СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВОЛОДИМИРА ДАЛЯ

Навчально-науковий інститут (факультет) інформаційних технологій та електроніки
Кафедра програмування та математики

Пояснювальна записка

до магістерської дипломної роботи
магістр
(освітньо-кваліфікаційний рівень)

на тему Застосування технологій BLoC та Redux для масштабування мобільних додатків на Flutter

Виконав: студент 2 курсу, групи ІСТ-21дм
126 «Інформаційні системи та технології»
(шифр і назва спеціальності)

Морозюк І.М.
(прізвище та ініціали).

Керівник Захожай О.І.
(прізвище та ініціали)

Рецензент
(прізвище та ініціали)

Сєвєродонецьк – 2022 року

СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВОЛОДИМИРА ДАЛЯ

Навчально-науковий інститут (факультет) інформаційних технологій та електроніки
Кафедра програмування та математики

Освітньо-кваліфікаційний рівень _____магістр_____
Спеціальність ___126 «Інформаційні системи та технології»__
(шифр і назва спеціальності)

ЗАТВЕРДЖУЮ
Завідувач кафедри ПМ

_____д.т.н., доц. Лифар В.О.
(підпис)
«____»_____202__р.

ЗАВДАННЯ
на магістерську дипломну роботу студенту

_____Морозюку Івану Миколайовичу_____
(прізвище, ім'я, по батькові)

1. Тема роботи _____Застосування технологій BLoC та Redux для масштабування мобільних додатків на Flutter_____

керівник роботи _____Захожай Олег Ігорович, д.т.н., доц _____
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу від «__»__202_року №_

2. Строк подання студентом роботи _____

3. Вихідні дані до роботи _____
_____
_____

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) _____
_____
_____

5. Перелік графічного матеріалу (з точним значенням обов'язків креслень)_____
_____
_____

6. Консультанти розділів проекту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання _____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка
1	Одержання завдання на виконання роботи	08.10.22	
2	Укладання і погодження з керівником плану і етапів виконання роботи	10.10.22	
3	Узагальнення даних літературних джерел	13.10.22	
4	Аналіз шляхів виконання завдання. Вибір і | погодження з керівником оптимального шляху виконання завдання	17.10.22	
5	Аналіз технічних засобів та існуючих систем	25.10.22	
6	Реалізація практичної частини завдання	01.11.22	
7	Укладання, оформлення та погодження пояснювальної записки з керівником	08.11.22	
8	Здача пояснювальної записки на кафедру	20.11.22	
9	Підготовка доповіді та презентації	25.11.22	

Студент _____ ___Морозюк І.М.___
(підпис) (прізвище та ініціали).

Керівник роботи _____ ___Захожай О.І.___
(підпис) (прізвище та ініціали).

ЗМІСТ

ВСТУП 6
РОЗДІЛ 1. АНАЛІЗ ТЕРМІНУ АРХІТЕКТУРА У FLUTTR РОЗРОБЦІ 9
1.1. Ефемерний стан і стан додатку 10
1.2. Огляд терміну архітектура 13
1.3. Загальні архітектурні принципи 14
1.3.1. Принцип розділення відповідальності 14
1.3.2. Принцип керування інтерфейсом користувача за допомогою моделей даних 15
1.3.3. Принцип єдине джерело правди 15
1.3.4. Принцип односпрямований потік даних 16
1.4. Основні архітектурні рівні 17
РОЗДІЛ 2. АНАЛІЗ АРХІТЕКТУР BLoC ТА REDUX 21
2.1 Перетворення стану додатку у інтерфейс користувача у Flutter 21
2.2. Аналіз архітектурного підходу BLoC у Flutter розробці 25
2.2.1. Огляд компонентів BLoC 25
2.2.2. Різниця між Cubit і Bloc 27
2.2.3. Огляд архітектури 28
2.3. Аналіз архітектурного підходу Redux у Flutter розробці 32
2.3.1. Огляд компонентів архітектури Redux 35
2.3.2. Огляд взаємозв’язку компонентів архітектури Redux 38
2.4. Проміжні висновки 39
РОЗДІЛ 3. АНАЛІЗ ВПЛИВУ ОБРАНОЇ АРХІТЕКТУРИ НА ЖИТТЄВИЙ ЦИКЛ ПРОДУКТУ 40
3.1. Вплив обраної архітектури на розробку та підтримку продукту 41
3.1.1. Аналіз життєвого циклу функції логування у додаток з використанням архітектури BLoC 42
3.1.2. Аналіз життєвого циклу функції логування у додаток з використанням архітектури Redux 43

3.1.3. Порівняння підходів BLoC та Redux для реалізації екрану логування 44

3.2. Вплив обраної архітектури на продукт з точки зору бізнесу 44

ВИСНОВКИ 46

СПИСОК ЛІТЕРАТУРИ 47

ДОДАТОК А.....48

ВСТУП

У нашому житті все більш важливу роль відіграють смартфони. Їх використовують для комунікацій, роботи, керування фінансами, розваг, тощо. Разом з цим зростає кількість різноманітних додатків у наших смартфонах.

З часом зростає складність цих додатків. Додається новий функціонал, модифікується старий. З'являються нові технології у розробці. Все це потребує підтримки з боку розробників. Зі зростанням складності, зростає час витрачений на імплементацію нових функцій, модифікацію, тощо. А це в свою чергу прямим чином впливає на вартість продукту.

Також якщо брати до уваги той факт, що зараз у світі поширені дві мобільні платформи(Android та iOS), і частіше за все бізнес вимушений підтримувати обидві ці платформи. Виходячи з цього вартість розробки стає в середньому удвічі дорожче.

Тому бізнес завжди прагнув скоротити витрати на розробку та підтримку продукту при цьому не знижуючи якості продукту для кінцевого користувача.

Один з доступних способів заощадити – це використовувати кросплатформові рішення, такі як Flutter. Це дозволяє заощадити на скороченні кількості необхідних розробників. Вам більше не потрібно утримувати дві команди розробників на кожну з основних мобільних платформ.

З точки зору розробки можна заощадити за допомогою якісно написаного вихідного коду додатка. Добре спланований, структурований код значно легше модифікувати, масштабувати, повторно використовувати різні компоненти цього коду. Добре спланована та гарно підібрана під конкретний продукт архітектура здатна заощадити багато часу та фінансів.

Архітектура програмного забезпечення – це організація системи яка включає всі компоненти цієї системи, те, як вони взаємодіють один з одним, середовище, в якому вони працюють, і принципи, які використовуються для розробки програмного забезпечення.

Додаток побудований з використанням добре продуманої архітектури допомагає:

- Прискорити розробку та зменшити її вартість;
- Полегшити підтримку продукту;
- Полегшити масштабування та розвиток;
- Полегшити тестування;
- Зменшити кількість помилок.

На цей час існує багато різних архітектурних підходів у мобільній розробці. Вони різняться своєю складністю, кількістю основних компонентів, способами їх комунікації тощо.

У Flutter розробці є багато підходів, таких як Provider BLoC та Redux. Ці три підходи приведені відповідно до їх складності від легшого до найважчого. Всі вони можуть бути найкращим вибором у кожній окремій ситуації та в залежності від потреб. Наприклад Provider дуже легко впровадити та почати використовувати, але з часом буде все важче масштабувати продукт, модифікувати тощо. Redux навпаки, складніше впровадити, він потребує більшої кількості коду для імплементації того ж самого функціоналу. Новачкам складніше почати одразу його використовувати. Але на далеку перспективу цей підхід значно краще себе показує.

Отже, вибір правильної архітектури це один із перших важливих кроків у життєвому циклі продукту. Правильно вибрана архітектура допоможе полегшить розробку і підтримку продукту, а також допоможе заощадити кошти.

