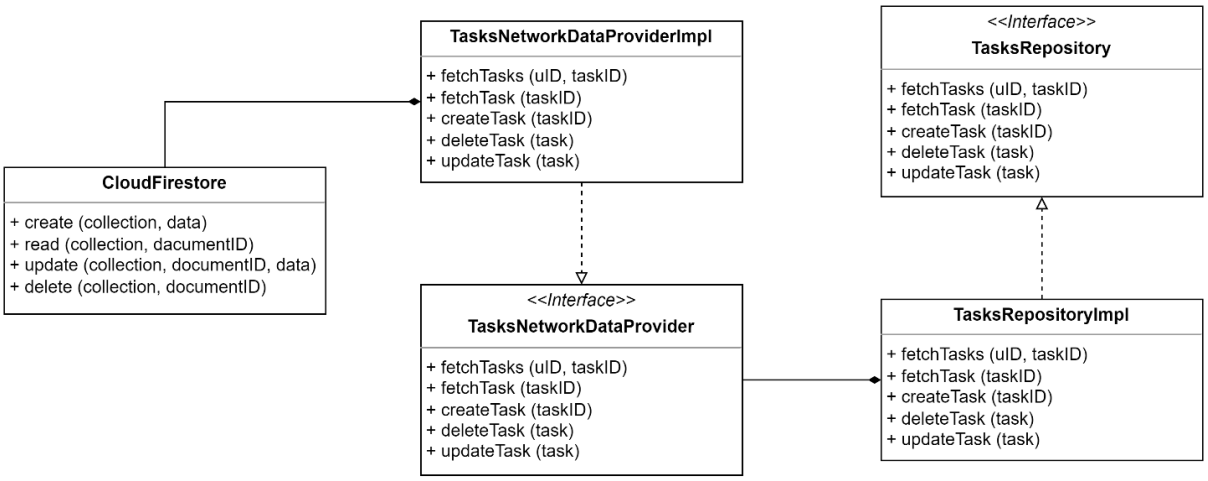


Рис. 2.1. Діаграма класів шару даних фреймворку Flutter

Шар даних (рис. 2.1) має хмарну базу даних, класи, що відповідають за інтерфейс та його працездатність. Клас бази даних пов’язаний з імплементований класом зв’язком чіткої агрегації. Імплментований клас задач пов’язаний з класом задач зв’язком імплементації, як і класи репозиторію з імплементованим репозиторієм.

Для методу управління станом MobX фреймворку Flutter була спроєктована діаграма класів, яка наведена на рисунку 2.2.

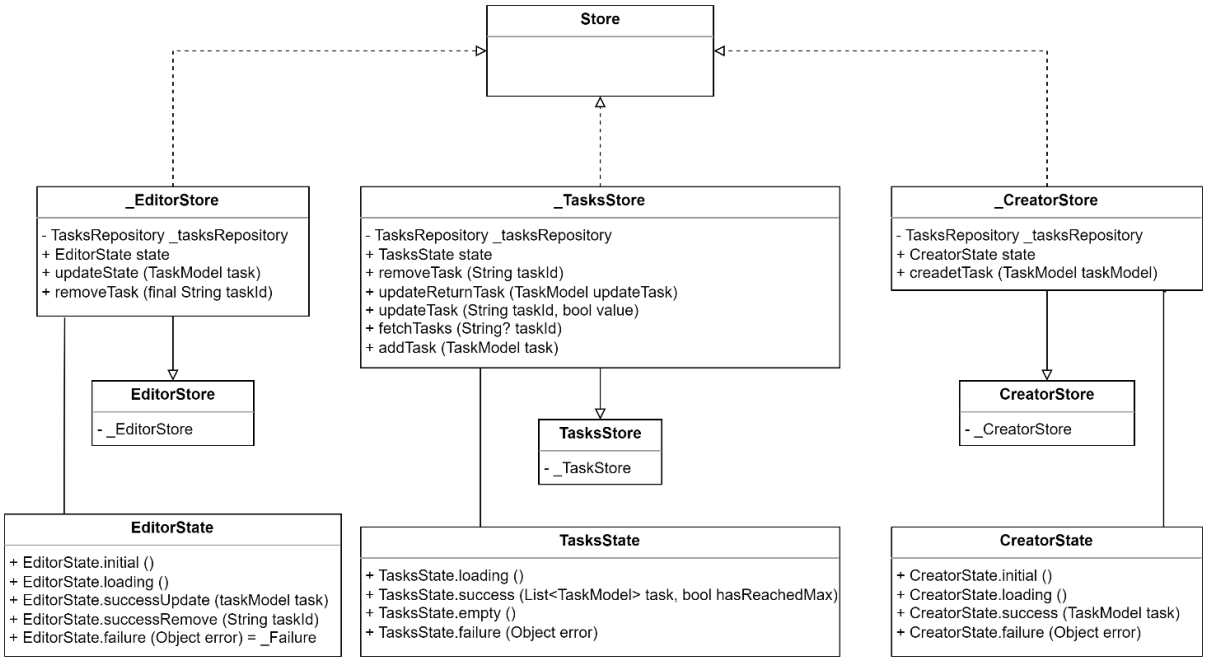


Рис. 2.2. Діаграма класів шару логіки MobX фреймворку Flutter

Як вже було продемонстровано, метод управління станом MobX керує станом завдяки класу Store. Він пов’язаний з класами зв’язками реалізації з сховищами класами створення, редагування та списком задач. А вони пов’язані з однойменними класами сховищами, які керують станами відповідними екранами мобільного застосунку.

Для методу управління станом Redux була спроєктована діаграма класів фреймворку Flutter, яка наведена на рисунку 2.3.

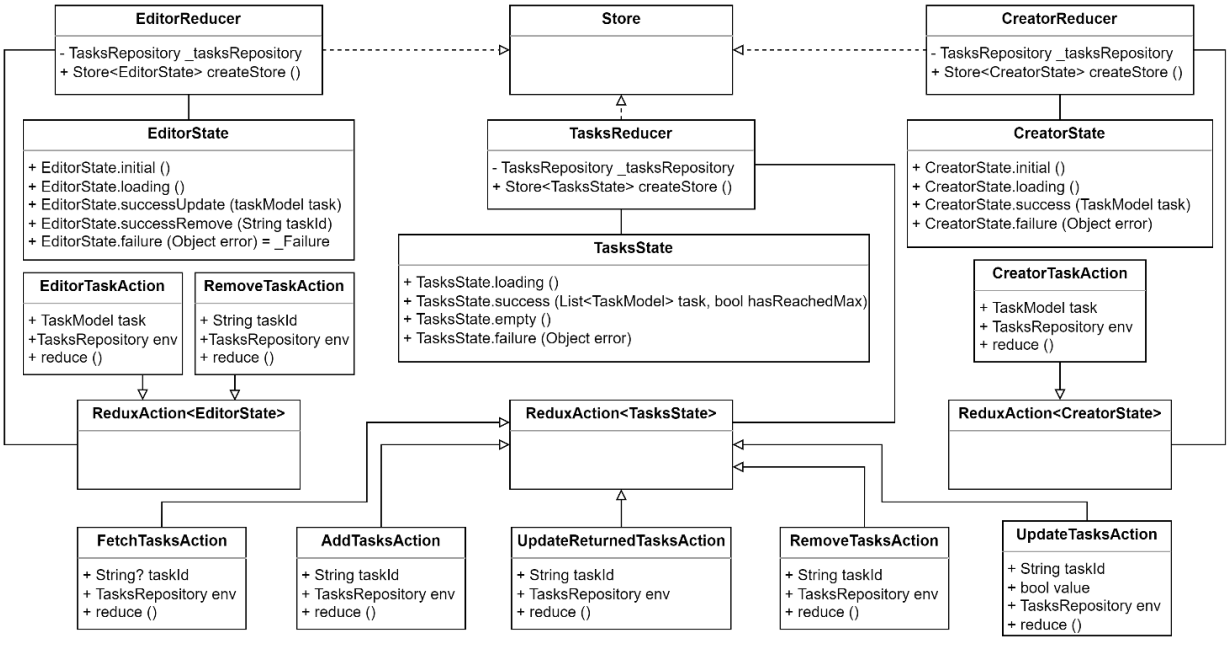


Рис. 2.3. Діаграма класів шару логіки Redux фреймворку Flutter

Як можна побачити, для методу управління станом необхідна більша кількість класів (рис. 2.3). У Redux також є сховище Store, в якому зберігаються стани застосунку. У даному випадку, як було зазначено вже, метод управління станом має так звані ред'юсери та дії, які є обов'язковими для створення та зв'язування з іншими класами.

Отож, уже можна побачити різницю між методами управління станом фреймворку Flutter на етапі проєктування.

2.1.2. Проєктування діаграми класів для фреймворку React Native

У фреймворку React Native діаграми класів будуть спроєктовані за аналогією до Flutter, так як структура проєктів не повинна відрізнятися для більш точної перевірки [8, 27].

Діаграма класів шару даних для фреймворку React Native наведена на рисунку 2.4.

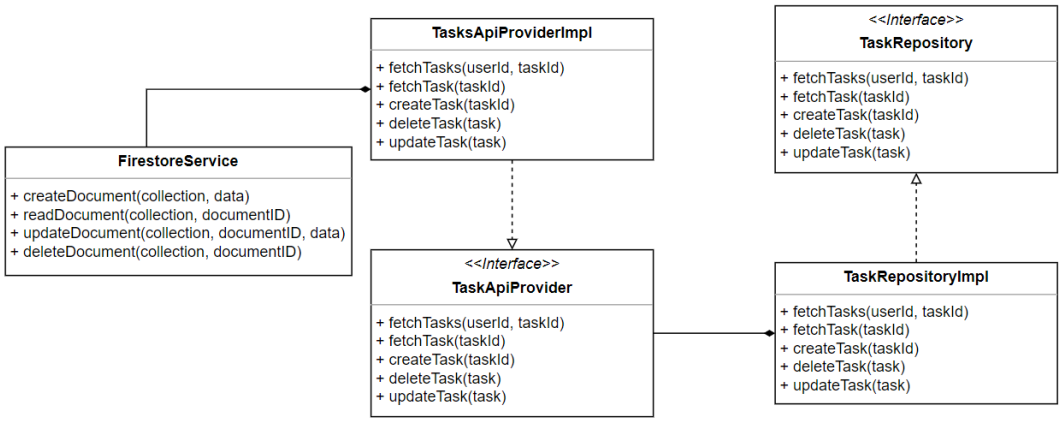


Рис. 2.4. Діаграма класів шару даних фреймворку React Native

На діаграмі (рис. 2.4) можна побачити аналогічну ситуацію до фреймворку Flutter, але вже з іншими класами. Діаграма класів методу управління станом MobX фреймворку React Native наведена на рисунку 2.5.

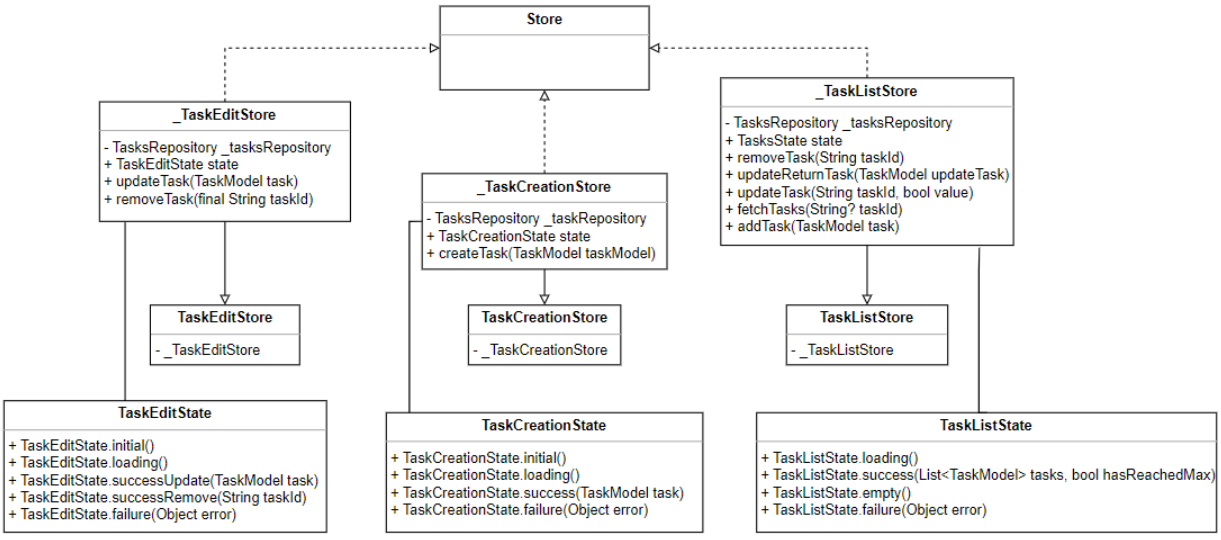


Рис. 2.5. Діаграма класів шару логіки MobX фреймворку React Native

За діаграмою (рис. 2.5) можна побачити класи, що необхідні для розробки логіки на MobX. Сховище даних є головним та зберігає в собі всі стани застосунку. Діаграма класів методу управління станом Redux фреймворку React Native наведена на рисунку 2.6.

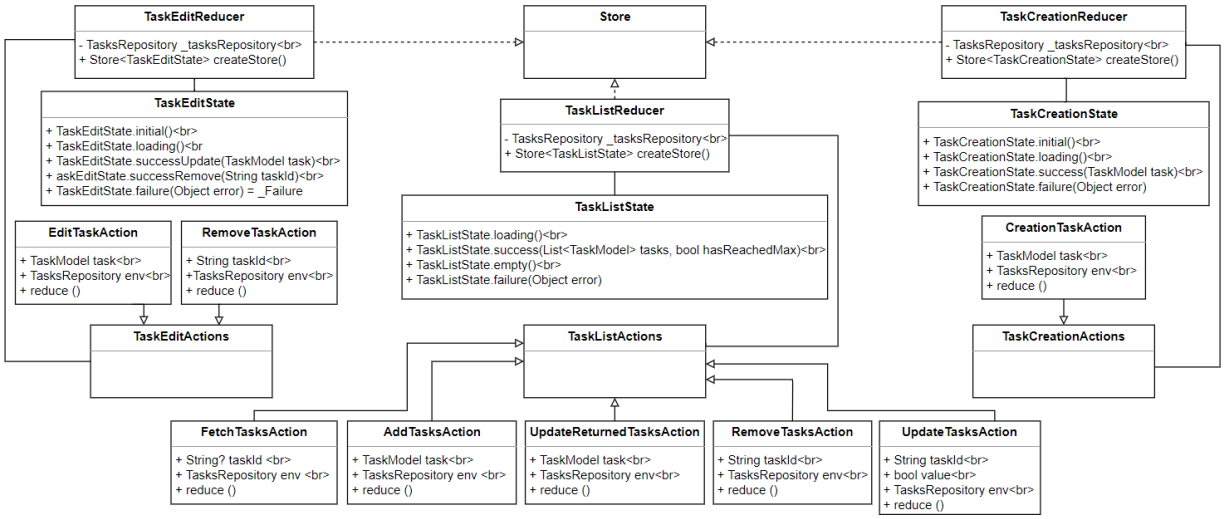


Рис. 2.6. Діаграма класів шару логіки Redux фреймворку React Native

За аналогією була спроєктована діаграма Redux. Також головним класом є сховище станів мобільного застосунку.