Об'єкт дослідження – вплив архітектури на вартість та складність розробки та підтримки мобільних додатків.

Предмет дослідження – архітектури програмного забезпечення BLoC та Redux у Flutter розробці.

Мета дослідження. Розробка методів обрання архітектури програмного забезпечення для зменшення вартості розробки мобільних та підтримки додатків за рахунок підвищення здатності масштабуватись на прикладі архітектур BLoC та Redux у Flutter розробці.

Завдання роботи:

1) Дослідити предметну область;
2) Проаналізувати вплив обраної архітектури на вартість і складність розробки;
3) Проаналізувати вплив обраної архітектури на здатність додатка масштабуватись;
4) Розробити критерії вибору архітектури для зменшення вартості та складності розробки та підтримки продукту.

РОЗДІЛ 1. АНАЛІЗ ТЕРМІНУ АРХІТЕКТУРА У FLUTTR РОЗРОБЦІ

Flutter — це набір програмного забезпечення для розробки інтерфейсу користувача з відкритим кодом, створений Google. Він використовується для розробки кросплатформних додатків для Android, iOS, Linux, macOS, Windows, Google Fuchsia та Інтернету з єдиною кодовою базою. Всі візуальні елементи у фреймворку створюються за допомогою власного графічного движка.

Побудова графічного інтерфейсу у Flutter відбувається за допомогою віджетів. Усе, що користувач бачить на екрані свого смартфону, є віджетом.

Віджети є центральною ієрархією класів у фреймворку Flutter. Віджет — це незмінний опис частини інтерфейсу користувача.

Головна ідея полягає в тому, що ви створюєте свій інтерфейс користувача з віджетів. Віджети описують, як має виглядати інтерфейс залежно від поточної конфігурацію та стану віджету. Коли стан віджета змінюється, віджет перебудовує свій інтерфейс.

Віджет у Flutter це одиниця композиції. Віджети є будівельними блоками інтерфейсу користувача додатку Flutter, і кожен віджет є незмінною декларацією частини інтерфейсу користувача.

Віджети утворюють ієрархію на основі композиції. Кожен віджет вкладається в батьківський елемент і може отримувати контекст від батьківського елемента.

Віджети можуть бути простими, для відображення тексту, та складними, представляючи цілий екран. Останні можуть містити у собі інші, менші за складністю віджети.

Під час написання додатку ви зазвичай створюєте нові віджети, які є підкласами StatelessWidget або StatefulWidget, залежно від того, чи керує ваш віджет будь-яким станом.

Наприклад, статичний текст, який не змінюється у продовж усього життєвого циклу додатку, цей віджет є stateless.

Якщо віджет повинен змінювати своє візуальне відображення чи поведінку в залежності від дій користувача, відповіді з серверу, тощо, цей віджет є stateful. Він містить у собі якусь інформацію яка може бути модифікована. І на основі цієї інформації, або стану, цей віджет будує свій інтерфейс.

1.1. Ефемерний стан і стан додатку

Окрім стану одного стану віджета, існує ще один, більш глобальний стан. Стан додатку.

У найширшому розумінні стан додатку – це все, що існує в пам’яті, коли додаток працює. Це включає в себе ассети додатку, усі змінні, які фреймворк Flutter зберігає про інтерфейс користувача, стан анімації, текстури, шрифти тощо. Хоча це найширше можливе визначення стану справедливе, воно не дуже корисне для розробки додатку.

По-перше, ви не керуєте деякими станами (наприклад, текстурами). Фреймворк обробляє це за вас. Отже, більш корисним визначенням стану є «будь-які дані, які вам потрібні для того, щоб перебудувати свій інтерфейс користувача в будь-який момент часу». По-друге, стан, яким ви самі керуєте, можна розділити на два концептуальні типи: ефемерний стан і стан додатку.

Таким чином дамо визначення цим типам стану.

Ефемерний стан (іноді його називають станом інтерфейсу користувача або локальним станом) — це стан, який можна вмістити в одному віджети.

Ось кілька прикладів:

- поточна сторінка віджета для відображення сторінок книги

- поточний хід складної анімації
- поточна вибрана вкладка на нижній панелі навігації

Інші частини дерева віджетів рідко потребують доступу до такого стану. Немає необхідності серіалізувати його, і він не змінюється складним чином.

Іншими словами, немає потреби використовувати архітектуру для такого стану. Все, що вам потрібно, це StatefulWidget.

Стан, який не є ефемерним, який використовується в багатьох частинах вашого додатку або який потрібно зберігати між сеансами користувача, ми називаємо станом додатку (іноді також називається спільним станом).

Приклади стану додатку:

- Налаштування користувача;
- Інформація для входу;
- Сповіщення в додатку соціальній мережі;
- Кошик для покупок у додатку електронної комерції;
- Прочитаний/непрочитаний стан статей у додатку новин.

Теоретично, можна використовувати StatefulWidget для керування цим станом також. Але швидкість зростання складності програмного коду у при такому підході буде занадто високою. Вже після написання коду для кількох екранів додатку автору програмного коду буде важко зрозуміти цей код.

Для прикладу візьмемо додаток інтернет магазин. На головному екрані ми бачимо список товарів, та кнопку з кошиком. При виборі товару ми попадаємо на сторінку деталей цього товару. Тут ми можемо додати цей товар до нашого кошику. При доданні товару до кошику, має відбутися декілька змін у різних частинах додатку, а саме:

- на екрані деталей товар помічається як доданий;

- на головному екрані, у списку, цей товар має бути помічений як доданий до кошику;
- лічильник на кнопці кошику має інкрементувати, показуючи, що на один товар у ньому стало більше;

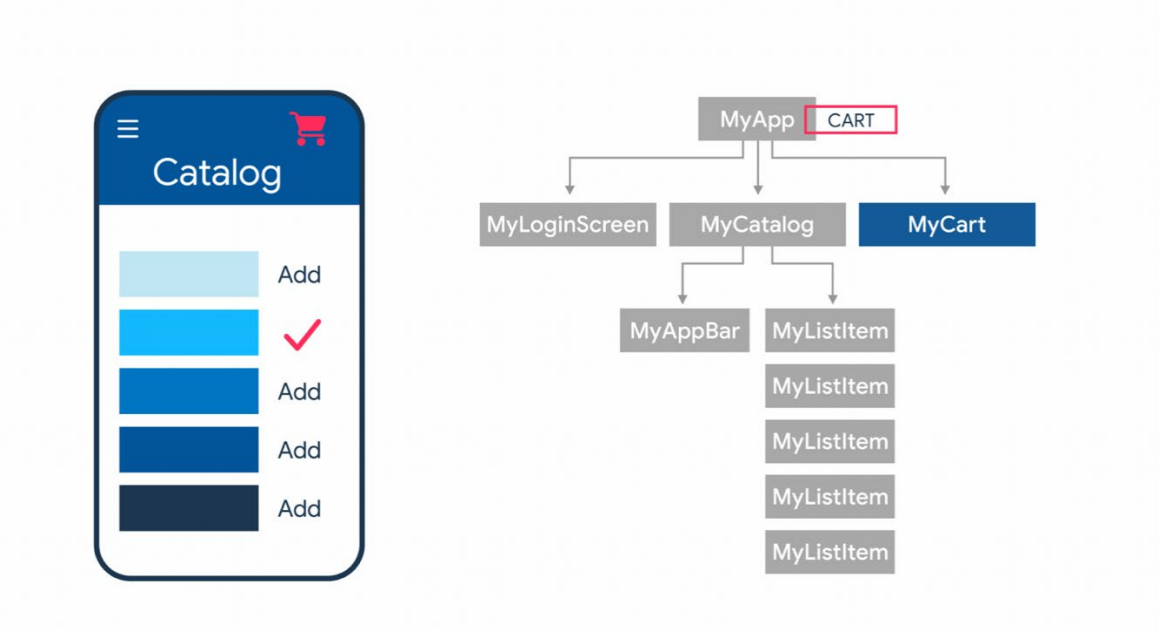

Рис. 1.1 - Візуалізація дерева віджетів додатку інтернет магазину

На рис. 1.1. зображені описані вище елементи. MyCart – кнопка кошику з лічильником. MyCatalog – список товарів. MyListItem – екран деталей.

Для досягнення бажаної поведінки з використанням StatefulWidget нам потрібно:

1. Зберігати стан кошику у віджети найвищого рівня, MyApp.
2. Передати зворотній виклик для оновлення цього стану через усе дерево віджетів від MyApp до MyListItem. Тобто MyCatalog буде приймати цей зворотній виклик тільки для того, щоб передати його MyListItem.

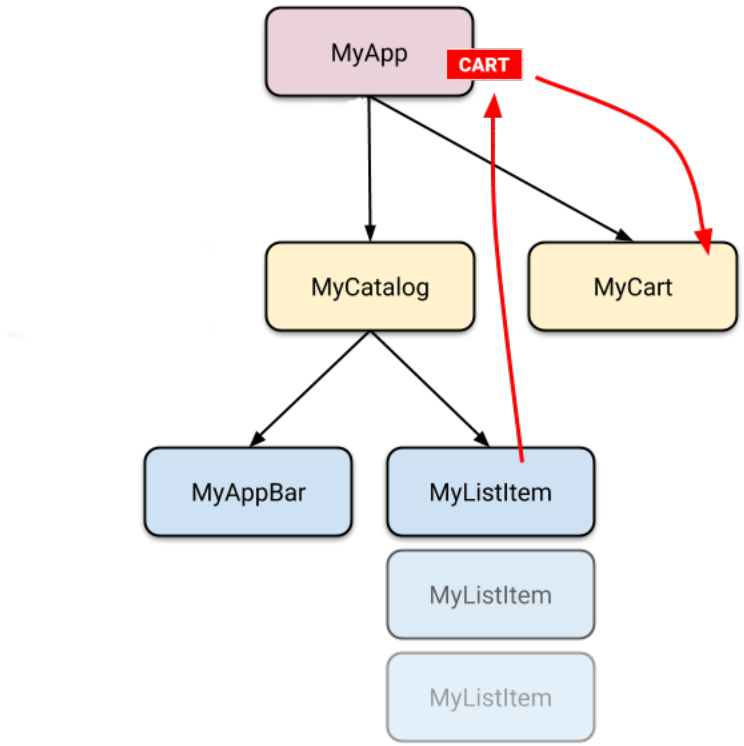

Рис. 1.2 - Схема роботи зворотного виклику для оновлення стану кошику

Цей підхід швидко перетвориться у так званий callback hell. Коли код складається із постійних вкладених один у інший зворотних викликів. Такий код важко читати, писати, підтримувати, масштабувати та модифікувати.

А якщо до команди приєднається новий розробник, він навряд зможе ознайомитись з кодом та почати писати свій за декілька днів або навіть тижнів, в залежності від складності додатку. Як наслідок, втрати часу і грошей. Не вигідно.

1.2. Огляд терміну архітектура

Для ефективного керування станом додатку, код організується у систему маленьких, якомога незалежних один від іншого елементів. Кожен

з цих елементів відповідає за конкретну операцію виконує тільки одну роботу. Система цих елементів, спосіб їх комунікації, тощо називається архітектурою[1].

Вибір архітектури є одним з перших етапів розробки додатку. Від цього вибору залежить те, наскільки складно буде підтримувати додаток, модифікувати його, тощо.

1.3. Загальні архітектурні принципи

Оскільки з часом розмір мобільного додатку зростає, важливо визначити архітектуру, яка дозволяє додатку масштабуватися, підвищує його надійність та полегшує тестування.

Архітектура додатку визначає межі між частинами додатку та обов’язки, які має мати кожна частина. Щоб задовольнити згадані вище потреби, треба обрати архітектуру свого додатка відповідно до кількох конкретних принципів.

1.3.1. Принцип розділення відповідальності

Найважливішим принципом, якого слід дотримуватися, є розділення відповідальності. Поширеною помилкою є написання всього коду додатку в віджетах. Ці класи на основі стану додатку повинні містити лише опис інтерфейсу користувача на основі ефемерного стану та стану додатку. Код для взаємодії з сервером виділяється у окремий компонент, який часто називають сервісом чи репозиторієм. Цей компонент відповідає тільки за взаємодію с сервером. Наприклад через HTTP протокол запитує у сервера данні, і повертає відповідь. Обробкою цієї відповіді буде займатись вже інший компонент.

1.3.2. Принцип керування інтерфейсом користувача за допомогою моделей даних

Інший важливий принцип полягає в тому, що ви повинні керувати своїм інтерфейсом користувача моделями даних, бажано постійними, не здатними модифікуватись моделями(immutable). Моделі даних представляють стан додатку. Вони не залежать від елементів інтерфейсу користувача та інших компонентів вашого додатку. Це означає, що вони не прив’язані до життєвого циклу інтерфейсу користувача та компонентів додатку, але все одно будуть знищені, коли операційна система вирішить видалити процес додатку з пам’яті.

Переваги незмінюваних моделей даних:

- Ваші користувачі не втратять дані, якщо операційна система знищить ваш додаток, щоб звільнити ресурси.
- Ваш додаток продовжує працювати, якщо з’єднання з мережею нестабільне або недоступне.
- Якщо ви базуєте архітектуру свого додатка на класах моделі даних, ви зробите його більш придатним для тестування та надійнішим.

1.3.3. Принцип єдине джерело правди

Коли у вашому додатку визначено новий тип даних, вам слід призначити йому “Єдине джерело правди” (Single source of truth або SSOT). SSOT є власником цих даних, і лише SSOT може їх змінювати. Щоб досягти цього, SSOT надає дані, використовуючи незмінний тип, а щоб змінити дані, SSOT надає функції або отримує події, які можуть викликати інші елементи цієї системи.

Цей принцип має багато переваг:

- Він централізує всі зміни певного типу даних в одному місці.
- Він захищає дані, щоб інші типи не могли їх підробити.
- Це робить зміни в даних більш простежуваними. Таким чином, помилки легше помітити.

У додатку, який працює в офлайн режимі, джерелом правдивих даних додатка зазвичай є база даних. У додатках які працюють в онлайн режимі все залежить від типу додатку. SSOT може бути сервер, або все ще база даних. Додаток може запитати у сервера про актуальні дані, оновити їх у локальній базі даних і передавати дали до інтерфейсу користувача саме звідти.

1.3.4. Принцип односпрямований потік даних

Принцип єдиного джерела правди часто використовується із шаблоном односпрямованого потоку даних (ОПД). В ОПД стан додатку переміщується тільки в одному напрямку у ланцюгу зв’язків компонентів архітектури. Події, які змінюють стан у протилежному напрямку.

Стан або дані зазвичай переходять від типів ієрархії з вищою областю до нижчих. Події зазвичай ініціюються з типів нижчого діапазону, доки вони не досягнуть SSOT для відповідного типу даних. Наприклад, дані додатку зазвичай надходять із джерел даних до інтерфейсу користувача. Події користувача, такі як натискання кнопок, переходять від інтерфейсу користувача до SSOT, де дані додатку змінюються та відображаються в незмінному типі.

Цей шаблон краще гарантує узгодженість даних, менш схильний до помилок, легше налагоджувати та забезпечує всі переваги шаблону SSOT.

1.4. Основні архітектурні рівні

Враховуючи загальні архітектурні принципи, згадані в попередньому розділі, кожен додаток повинен мати принаймні два рівні:

- Рівень інтерфейсу користувача, або рівень представлення, який відображає дані додатка на екрані.
- Рівень даних, який містить бізнес-логіку вашого додатку та надає дані додатку.