2.2. Проєктування діаграми станів

Діаграма станів повинна демонструвати логіку певних програмних функцій, спираючись на діаграму класів або користувальницький інтерфейс застосунку [18, 29].

Діаграма станів має такі елементи [18, 29]:

- коло початкового стану;
- стани застосунку або функціоналу;
- переходи у вигляді стрілок;
- розгалуження, які означають, що може бути декілька виходів із однієї

ситуації;

- коло кінцевого стану.

Для кожного з методів управління станом було спроєктовано по одній діаграмі станів для наступного функціоналу. Для методу управління станом MobX діаграми станів наведені на рисунку 2.7.

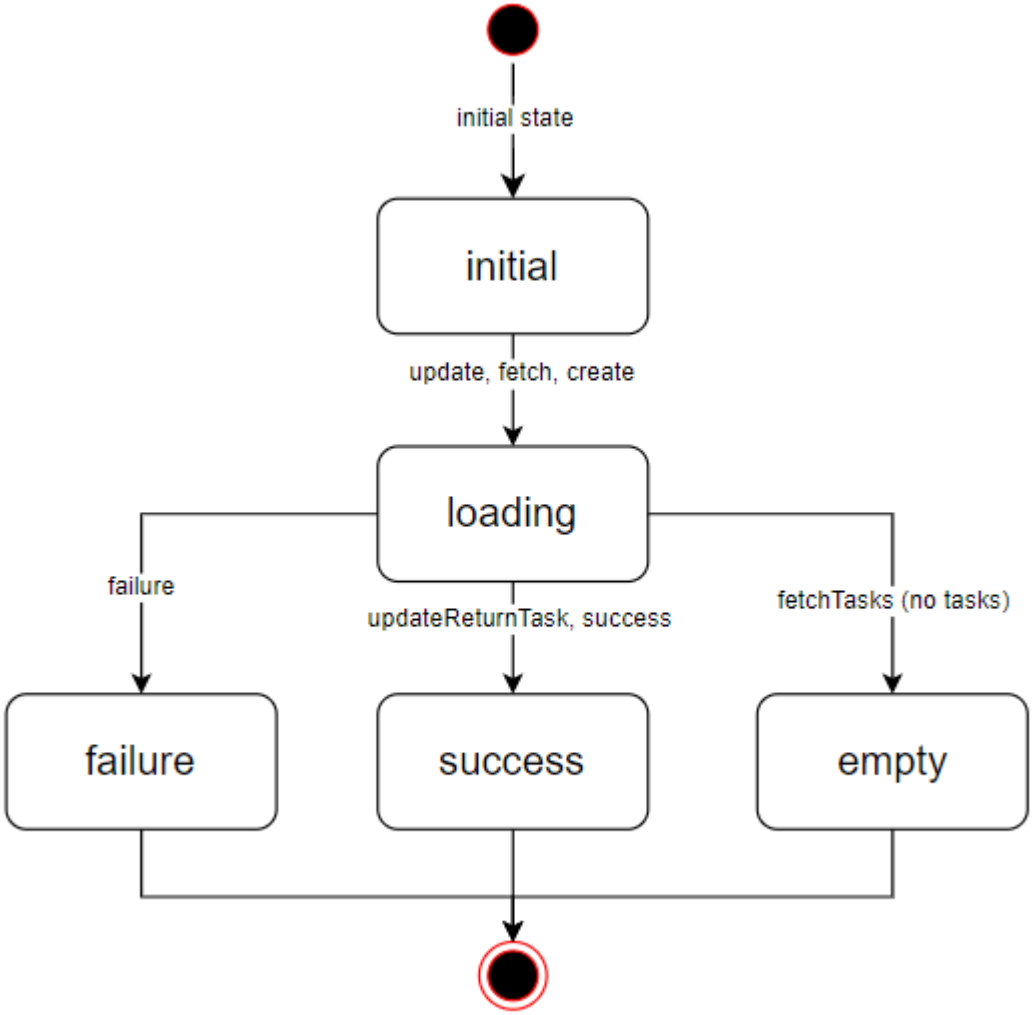


Рис. 2.7. Діаграма станів методу управління станом MobX

На діаграмі (рис. 2.7) наведені такі стани застосунку, як:

- стан ініціалізації;
- стан завантаження;
- стан помилки;

- стан успіху / оновлення задачі;
- стан пустоти / немає задач.

У даному випадку, у методі управління станом MobX, усі стани йдуть послідовно. Тобто, немає такої ситуації, коли б якийсь із станів можна було б пропустити.

Для методу управління станом Redux була спроєктована діаграма станів, яка наведена на рисунку 2.8.

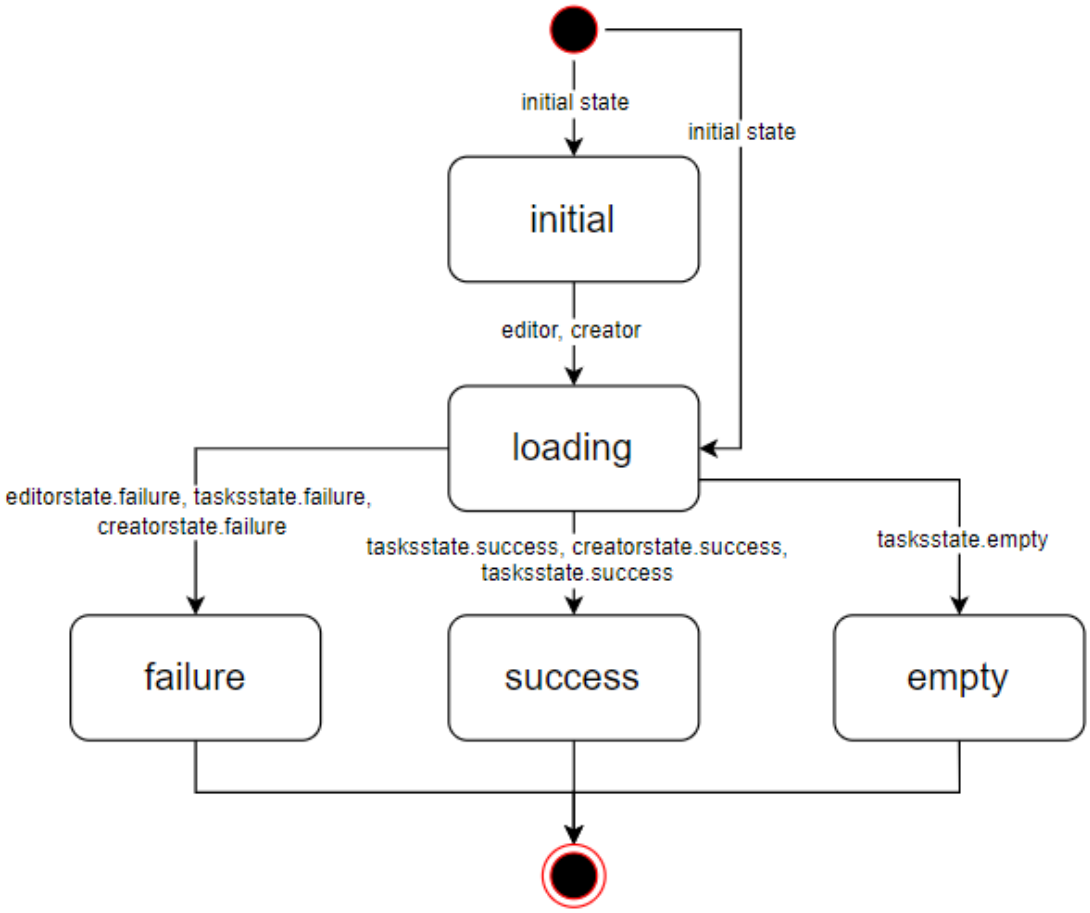


Рис. 2.8. Діаграма станів методу управління станом Redux

На діаграмі (рис. 2.8) можна побачити такі ж стани, що й на діаграмі станів MobX. Однак, у даному випадку не всі стани можуть іти послідовно. Як видно з наведених станів, стан ініціалізації може бути пропущеним на початку.

2.3. Проєктування діаграми діяльності

Діаграма діяльності відображає більш детальний процес працездатності станів застосунку. Вона описує дії, які виконують методи управління станом за допомогою наявних елементів [26, 34].

Діаграма має наступну конструкцію [26, 34]:

- початок процесу діяльності;
- дії / операції, що відбуваються;
- вузли керування, які призначені для координування потоків дій;
- рішення, які призначені для визначення правил розгалуження та подальших можливих варіантів подій;
- стрілки, що направляють потоки до необхідних елементів;
- кінець процесу діяльності.

Для кожного з методів управління станів були спроєктовані по три діаграми діяльності для наступних дій застосунку:

1) створення задачі;
2) редагування задачі;
3) доступ до списку задач або певної задачі.

Для методів управління станом MobX і Redux були спроєктовані діаграми діяльності, які наведені на рисунках 2.9-2.10.

На діаграмах (рис. 2.9-2.10) можна побачити діяльності, які мають такі загальні дії:

- ініціалізація дії;
- завантаження дії;
- вузол керування дією;
- розгалуження;
- кінець діяльності.

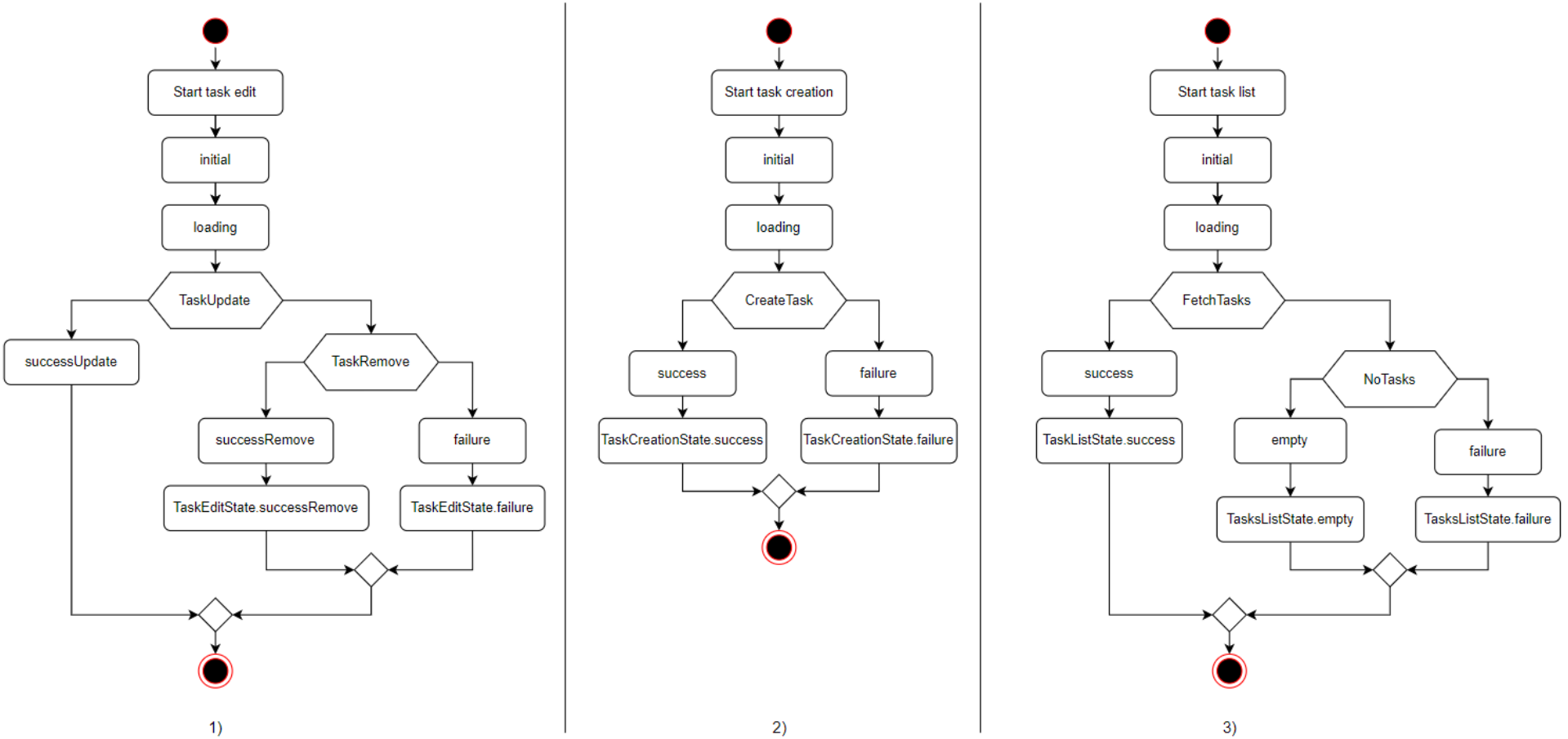


Рис. 2.9. Діаграми діяльності методу управління станом MobX

1) редагування задач; 2) створення задач; 3) доступ до списку задач або певної задачі

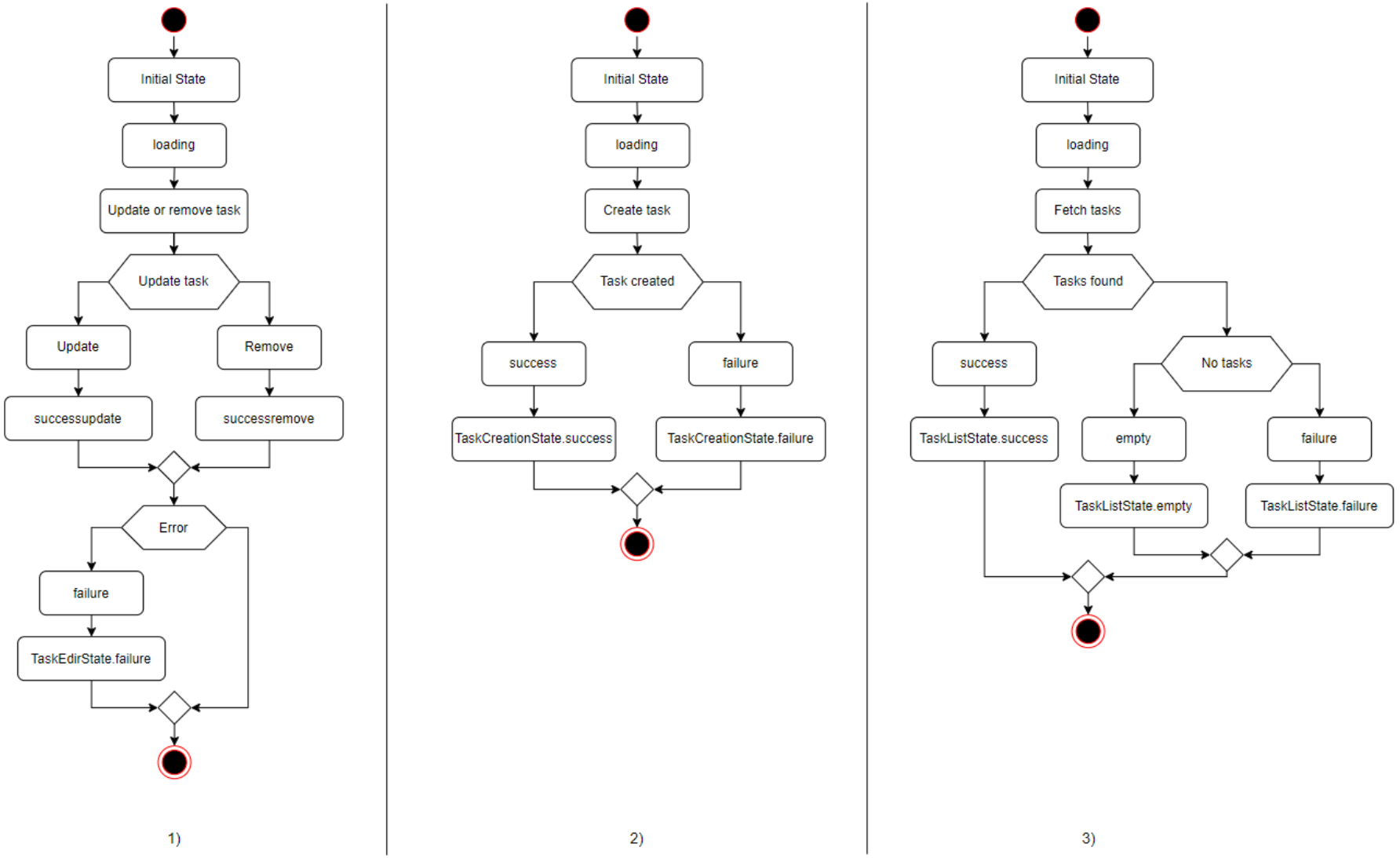


Рис. 2.10. Діаграми діяльності методу управління станом Redux

1) редагування задач; 2) створення задач; 3) доступ до списку задач або певної задачі

Діаграма діяльності необхідна для розуміння роботи процесів у програмній системі. Вона чітко та коротко надає відображення серверної частини проєкту процесів створення, редагування та доступу до задач.

2.4. Проєктування діаграми розгортання

Діаграма розгортання демонструє фізичне уявлення виконуваних процесів серверної частини застосунку. Основними елементами діаграм є вузли та компоненти, які пов'язані між собою з'єднаннями залежності. Також додатковими елементами є інтерфейси, об'єкти, коментарі тощо [30, 32].

Для фреймворків Flutter та React Native були спроєктовані узагальнені діаграми розгортання для обох методів управління станом. Вони наведені на рисунках 2.11-2.12.

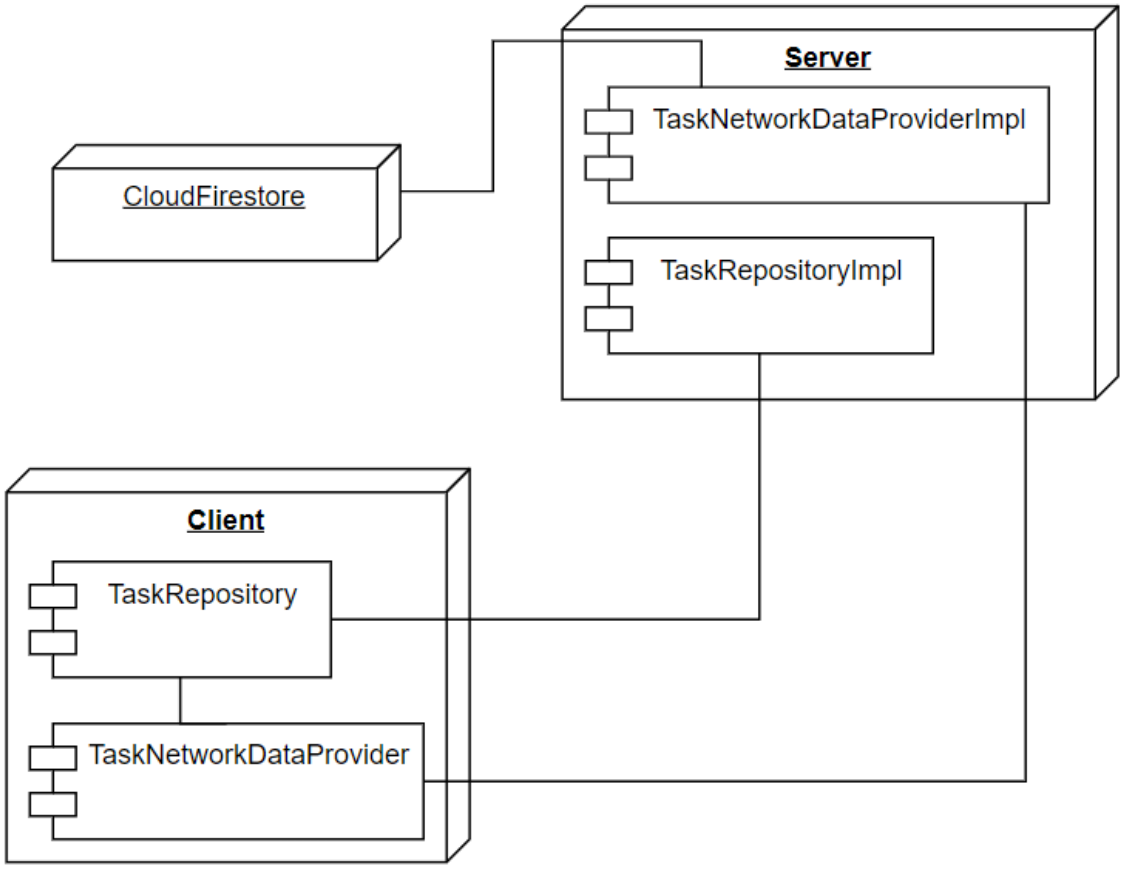


Рис. 2.11. Діаграма розгортання фреймворку Flutter

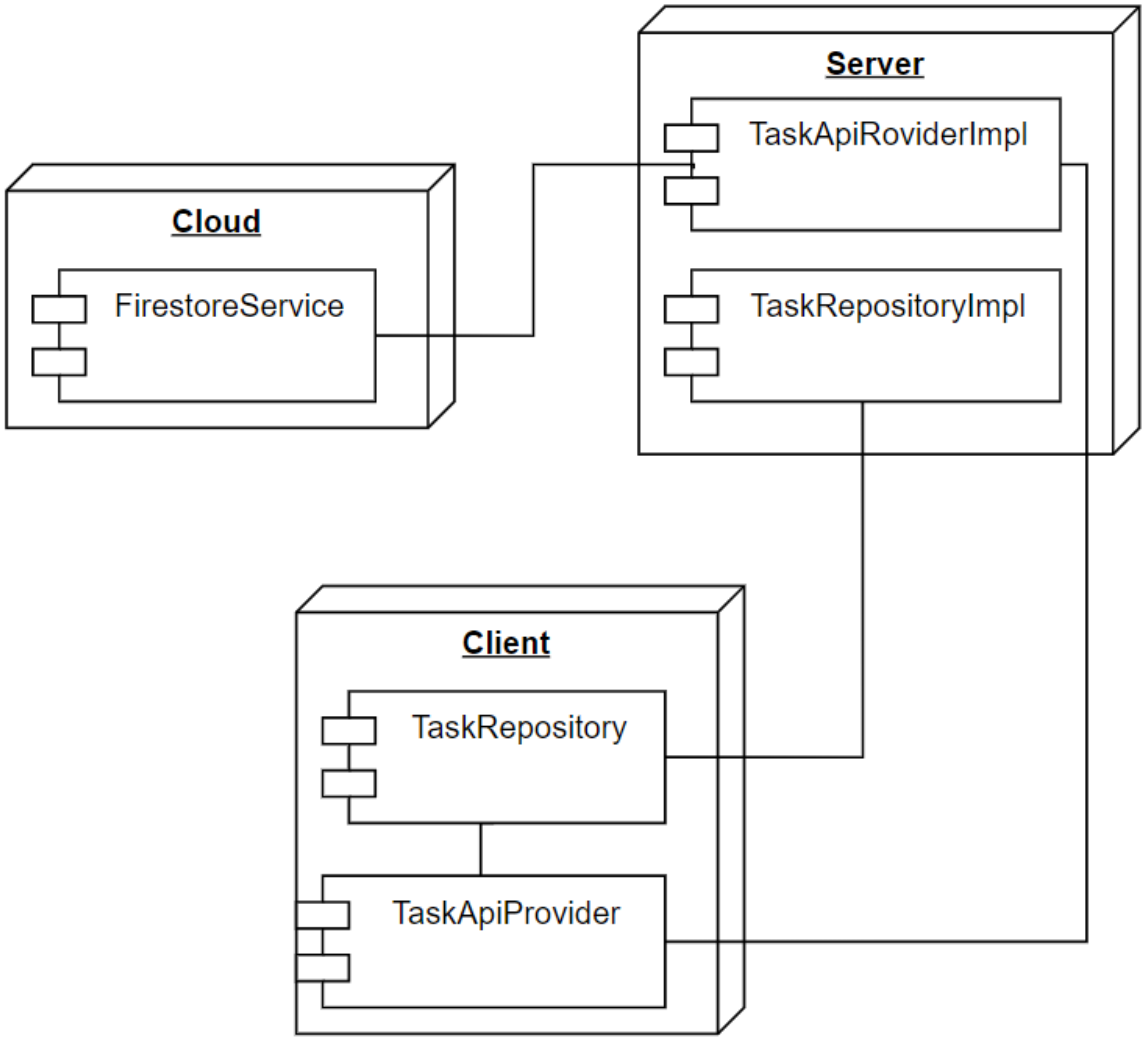


Рис. 2.12. Діаграма розгортання фреймворку React Native

Діаграми мають однаковий сенс для обох фреймворків:

- є сховище даних, яке містить у собі збережені дані із застосунку;
- є серверна частина, яка імплементує клієнтську та використовує дані зі сховища;
- є клієнтська частина, яка використовує дані із серверної.

Завдяки даній діаграмі буде зручно спиратися на зв’язки під час програмної реалізації серверної частини застосунку.

2.5. Висновки до розділу 2

Другий розділ був присвячений проєктуванню діаграм внутрішньої частини застосунку. Були спроєктовані діаграми класів для обох фреймворків та методів управління станом. Вони надали структурування класів, їх методів та атрибутів. Були спроєктовані діаграми станів, які показали, які саме стани можуть бути присутні на етапі розробки програмної системи. Були спроєктовані діаграми діяльності, які показали більш детально, як саме поводять себе окремо кожен стан застосунку. Були спроєктовані діаграми розгортання, які показують поведінку всіх вузлів системи. Уже на цьому етапі можна побачити різницю між фреймворками та методами управління стану.

РОЗДІЛ 3.
ПРОГРАМНА РОЗРОБКА МОБІЛЬНОГО ЗАСТОСУНКУ ТА ПОРІВНЯННЯ РЕЗУЛЬТАТІВ ПРАЦЕЗДАТНОСТІ МЕТОДІВ КЕРУВАННЯ СТАНОМ MOBX І REDUX ФРЕЙМВОРКІВ FLUTTER ТА REACT NATIVE

3.1. Програмна реалізація мобільного застосунку з використанням фреймворку Flutter

3.1.1. Програмна реалізація користувальницького інтерфейсу

Фреймворк Flutter у презентаційному шарі містить віджети та їх дерева. Приклад такого дерева наведений на рисунку 3.1. Завдяки ним відбувається промальовування користувальницького інтерфейсу та взаємодії користувача. Наприклад, StatelessWidget, StatefulWidget та подібні віджети відповідають за представлення та логіку відображення [9, 33, 50].

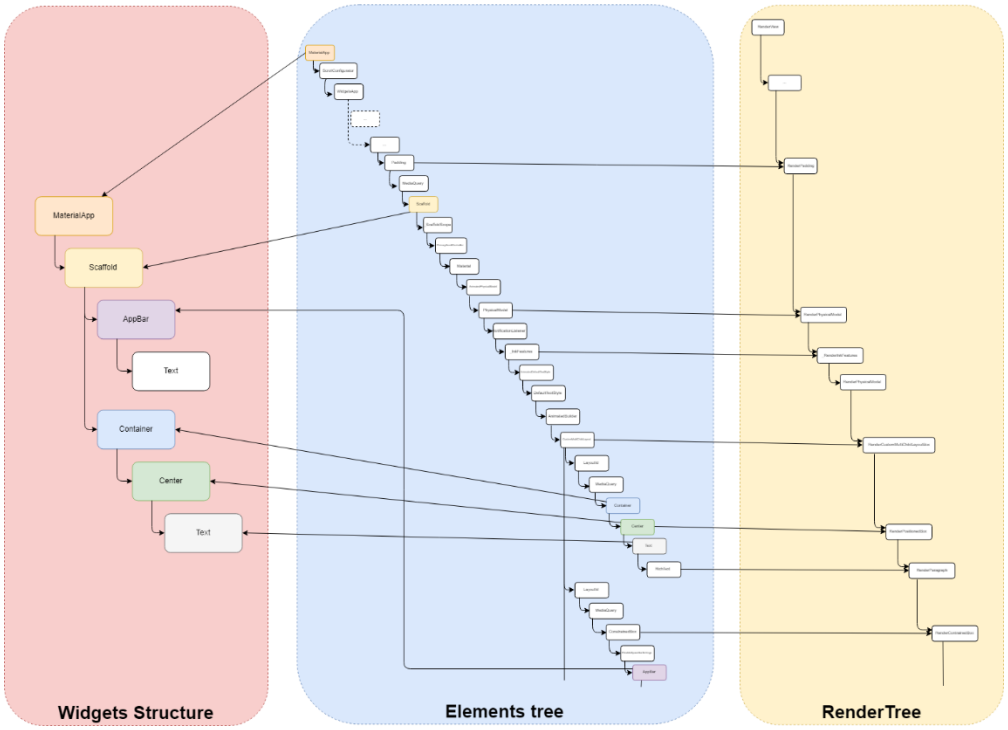


Рис. 3.1. Приклад дерева шару віджетів фреймворку Flutter [33]

Для графічного представлення екрану списку задач було створено за логікою створення контролерів для введення тексту в поля, а також побудову UI з використанням розроблених віджетів і відстеження та обробка стану. На рисунку 3.2. наведений фрагмент програмної реалізації екрану списку задач.

```
  @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. 3.2. Фрагмент програмної реалізації екрану зі списком задач фреймворку Flutter

За аналогією були реалізовані й два інших екрани застосунку: створення та редагування задач. Екран створення задач побудований на основі класу відображення задач. Основними моментами у програмній реалізації є:

- індикатор оновлення списку задач;
- віджети для використання складних та продуктивних списків з екраном прокрутки;
- використання свайпу для видалення задач;
- кнопка для додавання нових задач;
- оновлення стану, який залежить від поточного.

Фрагменти програмного коду даного екрану наведені на рисунках 3.3-3.5.

```
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. 3.3. Фрагмент програмної реалізації екрану створення задач фреймворку Flutter

```
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. 3.4. Фрагмент програмної реалізації індикатору оновлення, свайпу видалення та прокрутки фреймворку Flutter

```
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. 3.5. Фрагмент програмної реалізації кнопки додавання задач фреймворку Flutter

Екран редагування задач побудований на класі представлення редагування. Даний екран містить:

- форму змінення задачі;
- ініціалізацію стану віджету з інформацією про задачу;
- керування змінами стану;
- контролери полів для текстового вводу;
- перевірку змінених параметрів задачі;
- перевірку методу побудови користувальницького інтерфейсу тощо.