Ви можете додати рівень під назвою рівень домену, щоб спростити та повторно використовувати взаємодію між інтерфейсом користувача та рівнями даних.

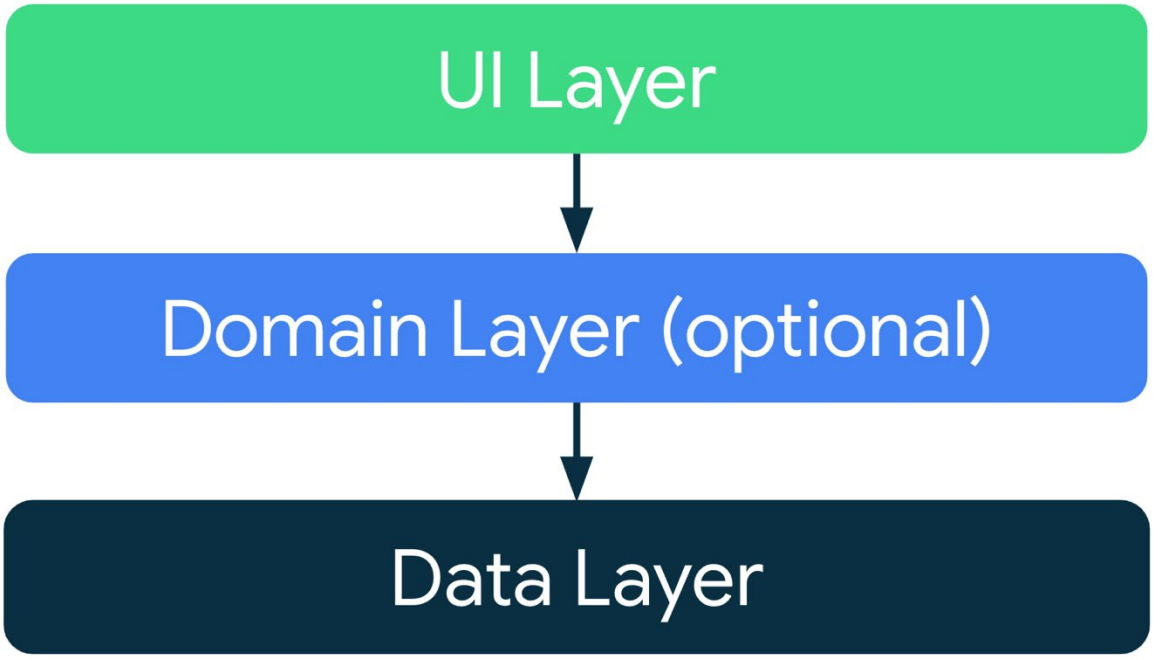

Рис. 1.3 - Базова схема архітектурних слоїв

Роль рівня інтерфейсу користувача (або рівня представлення) відповідає за відображення даних додатка на екрані. Щоразу, коли дані змінюються через взаємодію користувача (наприклад, натискання кнопки)

або зовнішній вхід (наприклад, відповідь серверу), інтерфейс користувача має оновлюватися, щоб відобразити зміни.

Рівень даних додатка містить бізнес-логіку. Бізнес-логіка — це те, що надає цінність вашому додатку — вона складається з правил, які визначають, як ваш додаток створює, зберігає та змінює дані.

Рівень даних складається із елементів, кожне з яких може містити від нуля до багатьох джерел даних. Ці елементи частіше за все називаються сервісом або репозиторієм. Ви повинні створити такий елемент для кожного окремого типу даних, які ви обробляєте у своєму додатку.

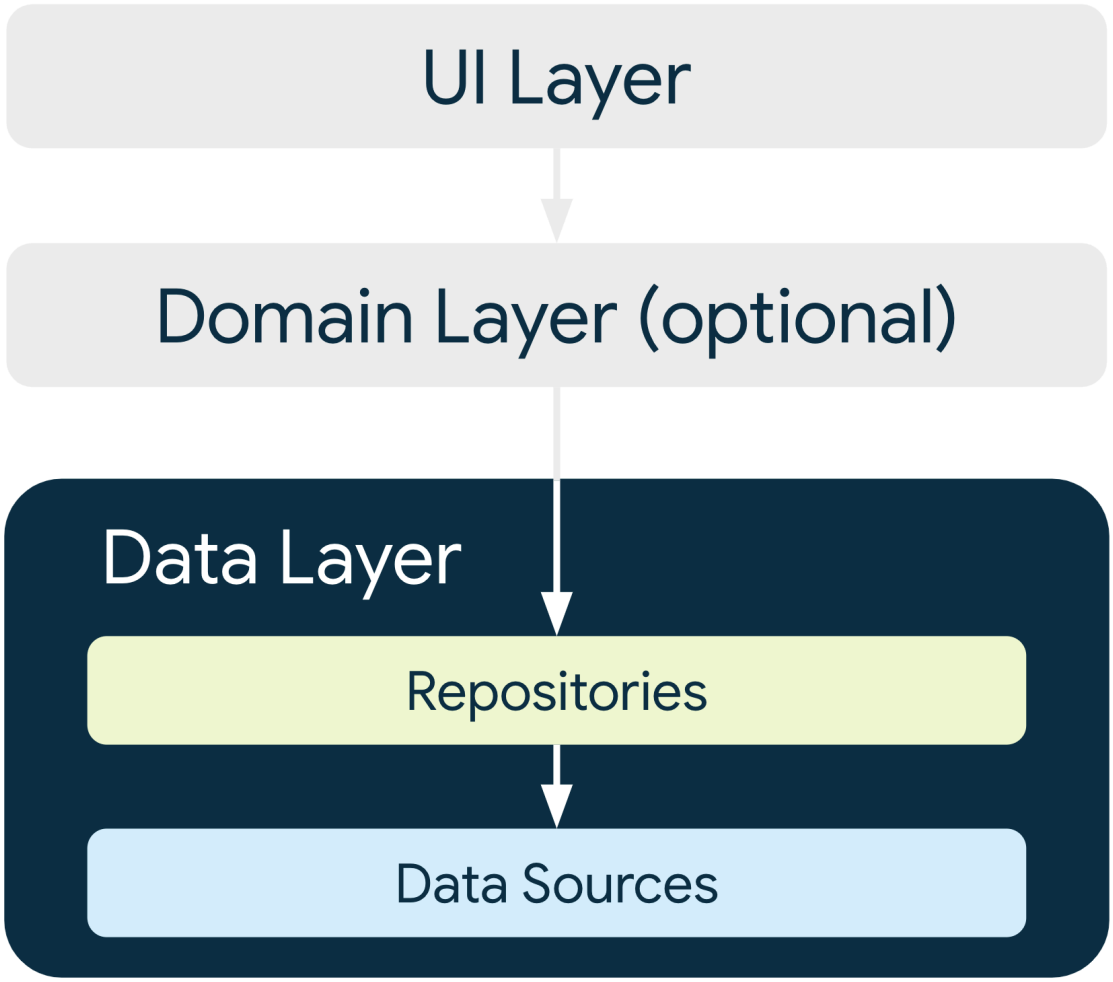

Рис. 1.4 - Схема рівня даних додатку

Елементи рівня даних відповідають за наступні завдання:

- Давати доступ до даних решті додатку.
- Централізація змін даних.
- Вирішення конфліктів між кількома джерелами даних.
- Абстрагування джерел даних від решти додатка.
- Містить бізнес-логіку.

Кожен клас джерела даних повинен нести відповідальність за роботу лише з одним джерелом даних, яким може бути файл, сервер або локальна база даних. Класи джерел даних є мостом між додатком і системою для операцій з даними.