Фрагмент програмного коду наведено на рисунку 3.6.

```
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. 3.6. Фрагмент програмної реалізації екрану редагування задач фреймворку Flutter

Таким чином, відбувається програмна реалізація шару UI. У результаті отримуємо відповідні екрани мобільного застосунку, які наведені в додатку А.

3.1.2. Програмна реалізація логіки з використанням методу керування станом MobX

Для реалізації логіки екранів мобільного застосунку необхідно виконати імпорт бібліотеки MobX, щоб співпрацювати з даним методом управління станом. Як було вже зазначено, даний метод управління станом містить @action методи, що визначають дії, які змінюють стан. Для роботи із задачами були задіяні методи, що наведені в таблиці 3.1.

Таблиця 3.1

Методи для роботи з задачами MobX фреймворку Flutter

Метод	Опис
removeTask	Метод видалення задачі за її унікальним номером та оновлення її стану
updateReturnTask	Метод оновлення задачі на основі нових даних та збереження оновлень у репозиторій
updateTask	Метод оновлення стану виконання задачі за прапорцем та збереження її в репозиторій
fetchTasks	Метод завантаження задачі з репозиторію для користувача та оновлення стану
addTask	Метод додання нової задачі до списку, сортування їх за датою та оновлення стану

Джерело: розробка автора

Фрагмент програмного коду наведено на рисунку 3.7.

```
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 {
```

Рис. 3.7. Фрагмент програмної реалізації логіки екрану списку задач методу управління станом MobX фреймворку Flutter

Для екрану створення задач існує головний метод, який виконує наступні кроки:

- змінює стан завантаження при початковому створенні задачі;

- створює задачу за допомогою репозиторію, викликаючи метод і встановлюючи стан успішного створення (якщо він такий);

- якщо відбулася помилка, стан змінюється на помилковий та відновлюється до початкового стану ініціалізації.

Фрагмент програмного коду створення задачі наведений на рисунку 3.8.

```
@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. 3.8. Фрагмент програмної реалізації логіки екрану створення задач методу управління станом MobX фреймворку Flutter

Для екрану редагування задач були задіяні два методи:

1) оновлення задачі, який виконує наступне:

1) установлює стан завантаження перед початком операції;

2) викликає метод з репозиторію для оновлення задачі;

3) якщо оновлення проходить успішно, стан змінюється на відповідний, вміщуючи в собі оновлену задачу;

4) у випадку помилки, стан змінюється на відповідний і відновлюється до початкової ініціалізації;

2) видалення задачі, який виконує наступне:

1) установлює стан завантаження перед початком видалення;

2) викликає метод з репозиторію для видалення задачі за унікальним номером;

3) якщо видалення успішне, стан змінюється на відповідний, включаючи унікальний номер видаленої задачі;

4) у випадку помилки, стан змінюється на відповідний і відновлюється до ініціалізації.

Фрагмент програмної реалізації наведений на рисунку 3.9.

```
  @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. 3.9. Фрагмент програмної реалізації логіки екрану редагування задач методу управління станом MobX фреймворку Flutter

Таким чином відбувається створення екранів мобільного застосунку фреймворку Flutter з використанням методу управління стану MobX.

3.1.3. Програмна реалізація логіки з використанням методу керування станом Redux

Як вже було описано головним у Redux є ред'юсер. Для створення екрану списку задач є відповідний метод, який виконує:

1) створення та повернення сховища для керування станом задач;

2) ініціалізація стану задає початковий стан як TasksState.loading(), вказуючи на те, що дані задач завантажуються;

3) залежності передають репозиторії задач в якості середовища для сховища, щоб мати доступ до операцій з задачами.

Фрагмент коду наведений на рисунку 3.10.

```
class TasksReducer {
  final TasksRepository _tasksRepository;

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

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

Рис. 3.10. Фрагмент програмної реалізації логіки екрану списку задач методу управління станом Redux фреймворку Flutter

Дії для роботи із задачами описані в таблиці 3.1. Вони аналогічні, як і в методі управління станом MobX.

Створення задач описується реалізацією класу для створення та керування станом задач, а відповідне поле містить посилання на репозиторій задач, який використовується для взаємодії з даними задачами. Також використовується метод createStore, який:

- створює та повертає сховище для керування станом;

- стан ініціалізації задає початковий стан, використовуючи константу CreatorState.initial();

- залежності передають репозиторій задач в якості оточення для сховища, щоб мати доступ до операцій із задачами.

Фрагмент реалізації наведений на рисунку 3.11.

```
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. 3.11. Фрагмент програмної реалізації логіки екрану створення задач методу управління станом Redux фреймворку Flutter

Для створення екрану редагування задач клас reducer відповідає за збереження та управління станом редагованих задач. Метод виконує наступні дії:

- створює та повертає сховище для керування станом редактору задач;

- задає початковий стан EditorState.initial(), що вказує на початковий стан редактору задач;

- залежності передають репозиторій задач в якості оточення для сховища, щоб забезпечити доступ до операцій із задачами.

Фрагмент наведено на рисунку 3.12.

```
class EditorReducer {
  final TasksRepository _tasksRepository;

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

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

Рис. 3.12. Фрагмент програмної реалізації логіки екрану редагування задач методу управління станом Redux фреймворку Flutter

Таким чином відбувається розробка логіки мобільного застосунку на Redux фреймворку Flutter.

3.1.4. Програмна реалізація зберігання даних

Для підключення та налаштування параметрів бази даних (БД) необхідно імпортувати потрібні бібліотеки та пакети. Для взаємодії з мобільним застосунком допоможе метод currentPlatform, який повертає параметри БД у залежності від платформи. Якщо застосунок запущено на тій платформі, яка не підтримується, метод надає виняток UnsupportedError. Для тих платформ, що підтримуються повертаються відповідні параметри конфігурації.

Клас CloudFirestore містить конструктор, який приймає екземпляр FirebaseFirestore та зберігає його в приватне поле, що дозволяє звертатися до БД в різних методах CRUD класу. Дані методи наведені в таблиці 3.2.

Таблиця 3.2

Методи класу CloudFirestore фреймворку Flutter

Метод	Опис
create	Дозволяє створювати новий документ в заданій колекції. Приймає ім’я колекції та дані у вигляді карти (словника). Використовує try-catch для обробки можливих помилок
read	Читає документ за його ідентифікатору в указаній колекції та повертає об’єкт DocumentSnapshot, що містить дані документу
update	Оновлює існуючий документ у колекції, використовуючи надані дані
delete	Видаляє документ за його ідентифікатором із вказаної колекції

Джерело: розробка автора

Фрагмент коду наведений на рисунку 3.13.

```
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. 3.13. Фрагмент програмної реалізації приєднання БД до мобільного застосунку фреймворку Flutter

3.2. Програмна реалізація мобільного застосунку з використанням фреймворку React Native

Реалізація на React Native виконується схожим чином, але має свої відмінності. Повна програмна реалізація наведена в додатку Б.

3.2.1. Програмна реалізація користувальницького інтерфейсу

Для створення екрану списку задач клас TaskListStore керує станом задач та виконує такі задачі:

- властивість state приймає початковий стан: loading, success або failure;
- викликає makeObservable для налаштування реактивності класу;
- повертає список задач, якщо стан успішний, в іншому випадку – пустий масив;
- асинхронні методи для виконання різних операцій над задачами оновлюють стан у залежності від результату операцій.

Для створення екрану створення задачі існує клас CreationStore, який виконує такі задачі:

- властивість стану зберігає поточний стан (початковий, успішний або невдалий) та інформацію про задачу або помилку;
- конструктор викликає makeAutoObservable для автоматичного спостереження за властивостями та методами класу;
- асинхронний метод створення задач симулює виклик API для створення задач, а після завершення оновлює стан у залежності від результату.

Для створення екрану редагування задач розроблений клас EditStore, який виконує такі задачі:

- переводить початковий стан в ініціалізацію;

- робить всі властивості та методи класу автоматично спостережуваними;

- методи редагування та видалення є асинхроними для відповідних своїх дій.

Також розроблений компонент EditView, який відображає форму для редагування. Має функції для оновлення та видалення задачі, а також перевіряє чи відбулися зміни в задачі. Використовує хуки для:

- зберігання поточних значень полів задачі (назва, опису, дати завершення, категорії, статусу, пріоритету);

- виконання побічних ефектів, оновлюючи компонент при зміні стану в сховищі.

На рисунку 3.14 відображений фрагмент коду створення UI з використанням фреймворку React Native.

```
function openPriorityDialog() {
  // Open a dialog to select priority
  Alert.alert('Select a Priority', '', [
    { text: 'High', onPress: () => setPriority('High') },
    { text: 'Medium', onPress: () => setPriority('Medium') },
    { text: 'Low', onPress: () => setPriority('Low') },
    { text: 'Cancel', style: 'cancel' },
  ]);
}

function openCategoryDialog() {
  // Open a dialog to select category
  Alert.alert('Select a Category', '', [
    { text: 'Work', onPress: () => setCategory('Work') },
    { text: 'Home', onPress: () => setCategory('Home') },
    { text: 'Other', onPress: () => setCategory('Other') },
    { text: 'Cancel', style: 'cancel' },
  ]);
```

Рис. 3.14. Фрагмент програмної реалізації UI фреймворку React Native

3.2.2. Програмна реалізація логіки з використанням методу керування станом MobX

Логіка екрану створення задач полягає в наступному:

- імпорт необхідних бібліотек і модулів;
- визначення станів;
- створення класу, його конструктору та методів;
- виконання експорту.

Клас TaskListStore відповідає за:

1) ініціалізацію стану завантаження LoadingState;
2) створення спостережуваного сховища;
3) використання таких методів:

1) removeTask(taskId) для видалення задачі за ідентифікатором; якщо задача видалена успішно, стан повинен оновитися; якщо задач не залишилося – установлюється порожній стан;

2) updateReturnTask(updateTask) для оновлення існуючої задачі; якщо оновлення успішне, змінює стан на оновлений список задач;

3) updateTask(taskId, value) для оновлення статусу виконання задачі за ідентифікатором;

4) fetchTasks(taskId) для завантаження задачі поточного користувача за ідентифікатором; оновлює стан у залежності від успішності отриманих даних;

5) addTask(task) для додавання нової задачі в список та оновлення стану.

Інші два класи в двох екранах мають аналогічний алгоритм роботи, але мають інші методи для їх розробки.

Отож, для екрану створення задач використовується клас CreationStore, який використовує метод createdTask(taskModel), який у свою чергу необхідний для:

- установлення стану в завантаження, що сигналізує про початок процесу створення задачі;

- викликає функцію створення задачі з її моделлю та чекає на результат;

- якщо задача успішно створена, з використанням runInAction оновлює задачу та встановлює стан в успішний;

- у випадку помилки встановлює її та змінює стан на відповідний;

- у блоці finally скидає стан назад в ініціалізацію, якщо це необхідно.

Створений клас відстежує стан процесу (початковий, завантаження, успіх та помилку) та зберігає дані про створену задачу або помилку, якщо вона виникла.

Для реалізації логіки екрану редагування задач був створений клас EditStore, у якому:

- змінна стану, яка відстежує поточний стан сховища та може приймати значення ініціалізації, завантаження, успішного оновлення, успішного видалення, помилки;

- змінна задачі для її збереження, яка буда успішно оновлена, а в початковому стані є нулем;

- змінна ідентифікатору для його збереження, яка була успішно видалена та початково є нулем;

- змінна похибки для її збереження, якщо вона виникла в процесі оновлення або видалення задачі, на початку є нулем.

У класі є конструктор, який приймає репозиторій задач в якості аргументу та зберігає його для подальшого використовування. Конструктор використовує makeAutoObservable для створення моніторингу сховища, що

дозволяє MobX автоматично відстежувати зміни стану та оновлювати інтерфейс користувача.

У класі присутні методи управління станом і виконання дій. Вони наведені в таблиці 3.3.

Таблиця 3.3

Методи управління станом та виконання дій фреймворку React Native методу управління станом MobX

Метод	Опис
1	2
Методи виконання дій	
updateTask(task)	Асинхронний метод для оновлення задачі. Спочатку встановлюється стан у завантаження, викликається метод оновлення задачі з репозиторію. Якщо операція успішна, стан змінюється на успішне оновлення та зберігається оновлена задача. В інакшому випадку стан змінюється на похибку та переходить до початкового
removeTask(taskId)	Асинхронний метод для видалення задачі за її ідентифікатором. Спочатку встановлює стан в завантаження, потім викликається метод видалення задачі з репозиторію. Якщо операція успішна, стан змінюється на успішне видалення та зберігається ідентифікатор видаленої задачі. У випадку помилки стан змінюється на відповідний і переходить у початковий
Методи управління станом	
setInitial()	Установлює стан в ініціалізацію та скидає всі змінні в нуль
setLoading()	Установлює стан в завантаження та скидає змінні
setSuccessUpdate (task)	Установлює стан в успішне оновлення та зберігає оновлену задачу, а інші змінні скидає
setSuccessRemove (taskId)	Установлює стан в успішне видалення та зберігає оновлену задачу, а інші змінні скидає

setFailure(error)	Установлює стан в похибку та зберігає її, інші змінні скидає

Джерело: розробка автора

Фрагмент коду логіки екрану редагування задач наведений на рисунку 3.15.

```
// Update task action
async updateTask(task) {
    this.setLoading();
    try {
        await this._tasksRepository.updateTask(task);
        runInAction(() => {
            this.setSuccessUpdate(task);
        });
    } catch (error) {
        runInAction(() => {
            this.setFailure(error);
            this.setInitial();
        });
        throw error; // Re-throw to handle it outside if needed
    }
}

// Remove task action
async removeTask(taskId) {
    this.setLoading();
    try {
        await this._tasksRepository.deleteTask(taskId);
        runInAction(() => {
            this.setSuccessRemove(taskId);
        });
```

Рис. 3.15. Фрагмент програмної реалізації логіки екрану редагування задач методу управління станом MobX фреймворку React Native

Таким чином відбувається програмна реалізація логіки екранів мобільного застосунку методу управління станом MobX фреймворку React Native.

3.2.3. Програмна реалізація логіки з використанням методу керування станом Redux

Для реалізації логіки екрану списку задач був створений ред’юсер taskListReducer для керування станом задач у застосунку. Він відповідає за обробку різних типів дій, пов’язаних із задачами та оновлює стан на основі цих дій. Функція приймає поточний стан та дію в якості аргументу та повертає новий стан у залежності від типу дії. У даному випадку виконується обробка різних типів дій:

- коли починається запит на отримання задачі, стан оновлюється с встановлення прапору завантаження в true;

- коли задачі успішно отримані, стан оновлюється з новими задачами, а прапор завантаження скидається в false; також оновлюється прапор hasReachedMax;

- у випадку невдачі при отриманні задач стан оновлюється з указанням помилки та скидання прапору завантаження в false;

- дії (видалення, додавання, оновлення) оброблюються аналогічно, але вони не змінюють стан напряму; вони можуть слугувати для виконання якихось асинхронних операцій;

- при успішному видаленні, доданні, оновленні задачі чи поверненні оновленої задачі, стан оновлюється новими даними задач;

- у випадку невдачі при видаленні, доданні, оновленні задачі або поверненні оновленої задачі стан оновлюється з вказанням похибки.

Для створення логіки екрану створення задачі реалізований ред’юсер creationReducer, який керує станом, пов’язаним із створенням задачі в застосунку. Функція приймає поточний стан та дії в якості аргументів та повертає новий стан на основі типу дій. Всередині ред’юсеру виконується конструкція перемикання для обробки наступних дій:

1) коли починається процес створення задачі, стан оновлюється:

1) loading встановлюється в true, що вказує на запуск процесу створення задачі;

2) error скидається в нульовий, що очищує можливі попередні помилки;

2) коли задача успішно створена, стан оновлюється:

1) loading встановлюється в false, що вказує на завершення процесу створення;

2) task оновлюється з даними, переданими в action.payload, що відображає успішне створення задачі;

3) у випадку невдачі при створенні задачі стан оновлюється:

1) loading встановлюється в false, що вказує на завершення процесу створення задачі;

2) error оновлюється з даними, переданими в action.error, що відображає помилку, що виникла.

Для розробки екрану редагування задач реалізований ред'юсер editorReducer, який керує станом, пов'язаним з видаленням та оновленням задач. Його функція приймає поточний стан і дії в якості аргументів та повертає новий стан у залежності від типу дії. Всередині ред'юсера використовується перемикач для обробки різних типів дій:

- коли задача успішно видалена, стан оновлюється, щоб відобразити це; стан повинен бути змінений, але конкретні зміни не вказані;

- у випадку невдачі при видаленні задачі стан оновлюється, щоб відобразити дану помилку;

- коли задача успішно оновлена, стан оновлюється, щоб відобразити успішне оновлення;

- у випадку невдачі при оновленні задачі стан оновлюється, щоб відобразити дану помилку.

Якщо тип дії не відповідає ні одному з оброблених випадків, ред'юсер повертає поточний стан без змін. Фрагмент коду наведений на рисунку 3.16.

Таким чином, виконується розробка екранів мобільного застосунку методу управління станом Redux фреймворку React Native.

```
const editorReducer = (state = initialState, action) => {
  switch (action.type) {
    case REMOVE_TASK_SUCCESS:
      // Handle success case
      return {
        ...state,
        // Update state to reflect successful task removal
      };
    case REMOVE_TASK_FAILURE:
      // Handle failure case
      return {
        ...state,
        // Update state to reflect failed task removal
      };
    case UPDATE_TASK_SUCCESS:
      // Handle success case
      return {
        ...state,
        // Update state to reflect successful task update
      };
    case UPDATE_TASK_FAILURE:
```

Рис. 3.16. Фрагмент програмної реалізації екрану редагування задач методу управління станом Redux фреймворку React Native

3.2.4. Програмна реалізація зберігання даних

Для підключення та ініціалізації бази даних була розроблена функція initializeFirebase, яка використовує відповідну конфігурацію в залежності від платформи. Вона експортується за замовчуванням для використання в інших частинах застосунку. Клас FirestoreService інкапсулює заємодію з БД та надає методи для створення, читання, оновлення та видалення документів у колекціях БД. Методами класу є:

1) додання нового документу у вказану колекцію;
2) читання даних з документу з указаним ідентифікатором у колекції;
3) оновлення даних в документі з указаним ідентифікатором у колекції;
4) видалення документу з указаним ідентифікатором у колекції.

Фрагмент коду наведений на рисунку 3.17.

```
async create(collection, data) {
  try {
    await this._firestore.collection(collection).add(data);
  } catch (error) {
    console.error('Error creating document:', error);
  }
}

async read(collection, documentId) {
  try {
    const doc = await this._firestore.collection(collection).doc(documentId).get();
    if (doc.exists) {
      return doc.data();
    } else {
      console.error('No such document!');
    }
  } catch (error) {
    console.error('Error reading document:', error);
  }
}
```

Рис. 3.17. Фрагмент програмної реалізації приєднання БД до мобільного застосунку фреймворку React Native

3.3. Порівняння методів керування станом MobX і Redux фреймворків Flutter та React Native

3.3.1. Порівняння результатів швидкодії

Для виміру швидкодії (початок та кінець роботи застосунку) для обох фреймворків були написані функції їх обчислення. Для Flutter обчислення швидкодії наведено на рисунку 3.18, а для React Native – на рисунку 3.19.

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

Рис. 3.18. Програмна реалізація функції обчислення швидкодії фреймворку Flutter

```
import React from 'react';
import { Text, View } from 'react-native';

const MeasureExecutionTime = () => {
  const startTime = new Date();

  // Виконайте операцію тут
  const endTime = new Date();
  const executionTime = endTime - startTime; // час виконання в мілісекундах

  return (
    <View>
      <Text>Час виконання операції: {executionTime} мс</Text>
    </View>
  );
};

export default MeasureExecutionTime;
```

Рис. 3.19. Програмна реалізація функції обчислення швидкодії фреймворку React Native

Результат виміру наведені в таблиці 3.4.

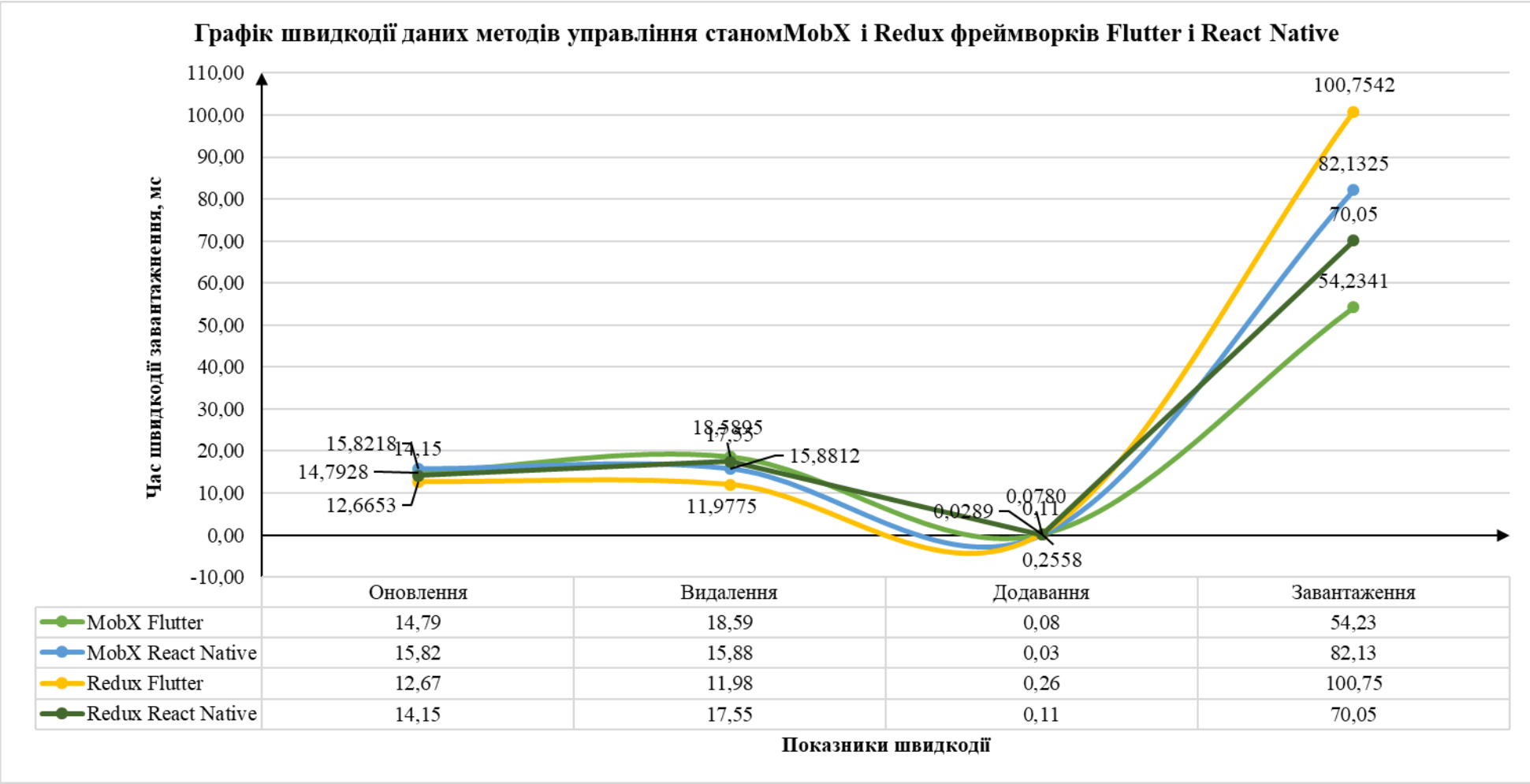
Таблиця 3.4

Результати виміру швидкодії мобільного застосунку фреймворків Flutter та React Native

Найменування операції	Результати виміру за методами управління станом, мс			
	MobX Flutter	MobX React Native	Redux Flutter	Redux React Native
Оновлення	14,79	15,82	12,67	14,15
Видалення	18,59	15,88	11,98	17,55
Додавання	0,08	0,03	0,26	0,11
Завантаження	54,23	82,13	100,75	70,05

Джерело: розробка автора

За таблицею результатів (табл. 3.4) був побудований графік, який наведено на рисунку 3.20.



	Оновлення	Видалення	Додавання	Завантаження
MobX Flutter	14,79	18,59	0,08	54,23
MobX React Native	15,82	15,88	0,03	82,13
Redux Flutter	12,67	11,98	0,26	100,75
Redux React Native	14,15	17,55	0,11	70,05

Рис. 3.20. Результат швидкодії мобільного застосунку фреймворків Flutter та React Native

3.3.2. Порівняння результатів часу відгуку

Для визначення часу відгуку в двох фреймворках був розроблений програмний код, який визначає час між початком натиснення та відгуком від мобільного застосунку. Обидві реалізації наведені на рисунках 3.21-3.22.

```
  await Future.delayed(Duration(seconds: 2)); // Приклад асинхронної операц

  setState(() {
    responseTime = DateTime.now().difference(startTime);
  });
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: Text('Перевірка часу відгуку')),
    body: Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          ElevatedButton(
            onPressed: _checkResponseTime,
            child: Text('Перевірити час відгуку'),
          ),
          SizedBox(height: 20),
```

Рис. 3.21. Фрагмент програмної реалізації обчислення часу відгуку фреймворку Flutter

```
const checkResponseTime = async () => {
  const startTime = new Date().getTime();

  // Ваша операція, наприклад, HTTP-запит
  await new Promise(resolve => setTimeout(resolve, 2000)); // Приклад асинхронної опе

  const endTime = new Date().getTime();
  const executionTime = endTime - startTime;
  setResponseTime(executionTime);
};