Рівень домену – це необов’язковий рівень, який знаходиться між інтерфейсом користувача та рівнями даних.

Рівень домену відповідає за інкапсуляцію складної бізнес-логіки або простої бізнес-логіки, яка повторно використовується. Цей рівень є необов’язковим, оскільки не всі додатки мають ці вимоги. Використовуйте його лише тоді, коли це необхідно, наприклад, щоб усунути складність або віддати перевагу повторному використанню.

Керуйте залежностями між компонентами. Класи у вашому додатку залежать від інших класів, щоб правильно функціонувати. Ви можете використовувати будь-який із наведених нижче шаблонів проектування, щоб зібрати залежності певного класу:

- Ін’єкція залежностей або dependency injection(DI): ін’єкція залежностей дозволяє класам визначати свої залежності, не створюючи їх. Під час виконання інший клас відповідає за надання цих залежностей.

- Локатор служб або service locator: шаблон локатора служб надає реєстр, де класи можуть отримувати свої залежності замість того, щоб створювати їх.

Ці шаблони дозволяють вам масштабувати код, оскільки вони надають чіткі шаблони для керування залежностями без дублювання коду чи додавання складності. Крім того, ці шаблони дозволяють швидко перемикатися між тестовою та робочою реалізаціями.

Таким чином, використовуючи наведені вище принципи та правила можна побудувати архітектуру додатка за допомогою якої:

- Прискорюється розробка та зменшити її вартість;
- Полегшується підтримка продукту;
- Полегшується масштабування та розвиток;
- Полегшується тестування;
- Зменшується кількість помилок.

У далекій перспективі все це дозволяє зменшити витрати, покращити враження користувача від якісного продукту, та покращити продуктивність команди розробки.

Далі зробимо аналіз двох популярних у Flutter розробці архітектур, BLoC та Redux, та на їх прикладі проаналізуємо вплив обраної архітектури на життєвий цикл додатка, на складність його підтримки та на його здатність масштабуватись. Це допоможе досягти таких цілей:

- Підвищить якість продукту;
- Зменшить вартість розробки та підтримки;
- Значно полегшить роботу команди розробників;
- Зменшить поріг входження нової команди розробників чи окремих її членів.

РОЗДІЛ 2. АНАЛІЗ АРХІТЕКТУР BLoC ТА REDUX

Flutter має багато різноманітних підходів до керування станом додатку. Вони різняться складністю, кількістю компонентів, способами взаємодії цих компонентів, тощо. Для створення дуже простих додатків можна використати Stateful віджети, зберігати стан додатку та керувати ним прямо у віджетах. Але цей підхід не підходить для більших додатків, у яких треба описувати бізнес логіку, виконувати запити до сервера, зберігати данні у локальній базі даних тощо.

Для таких задач використовують більш складні підходи для керування станом,

Перед оглядом архітектурних підходів, їх методів взаємодії їх компонентів та іншого зробимо коротких огляд того, як ці дані, або стан додатку доходить до користувача, та перетворюється на інтерфейс користувача. Зробимо це спочатку з декількох причин:

1. У більшості архітектурних підходів цей спосіб не сильно відрізняється. Це обумовлено особливостями фреймворку.
2. Це допоможе сконцентруватися на компонентах архітектури не відволікаючись на компоненти інтерфейсу користувача.

2.1 Перетворення стану додатку у інтерфейс користувача у Flutter

Flutter поставляється з двома типами віджетів: один — це віджет Stateless, а інший — Stateful. Після створення Stateless віджети не змінюються, вони незмінні. Стан віджета Stateful може змінюватися протягом життя програми.

Часто ми хочемо передати дані з одного віджета в інший. Якщо нам потрібно передати дані за допомогою Stateful віджетів, нам потрібно передавати за допомогою конструкторів віджетів. Якщо наше дерево

віджетів містить кілька рівнів, то передавання даних з будь-якого віджета верхнього рівня до віджетів нижнього може бути досить складним. Багато віджетів між ними можуть просто діяти як носії даних. Зі зростанням додатку, кількості даних, кількості вкладених віджетів, цей підхід дуже швидку приведе до проблем з менеджментом цих даних та віджетів. Що в свою чергу сповільнить швидкість розробки та призведе до ускладнення тестування та відладки.

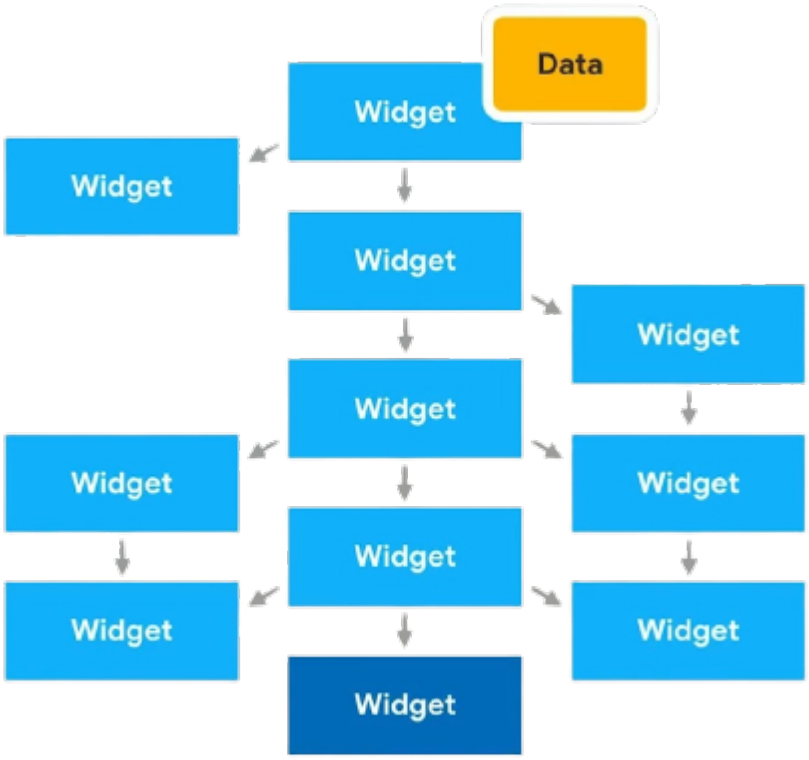

Рис. 2.1 - Дерево віджетів

На рисунку 2.1, якщо дані з самого верхнього віджета потрібні у віджеті внизу, нам потрібно передати їх із верхнього віджета через усі конструктори віджетів між ними. Це може бути досить виснажливим у великих програмах.

Але це не єдиний спосіб отримати доступ до даних або методів віджета верхнього рівня. Існує ще один тип спеціалізованих віджетів, Inherited віджет.

Inherited віджет можна додати у верхній частині дерева програми, і всі нащадкові віджети можуть успадковувати або отримати доступ до даних з цього віджета.