return (
  <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
    <Button title="Перевірити час відгуку" onPress={checkResponseTime} />
    {responseTime !== null && (
```

Рис. 3.22. Фрагмент програмної реалізації обчислення часу відгуку фреймворку React Native

За результат візьмемо час відгуку UI. Результати наведені в таблиці 3.5.

Таблиця 3.5

Результати часу відгуку мобільного застосунку фреймворків Flutter та React Native

Метод управління станом	Результати відгуку за фреймворками, мс	
	Flutter	React Native
MobX	42,15	63,00
Redux	78,05	44,50

Джерело: розробка автора

Наглядовий результат наведений на рисунку 3.23.

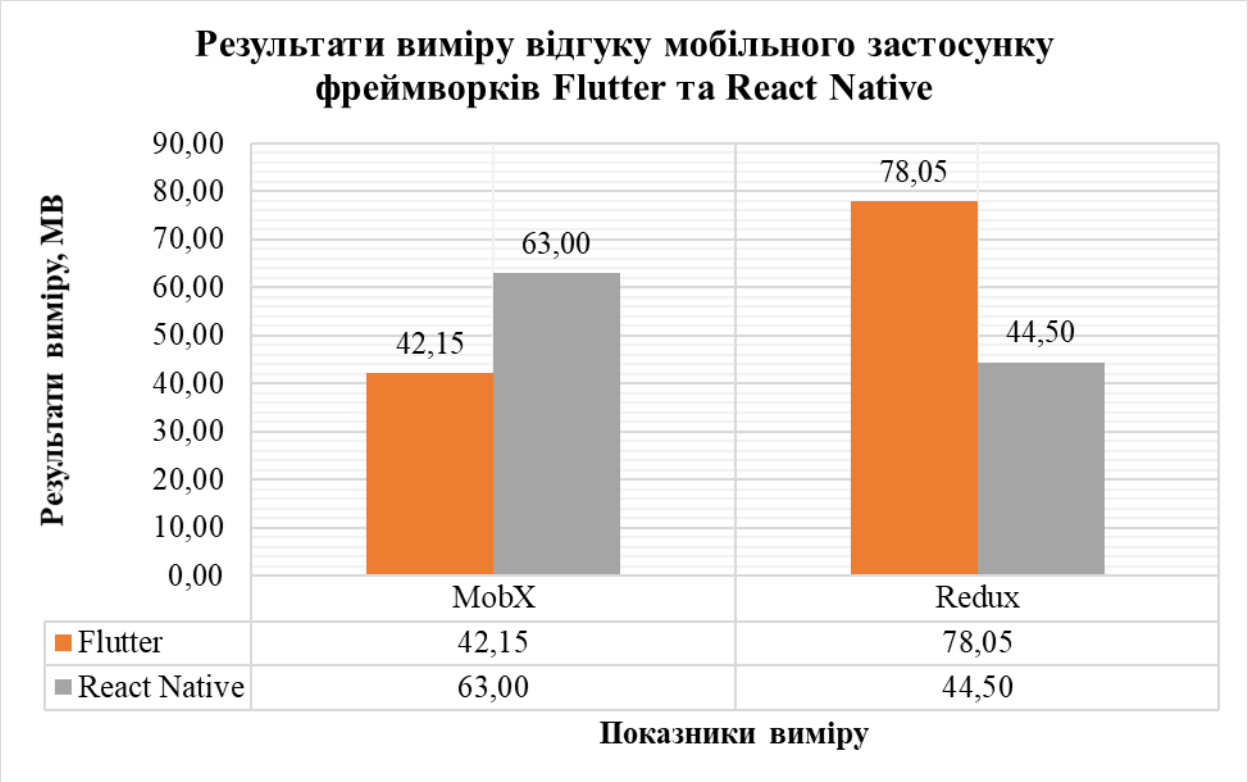


Рис. 3.23. Результат виміру відгуку мобільного застосунку фреймворків Flutter та React Native

3.3.3. Порівняння результатів використання пам’яті

Дані витраченої пам’яті необхідні для оптимізації процесів роботи застосунку, покращення користувальницького досвіду, підвищення стабільності, відповідності мобільним платформам і оптимізації старих пристроїв. Для вимірювання використовуваної пам’яті були задіяні сторонні програмні засоби які допомогли в цьому. Дані, які були зібрані:

- об’єм пам’яті завантажених бібліотек, стек та кучу;
- об’єм кучі на даний момент;
- об’єкти Dart та Flutter / JavaScript і React Native, які знаходяться в кучі;
- об’єкти Dart та Flutter Native / JavaScript і React Native, які не розташовані в кучі, але потрапляють в загальний об’єм пам’яті застосунку.

Результати вимірювання наведені на рисунках 3.24-3.25.

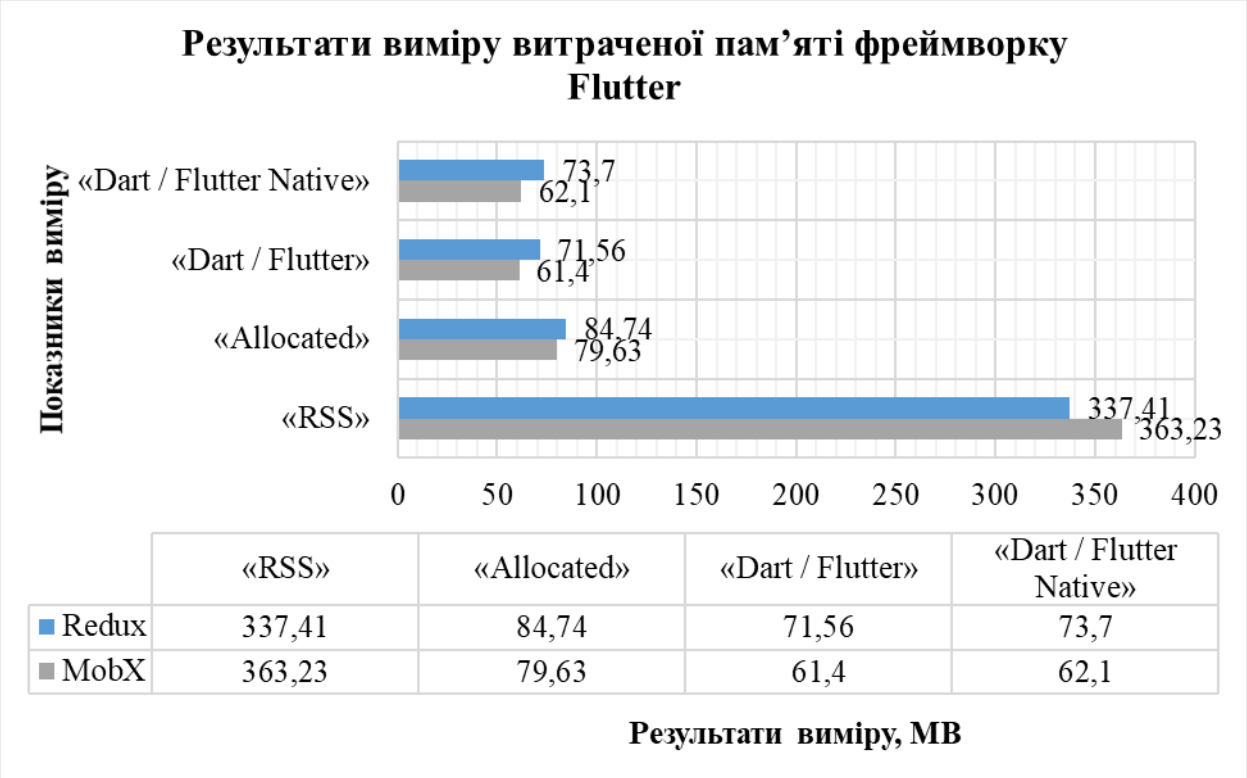


Рис. 3.24. Результати виміру витраченої пам’яті фреймворку Flutter

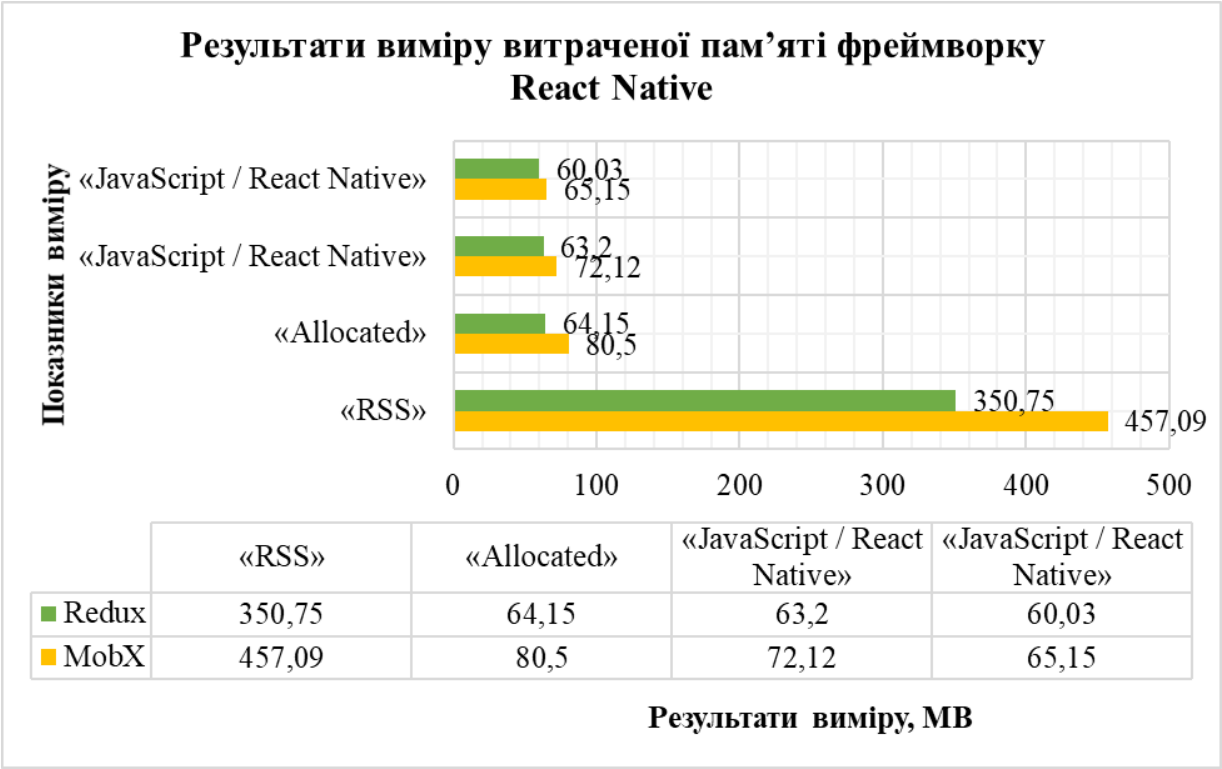


Рис. 3.25. Результати виміру витраченої пам'яті фреймворку React Native

3.3.4. Порівняння результатів використання енергії

Для визначення використання енергії мобільним застосунком були виконані налаштування у платформах пристроїв. Було виміряно використання енергії для всіх трьох екранів застосунку.

Результати наведені в таблицях 3.6-3.7.

Таблиця 3.6

Результати використання енергії мобільного застосунку фреймворку Flutter

Екран	MobX Flutter, Вт	Redux Flutter, Вт
Список задач	0,8	1,3
Створення задач	1,0	1,6
Редагування задач	1,2	1,8

Джерело: розробка автора

Таблиця 3.7

Результати використання енергії мобільного застосунку фреймворку React Native

Екран	MobX React Native, Вт	Redux React Native, Вт
Список задач	1,1	0,9
Створення задач	1,4	1,1
Редагування задач	1,5	1,3

Джерело: розробка автора

Наглядовий результат наведено на рисунку 3.26.

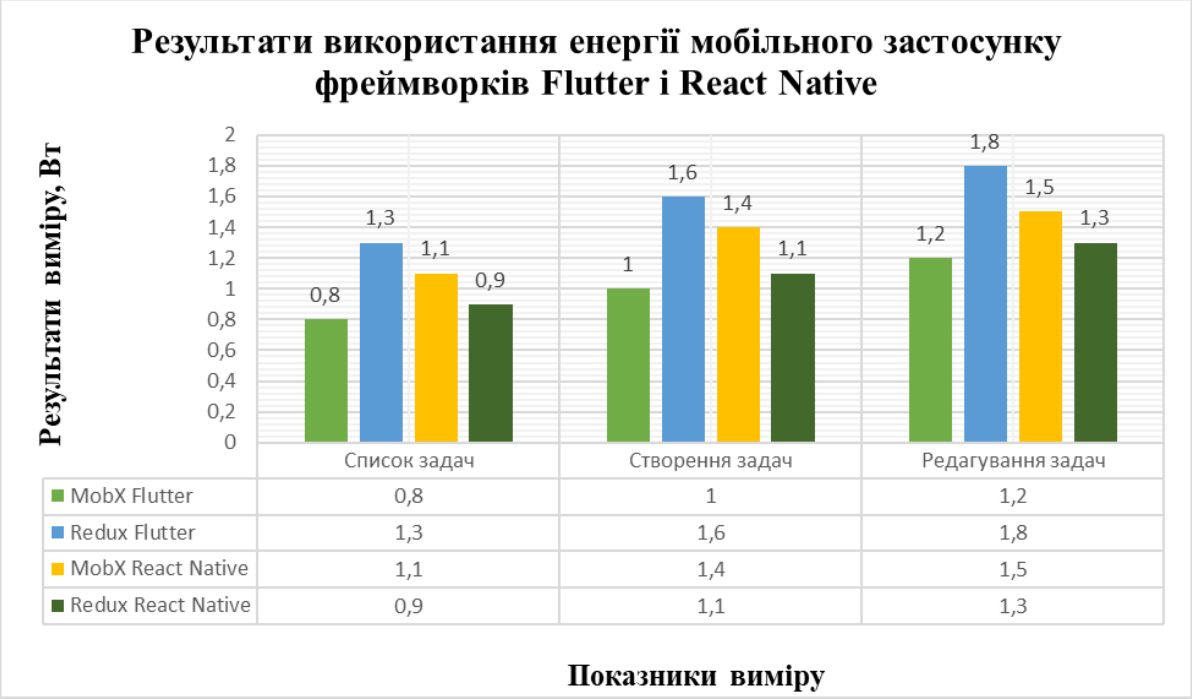


Рис. 3.26. Результат виміру використання енергії мобільного застосунку фреймворків Flutter та React Native

3.4. Висновки до розділу 3

У даному розділі були виконана програмна реалізація мобільного застосунку та порівняння методів управління станом фреймворків Flutter та

React Native.

ВИСНОВКИ

Мобільні застосунки має широкий спектр у програмній розробці. Популярність різних фреймворків зростає з кожним разом усе більше й більше. У даному дослідженні було досліджено порівняння двох найпопулярніших фреймворків у мобільній розробці: Flutter і React Native.

Було виконано теоретичний аналіз використання фреймворків у розробці мобільних застосунків і виконано їх порівняння. Були досліджені методи управління станом MobX і Redux у мобільних застосунках та виконане їх порівняння. Визначена методика дослідження використання фреймворків Flutter та React Native. Визначені вимоги до розробки мобільного застосунку. Виконане проєктування внутрішніх процесів застосунку у вигляді діаграм класів, станів, діяльності та розгортання. Уже на цьому етапі можна було побачити різницю між фреймворками із взаємодією з методами управління станом. Виконана програмна розробка мобільного застосунку, використовуючи методи управління станом MobX і Redux та фреймворки Flutter і React Native. При розробці були реалізовані шари UI, логіки та даних. Було здійснено порівняння фреймворків і методів управління станом за метриками: швидкодія застосунку, час відгуку, використання пам’яті та енергії.

За результатами порівняння метрик можна сказати, що у випадку швидкодії найкращий результат в оновленні та видаленні видає Redux Flutter, додаванні – MobX React Native, у завантаженні – MobX Flutter. За результатами часу відгуку застосунку у Flutter найкращий результат видав MobX, а у React Native – Redux. За результатами використання пам’яті в об’ємі пам’яті завантажених бібліотек, стеку та кучі найкращий результат надав Redux Flutter, в об’ємі кучі на поточний момент Redux React Native, в об’єктах Dart та Flutter / JavaScript і React Native, які знаходяться в кучі – MobX Flutter, а в об’єктах Dart та Flutter Native / JavaScript і React Native, які не розташовані в кучі, але потрапляють в загальний об’єм пам’яті застосунку Redux React

Native. За результатами використання енергії найкращий результат на всіх екрана надав MobX Flutter.

Отож, як видно з наведених результатів, однозначно найкращого варіанту для розробку мобільних застосунків немає. Більше всього найкращих результатів досяг метод управління станом MobX фреймворку Flutter, а на другому місці – Redux React Native. Це цілком логічні результати, адже Redux рідко використовується та оновлюється у Flutter.

Науковою новизною є результати дослідження, які допоможуть на які можуть спиратися інші розробники під час вибору стеку технологій.

Практичною новизною у дослідженні є розробка мобільного застосунку з використанням різних підходів та стеку, що дає змогу більш широко розглянути на програмну реалізацію мобільних застосунків.

Одержані результати можуть використовувати ІТ компанії для дослідження та визначення найкращого стеку для розробки.

Подальший розвиток дослідження полягає в доданні інших методів управління станом і фреймворків до порівняння.

ПЕРЕЛІК ПОСИЛАНЬ

1. Безверхий О. І., Куценко О. І. Шляхи оптимізації кросплатформенних додатків із використанням бібліотеки react та фреймворку react native. Systems and technologies. 2024. Т. 67. С. 30–35.

2. Епіцентр К. App Store. URL: https://apps.apple.com/ua/app/епіцентр/id1489179679?l=ua.

3. Ahuja V., Khazanchi D. Creation of a Conceptual Model for Adoption of Mobile Apps for Shopping from E-Commerce Sites–An Indian Context. Procedia Computer Science. 2016. Vol. 91. P. 609–616.

4. Alessandria S. Flutter Cookbook: 100+ step-by-step recipes for building cross-platform, professional-grade apps with Flutter 3.10.x and Dart 3.x. Packt Publishing, 2023.

5. Alturas B. Connection between UML use case diagrams and UML class diagrams: a matrix proposal. International Journal of Computer Applications in Technology. 2023. Vol. 72, no. 3. P. 161–168.

6. Assunção W. K. G., Vergilio S. R., Lopez-Herrejon R. E. Reengineering UML Class Diagram Variants into a Product Line Architecture. UML-Based Software Product Line Engineering with SMarty. 2022. P. 393–414.

7. A Study of Learning Environment for Initiating Flutter App Development Using Docker / S. T. Aung et al. Information. 2024. Vol. 15, no. 4. P. 191.

8. Boduch A., Derks R., Sakhniuk M. React and React Native: Build cross-platform JavaScript applications with native power for the web, desktop, and mobile. Packt Publishing, 2022.

9. Boukhary S., Colmenares E. A Clean Approach to Flutter Development through the Flutter Clean Architecture Package. 2019 International Conference on Computational Science and Computational Intelligence (CSCI) : International Conference, Las Vegas. 2019. P. 1115–1120.

10. Boukhary S., Colmenares E. A Clean Approach to Flutter Development through the Flutter Clean Architecture Package. 2019 International Conference on

Computational Science and Computational Intelligence (CSCI) : International Conference, Las Vegas. 2019. P. 1115–1120.

11. Chen H., Xiong J. Design and implementation of venue reservation based on React Native. 2022 International Conference on Artificial Intelligence and Computer Information Technology (AICIT) : International Conference, Yichang. 2022. P. 1–4.

12. Comparative Analysis of App Size Variations between React Native and Apache Cordova Powered Android Applications / S. P. Uniyal et al. 2023 Second International Conference on Augmented Intelligence and Sustainable Systems (ICAISS) : International Conference, Trichy. 2023. P. 1697–1702.

13. Critical Review and Fine-Tuning Performance of Flutter Applications / J. Nanavati et al. 2024 5th International Conference on Mobile Computing and Sustainable Informatics (ICMCSI) : International Conference, Lalitpur. 2024. P. 838–841.

14. Design of Mobile Application Lifecycle Security Management Platform / Y. Zeng et al. 2021 International Conference on Computer Network, Electronic and Automation (ICCNEA) : International Conference, Xi'an. 2021. P. 26–30.

15. Desmurs-Linczewska A., Kraman K. Simplifying State Management in React Native: Master state management from hooks and context through to Redux, MobX, XState, Jotai and React Query. Packt Publishing, 2023. 202 p.

16. Development of deep learning-based mobile application for predicting in-situ habitat suitability of Perilla frutescens L. in real-time / N. Singh et al. Smart Agricultural Technology. 2024. Vol. 8.

17. El-Kaliouby S. S., Selim S., Yousef A. H. Native Mobile Applications UI Code Conversion. 2021 16th International Conference on Computer Engineering and Systems (ICCES) : International Conference, Cairo. 2021. P. 1–5.

18. Ev S. E., Samuel P. Automatic Code Generation From UML State Chart Diagrams. IEEE Access. 2019. P. 99.

19. Fabbiane N., Bouillaut V., Lepage A. Aeroelastic Design Of A Drone For Research On Active Flutter Control. IFASD 2024 - International Forum on

Aeroelasticity and Structural Dynamics : International Forum, Hague. 2024.

20. Govoruhina A., Nikiforova A. Digital health shopping assistant with React Native: a simple technological solution to a complex health problem. 2022 International Conference on Intelligent Data Science Technologies and Applications (IDSTA) : International Conference, San Antonio. 2022. P. 34–40.

21. Huber F., Hagel G. Work-In-Progress: Converting textual software engineering class diagram exercises to UML models. 2022 IEEE Global Engineering Education Conference (EDUCON) : Engineering Education Conference, Tunis. 2022. P. 1–3.

22. Hybrid Development in Flutter and its Widgits / S. Sharma et al. 2022 International Conference on Cyber Resilience (ICCR) : International Conference, Dubai. 2022. P. 1–4.

23. Irawan A. J., Tobing F. A. T., Surbakti E. E. Implementation of Gamification Octalysis Method at Design and Build a React Native Framework Learning Application. 2021 6th International Conference on New Media Studies (CONMEDIA) : International Conference, Tangerang. 2021. P. 118–123.

24. JavaScript in mobile applications: React native vs ionic vs NativeScript vs native development / H. Brito et al. 2018 13th Iberian Conference on Information Systems and Technologies (CISTI) : Iberian Conference, Caceres. 2018. P. 1–6.

25. Kadrija S., Memeti A., Luma-Osmani S. Development of mobile app through React Native hybrid framework. 2022 11th Mediterranean Conference on Embedded Computing (MECO) : Mediterranean Conference, Budva. 2022. P. 1–6.

26. Kulkarni R. N., Srinivasa C. K. Novel approach to transform UML Sequence diagram to Activity diagram. Journal of University of Shanghai for Science and Technology. 2021. Vol. 23, no. 7. P. 1247–1255.

27. Kuttig A. B. Professional React Native: Expert techniques and solutions for building high-quality, cross-platform, production-ready apps. Packt Publishing, 2022.

28. Larsen J. Get Programming with JavaScript. Manning, 2016.

29. Mcintosh P., Hamilton M., Schyndel R. G. v. X3D-UML: 3D UML State

Machine Diagrams. 11th International Conference, MoDELS : International Conference, Toulouse, 28 September – 3 October 2008. P. 264–279.

30. Mohammadi R. G., Barforoush A. A. Enforcing component dependency in UML deployment diagram for cloud applications. 7'th International Symposium on Telecommunications (IST'2014) : International Symposium, Tehran. 2014. P. 412–417.

31. Mukhamedin A. A., Abitova G. A. Research Of A React Native Vs Flutter, Cross-Platform Mobile Application Frameworks. Bulletin of Shakarim University Technical Sciences. 2024. Vol. 2, no. 14. P. 26–33.

32. Multiclass Classification of UML Diagrams from Images Using Deep Learning / S. Shcherban et al. International Journal of Software Engineering. 2021. Vol. 31, no. 11.

33. Negi A. S., Vaidya H., Chauhan R. Design and Development of a LMS with MVVM Architecture. 2023 International Conference on Sustainable Emerging Innovations in Engineering and Technology (ICSEIET) : International Conference, Ghaziabad. 2023. P. 820–823.

34. Patel P., Patil N. N. Testcases Formation Using UML Activity Diagram. 2013 International Conference on Communication Systems and Network Technologies (CSNT) : International Conference, Gwalior. 2013. P. 884–889.

35. Performance and stability Comparison of React and Flutter: Cross-platform Application Development / K. Kishore et al. 2022 International Conference on Cyber Resilience (ICCR) : International Conference, Dubai. 2022. P. 1–4.

36. Perinello L., Gaggi O. Accessibility of Mobile User Interfaces using Flutter and React Native. 2024 IEEE 21st Consumer Communications & Networking Conference (CCNC) : Conference, Las Vegas. 2024. P. 1–6.

37. Qiao Y., Li Y., Li M. The Application of Mobile Technology in Educational Administration to Foster Continuous Learning and Professional Development. International Journal of Interactive Mobile Technologies (iJIM). 2024. Vol. 18, no. 12. P. 161–175.

38. Rauseo M., Zhao F., Vahdati M. Physics guided machine learning

modelling of compressor stall flutter. Journal of the Global Power and Propulsion Society. 2024. Vol. 8, no. 2. P. 295–309.

39. Renardi F., Faizah N., Koryanto L. Reimbursement Application Using Rest API Methods and Android-Based Flutter Framework (Case Study: PT. Protonema). Journal Mobile Technologies (JMS). 2023. P. 44–52.

40. Rieger C., Majchrzak T. A. Towards the definitive evaluation framework for cross-platform app development approaches. Journal of Systems and Software. 2019. Vol. 153. P. 175–199.

41. Sehgal J., Yadhav A. A Comparative Study of State Management Libraries: Redux, MobX, Recoil, and Zustand. International Journal for Research in Applied Science and Engineering Technology. 2023. Vol. 11, no. 9. P. 432–438.

42. Shah K., Sinha H., Mishra P. Analysis of Cross-Platform Mobile App Development Tools. 2019 IEEE 5th International Conference for Convergence in Technology (I2CT) : International Conference, Bombay. 2019. P. 1–7.

43. Svekis L. L., Putten M. v., Percival C. B. R. JavaScript from Beginner to Professional: Learn JavaScript quickly by building fun, interactive, and dynamic web apps, games, and pages. Packt Publishing, 2021.

44. Wambua A. W. What Do Flutter Developers Ask About? An Empirical Study on Stack Overflow Posts. Journal of Software Engineering Research and Development. 2024. Vol. 12, no. 1.

45. Windmill E. Flutter in Action. Manning, 2020.

46. Yang S., Sahraoui H. Towards Automatically Extracting UML Class Diagrams from Natural Language Specifications. 2022. 8 p. (Preprint. Cornel University).

47. Yudin A. Introduction to React Native. Building Versatile Mobile Apps with Python and REST, RESTful Web Services with Django and React. Berkeley, 2020. P. 139–183.

48. Zasso A. Flutter derivatives: Advantages of a new representation convention. Journal of Wind Engineering and Industrial Aerodynamics. 1996. Vol. 60. P. 35–47.

49. Zelinskyi R. How to Write a Proper Mobile App Requirements Document in 5 Steps. nix. URL: https://nix-united.com/blog/how-to-write-a-proper-mobile-app-requirements-document-in-5-steps/.

50. Zhao J. Application and Practice of MVC Architecture Pattern in On-the-job Internship Management System. 2022 International Conference on Networks, Communications and Information Technology (CNCIT) : International Conference, Beijing. 2022. P. 25–30.

51. Zhuang Y., Yuan G. Study on coupled mode flutter parameters of large wind turbine blades. Scientific Reports. 2024. Vol. 14.

ДОДАТКИ

Додаток А

Екрани мобільного застосунку

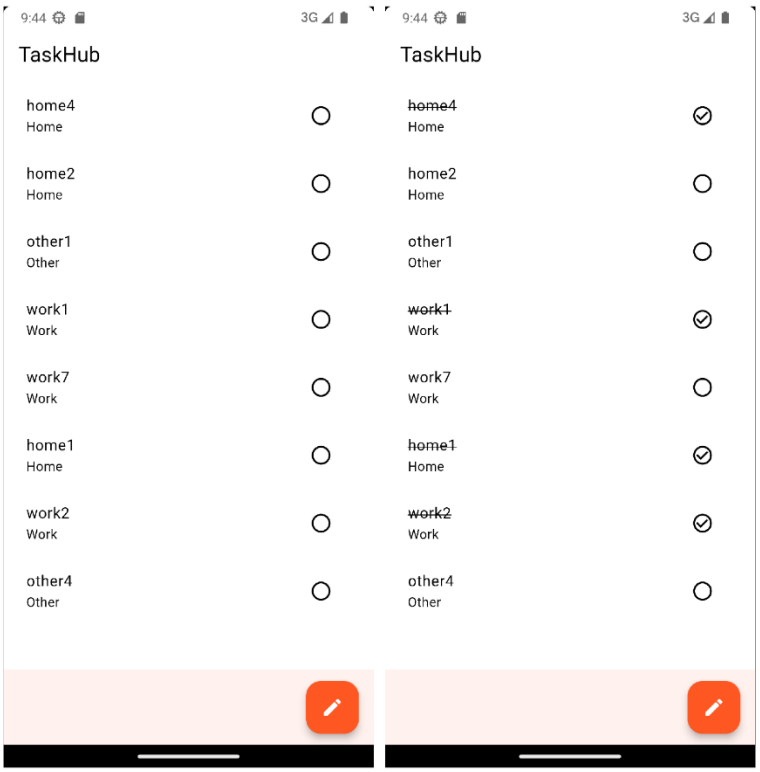


Рис. А.1. Екрани створення задач мобільного застосунку

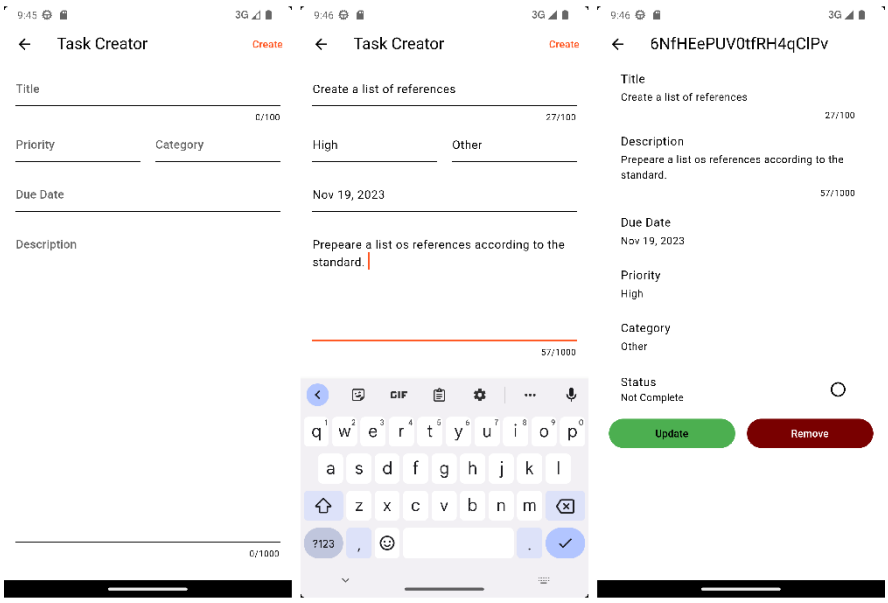


Рис. А.2. Екрани створення задач мобільного застосунку

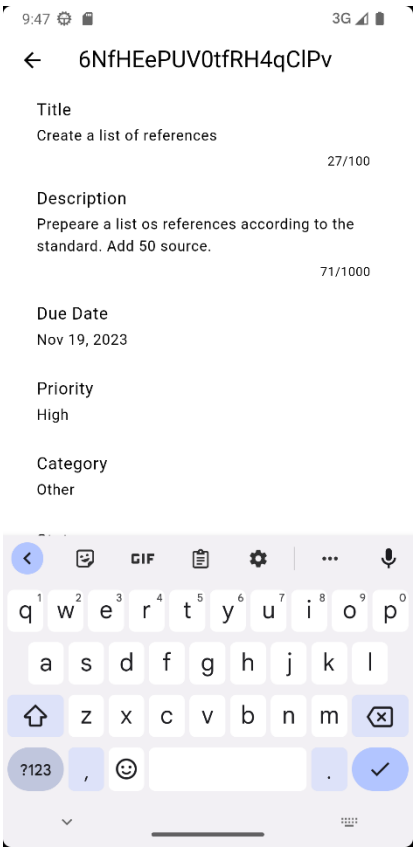


Рис. А.3. Екран редагування задач мобільного застосунку

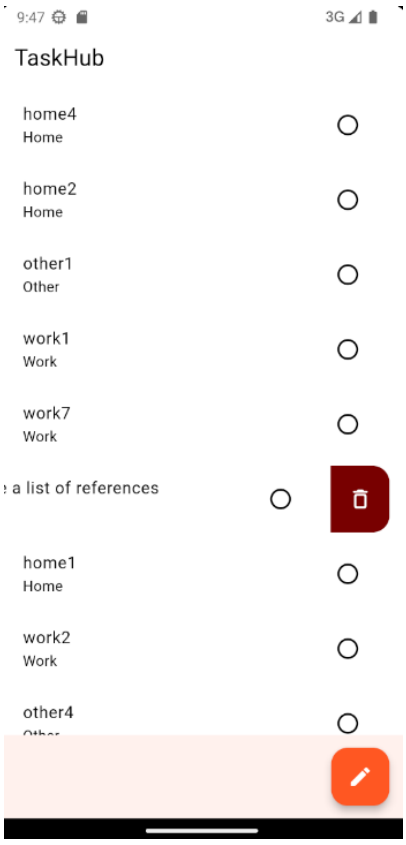


Рис. А.4. Екран видалення задач мобільного застосунку

Додаток Б

Програмний код

Б.1. editor_view.dart

```
// Створення екрану UI редагування задач flutter
class TaskEditor extends StatefulWidget {
  final TaskModel task;
  final EditorStore store;
  const TaskEditor({
    Key? key,
    required this.task,
    required this.store,
  }) : super(key: key);
  @override
  _TaskEditorState createState() => _TaskEditorState();
}
class _TaskEditorState extends State<TaskEditor> {
  late TextEditingController titleController;
  late TextEditingController descController;
  late TextEditingController dateController;
  late TextEditingController categoryController;
  late TextEditingController priorityController;
  late bool isCompleted;
  @override
  void initState() {
    super.initState();
    titleController = TextEditingController(text: widget.task.title);
    descController = TextEditingController(text: widget.task.description);
    dateController = TextEditingController(
        text: widget.task.dueDate != null
            ? DateFormat.yMMMd().format(widget.task.dueDate!)
            : '');
    categoryController = TextEditingController(text: widget.task.category);
    priorityController = TextEditingController(text: widget.task.priority);
    isCompleted = widget.task.completed;
  }
  @override
  void dispose() {
    titleController.dispose();
    descController.dispose();
    dateController.dispose();
    categoryController.dispose();
    priorityController.dispose();
    super.dispose();
  }
  bool isTaskUpdated() {
    return titleController.text != widget.task.title ||
        descController.text != widget.task.description ||
        dateController.text !=
            (widget.task.dueDate != null
                ? DateFormat.yMMMd().format(widget.task.dueDate!)
                : '') ||
        categoryController.text != widget.task.category ||
        priorityController.text != widget.task.priority ||
        isCompleted != widget.task.completed;
  }
  @override
  Widget build(BuildContext context) {
```

```
final formKey = GlobalKey<FormState>();
return ReactionBuilder(
 builder: (context) => reaction((_) => widget.store.state, (result) {
  result.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([
        _TitleInput(controller: titleController),
        _DescriptionInput(controller: descController),
        _DueDateInput(controller: dateController),
        _PriorityInput(controller: priorityController),
        _CategoryInput(controller: categoryController),
        _StatusToggle(
         status: isCompleted,
         onStatusChanged: (newStatus) {
          setState(() {
           isCompleted = newStatus;
          });
         },
        ),
        Row(
         children: [
          Expanded(
           child: Observer(
            builder: (BuildContext context) {
             final state = widget.store.state;
             return FilledButton.tonal(
              onPressed: state.mapOrNull(
               initial: (state) => () {
                HapticFeedback.vibrate();
                if (formKey.currentState!.validate()) {
                 if (isTaskUpdated()) {
                  widget.store.updateTask(
                   widget.task.copyWith(
                    category: categoryController.text,
                    completed: isCompleted,
                    description: descController.text,
                    dueDate: dateController.text.isNotEmpty
                      ? DateFormat.yMMMd().parse(
                        dateController.text)
                      : null,
```