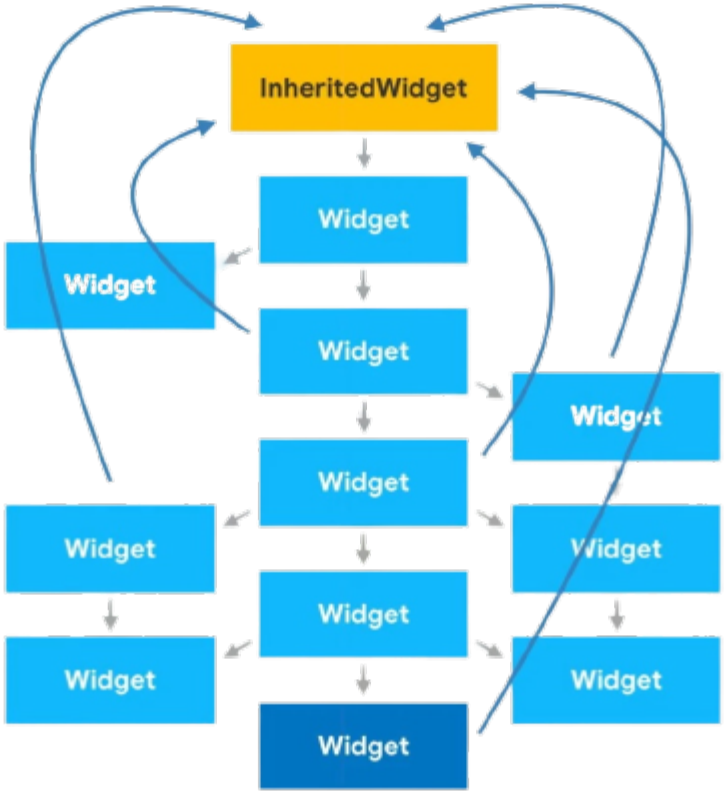

Рис. 2.2 - Inherited віджет як віджет верхнього рівня у дереві

Ми можемо створити віджет, який розширює InheritedWidget, щоб зробити його Inherited віджетом. Коли ми розширюємо InheritedWidget, ми повинні реалізувати метод updateShouldNotify, який повинен повертати логічне значення.

Коли Flutter повторно створює цей віджет, метод updateShouldNotify інформує Flutter, чи інформувати віджети нащадки, про зміни чи ні.

Щоб використовувати Inherited віджет, нам потрібно вставити його на рівень у дереві програми, з якого ми хочемо, щоб усі віджети нижче цього рівня успадковували його дані. Зазвичай розробники роблять це на самій вершині дерева додатків.

Після додавання Inherited віджета на будь-якому необхідному рівні ми можемо використовувати його в будь-якому віджеті нижче цього рівня. Для прикладу див. Додаток А.

Поширеною практикою, яку ми бачимо всюди, включаючи вбудовані Inherited віджети від Flutter (такі як Theme або MediaQuery), є створення методу static of у класі, який повертає дані.

Inherited віджети незмінні. Це означає, що всі поля в ньому остаточні, і ми не можемо змінити їх значення.

Якщо ви хочете, щоб дані всередині успадкованого віджета змінювалися протягом життя програми, тоді замість примітивних полів ми можемо використовувати класи які будуть агрегувати дані.

Inherited віджети дуже корисні в багатьох додатках. Не тільки Flutter має кілька вбудованих успадкованих віджетів, більшість популярних пакетів для керування станом програми, таких як Provider, BLoC та Redux, використовують Inherited віджети.

За їх допомоги інтерфейс користувача може спілкуватись з архітектурним компонентом відповідальним за їх комунікацію, обробку дій користувача, та передачу даних до інтерфейсу користувача для відображення.

Тепер, розуміючи що таке Inherited віджети, та як вони допомагають у організації архітектури, ми можемо приступити до огляду та аналізу одних за найпопулярніших архітектурних підходів у Flutter розробці, BLoC та Redux.

2.2. Аналіз архітектурного підходу BLoC у Flutter розробці

BLoC є однією із архітектур для додатків Flutter. BLoC досить простий у використанні, тому ви та ваша команда швидко зрозумієте концепцію, незалежно від вашого рівня. Також бібліотека має дуже хорошу документацію з безліччю прикладів, а також є однією з найбільш використовуваних у спільноті Flutter, тому, якщо у вас виникне запитання чи проблема, ви, ймовірно, знайдете рішення за допомогою простого пошуку в Інтернеті.

Є досить простим та у той же час досить потужним підходом, тому може бути використаний як для написання досить складних додатків, так і для простих. Але у той самий час BLoC використовує деякі концепції язику програмування Dart та фреймворку Flutter які потрібно розуміти для роботи з цією архітектурою. Це потоки даних(Stream, не плутати за Thread), та Inherited віджет, з якими ми вже ознайомились.

Stream — це послідовність асинхронних даних. Щоб використовувати бібліотеку BLoC, дуже важливо мати базове розуміння потоків і їх роботи.

2.2.1. Огляд компонентів BLoC

Cubit — це архітектурний компонент який відповідає за бізнес логіку додатку.

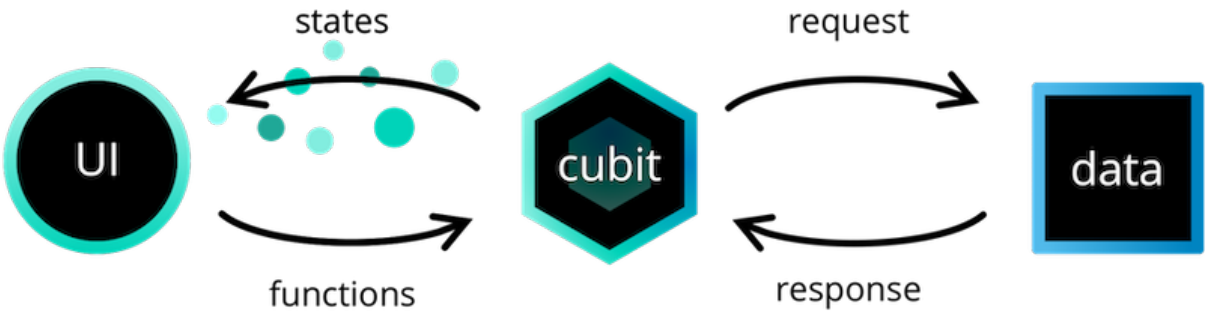

Рис. 2.3 - Відображення взаємозв’язків компоненту Cubit

Цей компонент установлює певні правила, за якими користувач може взаємодіяти з даними та змінювати стан додатку. Для цього Cubit визначає функції, за допомогою яких відбувається та чи інша операція. На рис. 2.3. цей зв'язок показаний стрілкою functions.

У відповідь на певні дії користувача чи відповіді з сервера cubit, обробивши цю подію та змінивши певним чином стан додатку сповіщає інтерфейс користувача за допомогою потоку даних через який проходить стан додатку на основі якого інтерфейс користувача перемальовується щоб відповідати актуальному стану додатка. На рис. 2.3. цей зв'язок показаний стрілкою states.

Створюючи Cubit, нам потрібно:

- Визначити тип стану, яким буде керувати Cubit. Стан може бути представлений через примітивний тип, але в більш складних випадках може знадобитися клас відповідаючий потребам додатка в цілому або його частини замість примітивного типу;
- Друге, що нам потрібно зробити під час створення Cubit, це вказати початковий стан. Ми можемо зробити це, викликавши конструктор super зі значенням початкового стану.

Bloc — це більш просунутий клас, який покладається на події, щоб ініціювати зміни стану, а не на функції. Замість того, щоб викликати функцію класу спадника Bloc та безпосередньо створювати новий стан, Bloc отримує події та перетворює вхідні події у вихідні стани. Тобто у випадку Bloc потоки використовуються і на вході і на виході.

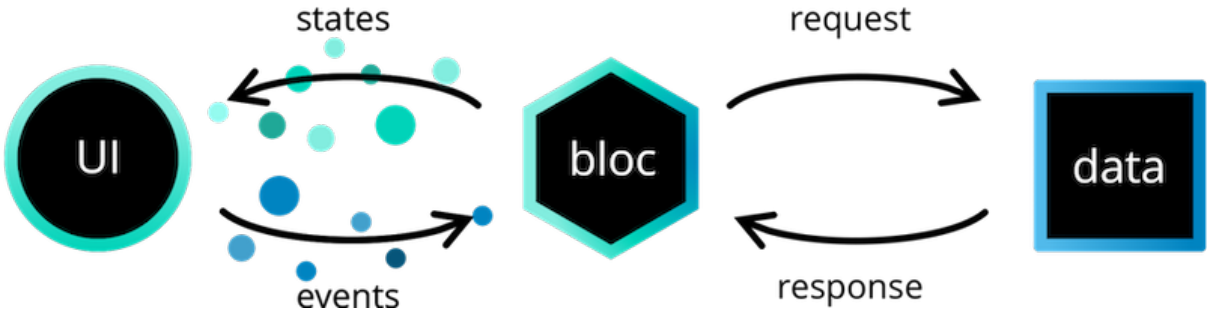

Рис. 2.4 - Відображення взаємозв'язків компоненту Cubit

Створення Bloc подібне до створення Cubit, але окрім визначення стану, яким ми будемо керувати, ми також маємо визначити подію, яку блок зможе обробити.