```
                                priority: priorityController.text,
                                title: titleController.text,
                              ),
                            );
                          } else {
                            Navigator.pop(context);
                          }
                        }
                      },
                    ),
                    child: const Text('Save Changes'),
                  );
                },
              ),
            ),
            const SizedBox(width: 16.0),
            Expanded(
              child: Observer(
                builder: (BuildContext context) {
                  final state = widget.store.state;
                  return FilledButton.tonal(
                    style: FilledButton.styleFrom(
                      backgroundColor:
                          Theme.of(context).colorScheme.error,
                    ),
                    onPressed: state.mapOrNull(
                      initial: (state) => () {
                        HapticFeedback.vibrate();
                        widget.store.removeTask(widget.task.taskId);
                      },
                    ),
                    child: Text(
                      'Delete Task',
                      style: TextStyle(
                        color:
                            Theme.of(context).colorScheme.onError,
                      ),
                    ),
                  );
                },
              ),
            ),
          ],
        ),
      ]),
    ),
  ],
),
),
),
),
),
);
}
}
class _StatusToggle extends StatelessWidget {
final bool status;
final ValueChanged<bool> onStatusChanged;
const _StatusToggle({
  Key? key,
  required this.status,
  required this.onStatusChanged,
}) : super(key: key);
@override
```

```
  Widget build(BuildContext context) {
    return ListTile(
      title: const Text('Completion Status'),
      subtitle: Text(status ? 'Completed' : 'Pending'),
      trailing: IconButton(
        icon: Icon(status ? Icons.check_circle_outline : Icons.circle_outlined),
        onPressed: () {
          HapticFeedback.vibrate();
          onStatusChanged(!status);
        },
      ),
    );
  }
}
class _DescriptionInput extends StatelessWidget {
  const _DescriptionInput({
    Key? key,
    required this.controller,
  }) : super(key: key);
  final TextEditingController controller;
  @override
  Widget build(BuildContext context) {
    return ListTile(
      title: const Text('Description'),
      subtitle: TextFormField(
        controller: controller,
        keyboardType: TextInputType.text,
        decoration: const InputDecoration(
          hintText: 'Enter task description',
        ),
        maxLines: null,
        maxLength: 1000,
        validator: (String? value) {
          if (value != null && value.length > 1000) {
            return 'Description can have a maximum of 1000 characters.';
          }
          return null;
        },
      ),
    );
  }
}
class _PriorityInput extends StatelessWidget {
  const _PriorityInput({
    Key? key,
    required this.controller,
  }) : super(key: key);
  final TextEditingController controller;
  @override
  Widget build(BuildContext context) {
    final priorities = <String>['High', 'Medium', 'Low'];
    Future<void> _showPriorityDialog() async {
      final selectedPriority = await showDialog<String?>(
        context: context,
        builder: (BuildContext context) {
          return AlertDialog(
            title: const Text('Choose Priority'),
            content: Column(
              mainAxisSize: MainAxisSize.min,
              children: priorities
                  .map((priority) => ListTile(
                        title: Text(priority),
                        onTap: () {
```

```
                    Navigator.of(context).pop(priority);
                  },
                ))
            .toList(),
        ),
      );
    },
  );
  if (selectedPriority != null) {
    controller.text = selectedPriority;
  }
 }
 return ListTile(
  title: const Text('Priority'),
  subtitle: TextFormField(
   controller: controller,
   showCursor: false,
   readOnly: true,
   onTap: () => _showPriorityDialog(),
   decoration: const InputDecoration(
    hintText: 'Select priority level',
   ),
   validator: (String? value) {
    if (value == null || value.isEmpty) {
     return 'Priority selection is required.';
    }
    return null;
   },
  ),
 );
 }
}
class _TitleInput extends StatelessWidget {
 const _TitleInput({
  Key? key,
  required this.controller,
 }) : super(key: key);
 final TextEditingController controller;
 @override
 Widget build(BuildContext context) {
  return ListTile(
   title: const Text('Task Title'),
   subtitle: TextFormField(
    controller: controller,
    decoration: const InputDecoration(
     hintText: 'Enter task title',
    ),
    maxLength: 100,
    validator: (String? value) {
     if (value == null || value.isEmpty) {
      return 'Title cannot be empty.';
     } else if (value.length > 100) {
      return 'Title can have a maximum of 100 characters.';
     }
     return null;
    },
   ),
  );
 }
}
class _CategoryInput extends StatelessWidget {
 const _CategoryInput({
  Key? key,
```

```
    required this.controller,
  }) : super(key: key);
  final TextEditingController controller;
  @override
  Widget build(BuildContext context) {
    final categories = <String>['Work', 'Home', 'Other'];
    Future<void> _showCategoryDialog() async {
      final selectedCategory = await showDialog<String?>(
        context: context,
        builder: (BuildContext context) {
          return AlertDialog(
            title: const Text('Select Category'),
            content: Column(
              mainAxisSize: MainAxisSize.min,
              children: categories
                  .map((category) => ListTile(
                        title: Text(category),
                        onTap: () {
                          Navigator.of(context).pop(category);
                        },
                      ))
                  .toList(),
            ),
          );
        },
      );
      if (selectedCategory != null) {
        controller.text = selectedCategory;
      }
    }
    return ListTile(
      title: const Text('Category'),
      subtitle: TextFormField(
        controller: controller,
        showCursor: false,
        readOnly: true,
        onTap: () => _showCategoryDialog(),
        decoration: const InputDecoration(
          hintText: 'Select task category',
        ),
        validator: (String? value) {
          if (value == null || value.isEmpty) {
            return 'Category selection is required.';
          }
          return null;
        },
      ),
    );
  }
}
class _DueDateInput extends StatelessWidget {
  const _DueDateInput({
    Key? key,
    required this.controller,
  }) : super(key: key);
  final TextEditingController controller;
  @override
  Widget build(BuildContext context) {
    Future<void> _selectDate(BuildContext context) async {
      final picked = await showDatePicker(
        context: context,
        initialDate: DateTime.now(),
        firstDate: DateTime(2000),
```

```
        lastDate: DateTime(2100),
      );
      if (picked != null) {
        controller.text = DateFormat.yMMMd().format(picked);
      }
    }
    return ListTile(
      title: const Text('Due Date'),
      subtitle: TextFormField(
        controller: controller,
        showCursor: false,
        readOnly: true,
        onTap: () => _selectDate(context),
        decoration: const InputDecoration(
          hintText: 'Pick due date',
        ),
      ),
    );
  }
}
```

Б.2. editor_store.dart

```
// Створення логіки екрану редагування задач mobx flutter
part 'editor_store.freezed.dart';
part 'editor_store.g.dart';
part 'editor_state.dart';
class TaskEditorStore = _TaskEditorStoreBase with _$TaskEditorStore;
abstract class _TaskEditorStoreBase with Store {
  final TasksRepository tasksRepository;
  _TaskEditorStoreBase({required this.tasksRepository});
  @observable
  EditorState state = const EditorState.initial();
  @action
  Future<void> editTask(final TaskModel task) async {
    state = const _Loading();
    try {
      await tasksRepository.updateTask(task: task);
      state = _SuccessUpdate(task: task);
    } catch (error) {
      state = _Failure(error: error);
      state = const _Initial();
      rethrow;
    }
  }
  @action
  Future<void> deleteTask(final String taskId) async {
    state = const _Loading();
    try {
      await tasksRepository.deleteTask(taskId: taskId);
      state = _SuccessRemove(taskId: taskId);
    } catch (error) {
      state = _Failure(error: error);
      state = const _Initial();
      rethrow;
    }
  }
}
```

Б.3. editor_reducer.dart

```
// Створення логіки екрану редагування задач redux flutter
part 'editor_actions.dart';
part 'editor_state.dart';
part 'editor_reducer.freezed.dart';
class TaskEditorReducer {
  final TasksRepository tasksRepository;
  TaskEditorReducer({required this.tasksRepository});
  Store<EditorState> initializeStore() => Store<EditorState>(
    initialState: const EditorState.initial(),
    environment: tasksRepository,
  );
}
```

Б.4. task_edit_view.js

```
// Створення UI екрану редагування задач react native
// Define the EditStore class
class EditStore {
  state = 'initial'; // or other initial state
  constructor() {
    makeAutoObservable(this);
  }
  // Mock functions to simulate async updates
  async updateTask(task) {
    // Simulate an async action
    this.state = 'successUpdate'; // Example state change
  }
  async removeTask(taskId) {
    // Simulate an async action
    this.state = 'successRemove'; // Example state change
  }
}
// Instantiate store
const store = new EditorStore();
// Define the EditorView component
const EditView = observer(({ route }) => {
  const { task } = route.params;
  const [name, setName] = useState(task.title);
  const [description, setDescription] = useState(task.description);
  const [dueDate, setDueDate] = useState(task.dueDate ? new Date(task.dueDate) : new Date());
  const [category, setCategory] = useState(task.category);
  const [priority, setPriority] = useState(task.priority);
  const [status, setStatus] = useState(task.completed);
  useEffect(() => {
    // Update state if needed
  }, [store.state]);
  const handleUpdateTask = async () => {
    if (hasTaskChanged()) {
      await store.updateTask({
        ...task,
        title: name,
        description,
        dueDate,
        category,
        priority,
        completed: status,
      });
    } else {
      // Navigate back if no changes
    }
```

```
};
const handleRemoveTask = async () => {
  await store.removeTask(task.taskId);
};
const hasTaskChanged = () => {
  return name !== task.title ||
    description !== task.description ||
    dueDate !== new Date(task.dueDate) ||
    category !== task.category ||
    priority !== task.priority ||
    status !== task.completed;
};
return (
  <View style={{ padding: 16 }}>
    <Text>Title</Text>
    <TextInput
      value={name}
      onChangeText={setName}
      placeholder="Title"
      style={{ borderBottomWidth: 1, marginBottom: 8 }}
    />
    <Text>Description</Text>
    <TextInput
      value={description}
      onChangeText={setDescription}
      placeholder="Description"
      multiline
      style={{ borderBottomWidth: 1, marginBottom: 8 }}
    />
    <Text>Due Date</Text>
    <TouchableOpacity onPress={() => showDatePicker()}>
      <TextInput
        value={dueDate.toDateString()}
        editable={false}
        style={{ borderBottomWidth: 1, marginBottom: 8 }}
      />
    </TouchableOpacity>
    <Text>Priority</Text>
    <TouchableOpacity onPress={() => openPriorityDialog()}>
      <TextInput
        value={priority}
        editable={false}
        style={{ borderBottomWidth: 1, marginBottom: 8 }}
      />
    </TouchableOpacity>
    <Text>Category</Text>
    <TouchableOpacity onPress={() => openCategoryDialog()}>
      <TextInput
        value={category}
        editable={false}
        style={{ borderBottomWidth: 1, marginBottom: 8 }}
      />
    </TouchableOpacity>
    <Text>Status</Text>
    <TouchableOpacity onPress={() => setStatus(!status)}>
      <Text>{status ? 'Complete' : 'Not Complete'}</Text>
    </TouchableOpacity>
    <Button title="Update" onPress={handleUpdateTask} />
    <Button title="Remove" color="red" onPress={handleRemoveTask} />
  </View>
);
function showDatePicker() {
  DateTimePickerAndroid.open({
```

```
      mode: 'date',
      value: dueDate,
      onChange: (event, selectedDate) => {
        if (selectedDate) {
          setDueDate(selectedDate);
        }
      },
    });
  }
  function openPriorityDialog() {
    // Open a dialog to select priority
    Alert.alert('Select a Priority', '', [
      { text: 'High', onPress: () => setPriority('High') },
      { text: 'Medium', onPress: () => setPriority('Medium') },
      { text: 'Low', onPress: () => setPriority('Low') },
      { text: 'Cancel', style: 'cancel' },
    ]);
  }
  function openCategoryDialog() {
    // Open a dialog to select category
    Alert.alert('Select a Category', '', [
      { text: 'Work', onPress: () => setCategory('Work') },
      { text: 'Home', onPress: () => setCategory('Home') },
      { text: 'Other', onPress: () => setCategory('Other') },
      { text: 'Cancel', style: 'cancel' },
    ]);
  }
});
export default EditView;
```

Б.5. task_edit_store.js

```
// Створення логіки екрану редагування задач mobx react native
class EditStore {
    // Observable state
    state = 'initial'; // Possible values: 'initial', 'loading', 'successUpdate', 'successRemove', 'failure'
    task = null; // TaskModel equivalent
    taskId = null; // For successRemove state
    error = null; // For failure state
    constructor(tasksRepository) {
        makeAutoObservable(this);
        this._tasksRepository = tasksRepository;
    }
    // Set state to initial
    setInitial() {
        this.state = 'initial';
        this.task = null;
        this.taskId = null;
        this.error = null;
    }
    // Set state to loading
    setLoading() {
        this.state = 'loading';
        this.task = null;
        this.taskId = null;
        this.error = null;
    }
    // Set state to successUpdate
    setSuccessUpdate(task) {
        this.state = 'successUpdate';
        this.task = task;
        this.taskId = null;
```

```
        this.error = null;
    }
    // Set state to successRemove
    setSuccessRemove(taskId) {
        this.state = 'successRemove';
        this.task = null;
        this.taskId = taskId;
        this.error = null;
    }
    // Set state to failure
    setFailure(error) {
        this.state = 'failure';
        this.task = null;
        this.taskId = null;
        this.error = error;
    }
    // Update task action
    async updateTask(task) {
        this.setLoading();
        try {
            await this._tasksRepository.updateTask(task);
            runInAction(() => {
                this.setSuccessUpdate(task);
            });
        } catch (error) {
            runInAction(() => {
                this.setFailure(error);
                this.setInitial();
            });
            throw error; // Re-throw to handle it outside if needed
        }
    }
    // Remove task action
    async removeTask(taskId) {
        this.setLoading();
        try {
            await this._tasksRepository.deleteTask(taskId);
            runInAction(() => {
                this.setSuccessRemove(taskId);
            });
        } catch (error) {
            runInAction(() => {
                this.setFailure(error);
                this.setInitial();
            });
            throw error; // Re-throw to handle it outside if needed
        }
    }
}
// Create and export an instance of EdirStore
const tasksRepository = new TasksRepository(); // Adjust as needed
const editStore = new EditStore(tasksRepository);
export default editStore;
```

Б.6. task_edit_reducer.js

```
// Створення логіку екрану редагування задачі redux react native
// reducers/editorReducer.js
} from '../actions/actionTypes';
const initialState = {
  // Define your initial state here
};
```

```
const editorReducer = (state = initialState, action) => {
  switch (action.type) {
    case REMOVE_TASK_SUCCESS:
      // Handle success case
      return {
        ...state,
        // Update state to reflect successful task removal
      };
    case REMOVE_TASK_FAILURE:
      // Handle failure case
      return {
        ...state,
        // Update state to reflect failed task removal
      };
    case UPDATE_TASK_SUCCESS:
      // Handle success case
      return {
        ...state,
        // Update state to reflect successful task update
      };
    case UPDATE_TASK_FAILURE:
      // Handle failure case
      return {
        ...state,
        // Update state to reflect failed task update
      };
    default:
      return state;
  }
};
export default editorReducer;
```