Bloc вимагає від нас реєстрації обробників подій через *on<Event>* API, на відміну від функцій у Cubit. Обробник подій відповідає за перетворення будь-яких вхідних подій у нуль або більше вихідних станів.

2.2.2. Різниця між Cubit і Bloc

Тепер, коли ми розглянули основи класів Cubit і Bloc, ви можете задатися питанням, коли вам слід використовувати Cubit, а коли – Bloc.

Однією з найбільших переваг використання Cubit є простота. Під час створення Cubit нам потрібно лише визначити стан, а також функції, які ми хочемо надати для зміни стану. Для порівняння, під час створення Bloc ми повинні визначити стани, події та реалізацію обробника подій. Це робить Cubit легшим для розуміння та потребує менше коду.

Реалізація Cubit більш лаконічна, і замість того, щоб визначати події окремо, функції діють як події. Крім того, використовуючи Cubit, ми можемо просто викликати emit з будь-якого місця, щоб ініціювати зміну стану.

Однією з найбільших переваг використання Bloc є знання послідовності змін стану, а також того, що саме викликало ці зміни. Для стану, який є критичним для функціональності програми, може бути дуже корисним використовувати більш керований подіями підхід, щоб фіксувати всі події на додаток до змін стану.

Може бути багато причин, чому стан програми може змінитися з одного на інший. Для прикладу візьмемо стан авторизації користувача. Наприклад, користувач міг натиснути кнопку виходу та попросити вийти з програми. З іншого боку, можливо, токен доступу користувача було

відкликано, і його було примусово виведено з системи. При використанні Bloc ми можемо чітко простежити, як стан додатку дійшов до певного стану.

Ще одна сфера, в якій Bloc перевершує Cubit, це коли нам потрібно скористатися перевагами реактивних операторів, таких як буфер, debounceTime, throttle тощо.

Bloc має приймач подій, який дозволяє нам контролювати та трансформувати вхідний потік подій.

2.2.3. Огляд архітектури

Використання бібліотеки BLoC дозволяє нам розділити нашу програму на три рівні:

- Рівень представлення;
- Бізнес-логіка;
- Доменний рівень;
    - Репозиторій;
    - Постачальник даних;

Ми почнемо з рівня найнижчого рівня (найдальшого від інтерфейсу користувача) і просунемося до рівня презентації.

Доменний рівень. Відповідальністю доменного рівня є отримання/обробка даних з одного чи кількох джерел.

Доменний рівень можна розділити на дві частини:

- Репозиторій;
- Постачальник даних;

Цей рівень є найнижчим рівнем програми та взаємодіє з базами даних, мережевими запитами та іншими асинхронними джерелами даних.

Відповідальністю постачальника даних є надання вихідних даних. Постачальник даних має бути загальним і універсальним. Постачальник даних зазвичай надає прості API для виконання операцій CRUD. Ми могли б мати метод createData, readData, updateData та deleteData як частину нашого рівня даних.

Репозиторій є оболонкою навколо одного або кількох постачальників даних, з якими взаємодіє рівень бізнес-логіки. Рівень сховища може взаємодіяти з кількома постачальниками даних і виконувати перетворення даних перед передачею результату рівню бізнес-логіки.

Рівень бізнес-логіки. Відповідальність рівня бізнес-логіки полягає в тому, щоб реагувати на вхідні дані від рівня представлення новими станами. Цей рівень може залежати від одного або кількох сховищ для отримання даних, необхідних для створення стану програми.

Думайте про рівень бізнес-логіки як про міст між інтерфейсом користувача (рівнем презентації) і доменним рівнем. Рівень бізнес-логіки отримує сповіщення про події/дії з рівня презентації, а потім зв’язується з репозиторієм, щоб створити новий стан для використання рівнем презентації.

Рівень бізнес-логіки представлений у даному архітектурному підході класами-спадниками Cubit та Bloc, оглянутими раніше.

Оскільки Bloc відкривають потоки даних, може виникнути спокуса створити Bloc, який слухає інший Bloc. Ви не повинні цього робити. Існують кращі альтернативи. Загалом слід будь-якою ціною уникати однорідних залежностей між двома об’єктами на одному архітектурному рівні, оскільки це створює тісний зв’язок, який важко підтримувати. Оскільки Bloc знаходяться на рівні архітектури бізнес-логіки, жоден Bloc не повинен знати про будь-який інший Bloc.

Блок має отримувати інформацію лише через події та з ін’єктованих репозиторіїв (тобто репозиторіїв, наданих Bloc в конструкторі).

Якщо ви опинилися в ситуації, коли Bloc повинен відповісти іншому, у вас є два інших варіанти. Ви можете перенести проблему на рівень вище (на рівень представлення) або на рівень вниз (на рівень домену).

Рівня представлення. Відповідальність рівня представлення полягає в тому, щоб визначити, як відобразити себе на основі одного або кількох станів Bloc. Крім того, він повинен обробляти введення даних користувача та події життєвого циклу програми.

Рівень представлення використовує віджет який називається BlocProvider. Це особливий тип віджету, який є спадником InheritedWidget. Цей віджет є відповідальним за створення компонентів бізнес-логіки, тобто Bloc або Cubit. Зазвичай цей віджет є найвищім у дереві віджетів відповідальних за один екран додатку.

BlocBuilder — це віджет Flutter, який обробляє створення дерева віджетів у відповідь на нові стани. Він прослуховує потік станів одного конкретного Bloc компонента та перебудовує себе кожен раз, коли стан змінився. Таким чином інтерфейс користувача завжди є актуальним.

BlocListener — це віджет Flutter, який приймає «BlocWidgetListener» і Bloc і викликає слухач у відповідь на зміни стану в Bloc. Його слід використовувати для функцій, які мають відбуватися один раз за зміну стану, як-от навігація, показ діалогового вікна тощо...

Доступ до компонентів бізнес-логіки здійснюється за допомогою InheritedWidget. BlocProvider має статичну функцію, за допомогою якої можна отримати доступ до компонента бізнес-логіки та викликати потрібну подію цього компонента.

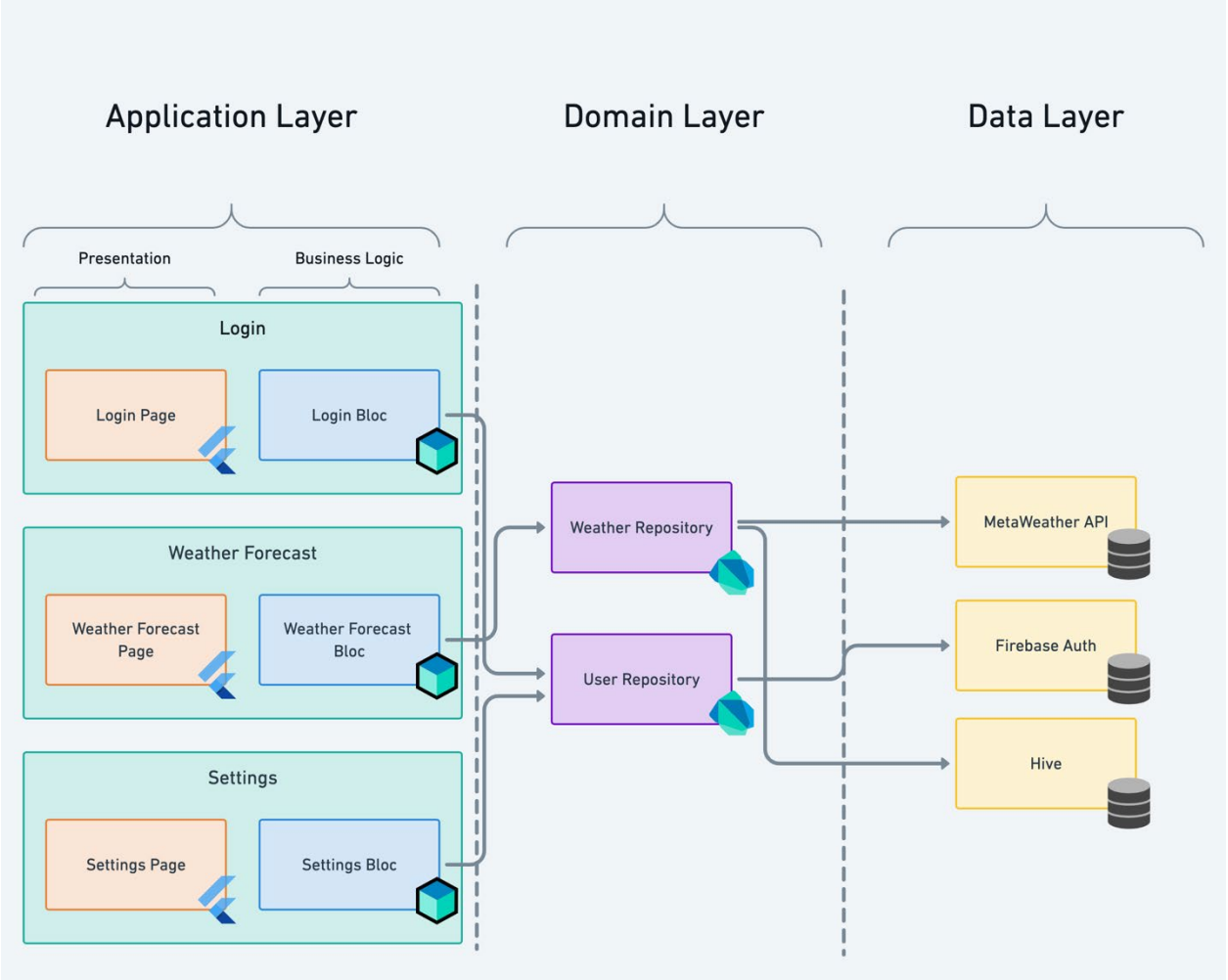

Рис. 2.5 - Схема архітектури BLoC

На рис. 2.5 приведена схема взаємодії архітектурних компонентів. Розробники мобільного додатку відповідальні за імплементацію двох рівнів. Рівня додатка, та доменного рівня. За рівень даних відповідальні Backend розробники. Взаємодія з рівнем даних відбувається за допомогою REST API, GraphQL, тощо, через постачальників даних доменного рівня.

Рівень представлення реалізовується за допомогою комбінації Flutter віджетів та пакету “flutter_bloc”, який відкриває доступ до віджетів, таких як BlocProvider, BlocBuilder, та інших.

Рівень бізнес-логіки реалізовується за допомогою пакету “bloc”, та в цілому не потребує фреймворку Flutter, що в теорії дозволяє

використовувати цю архітектуру у будь якому додатку написаному на мові програмування Dart. Але рівень презентації буде виглядати дещо інакше.

Доменний рівень це чисті класи Dart, у яких реалізовується доступ до того чи іншого сховища даних.

2.3. Аналіз архітектурного підходу Redux у Flutter розробці

Redux це більш гнучкий, та потужний архітектурний підхід, але у той самий час більш складний для розуміння, та імплементації.

Redux — це бібліотека архітектури керування станом, яка успішно розподіляє дані між віджетами повторюваним способом. Він керує станом програми через односпрямований потік даних. Давайте розглянемо діаграму нижче:

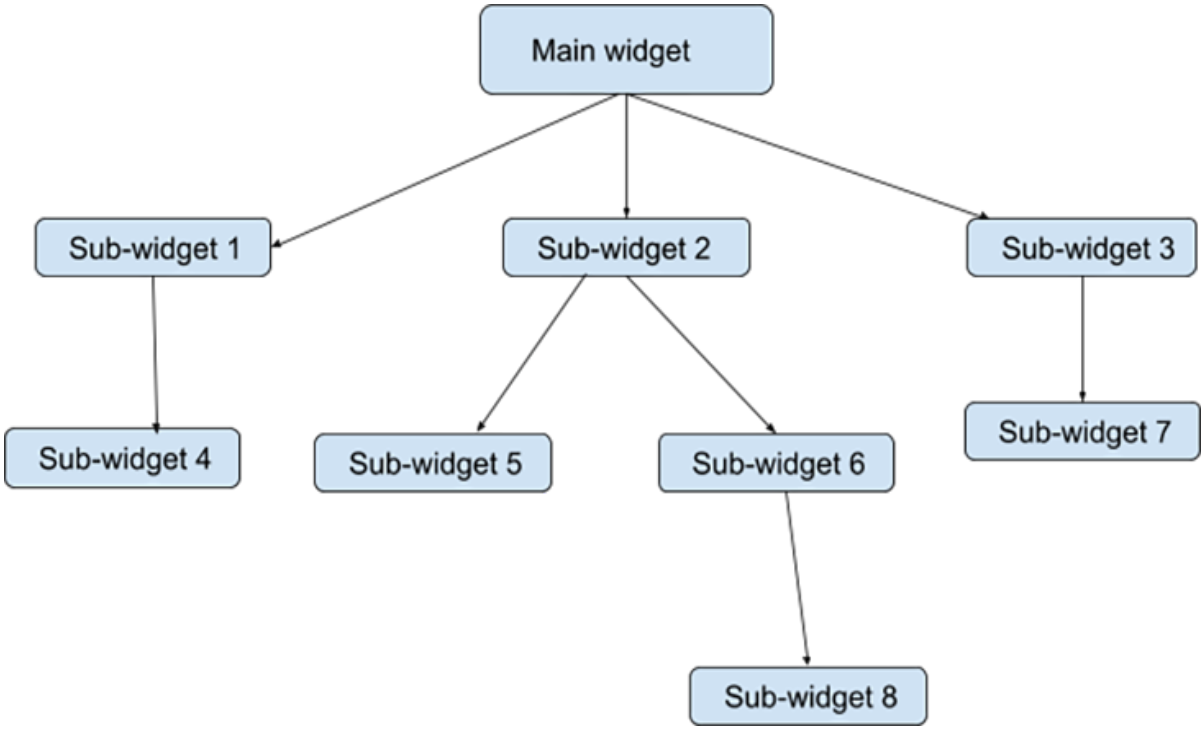

Рис. 2.6 - Типове дерево віджетів Flutter

У цьому прикладі дані, згенеровані у віджеті верхнього, потрібні у віджеті номер 8. Зазвичай ці дані проходять через віджет 2 до віджета 6, а потім, нарешті, досягають віджета 8. Це також відбувається для віджетів, яким потрібні дані, згенеровані або збережені в стані будь-якого віджета, який стоїть вище в ієрархії.

За допомогою Redux ви можете структурувати свою програму так, щоб стан зберігався в одному центральному сховищі. До даних у цьому централізованому сховищі може отримати доступ будь-який віджет, якому потрібні дані, без необхідності проходити через ланцюжок інших віджетів у дереві.

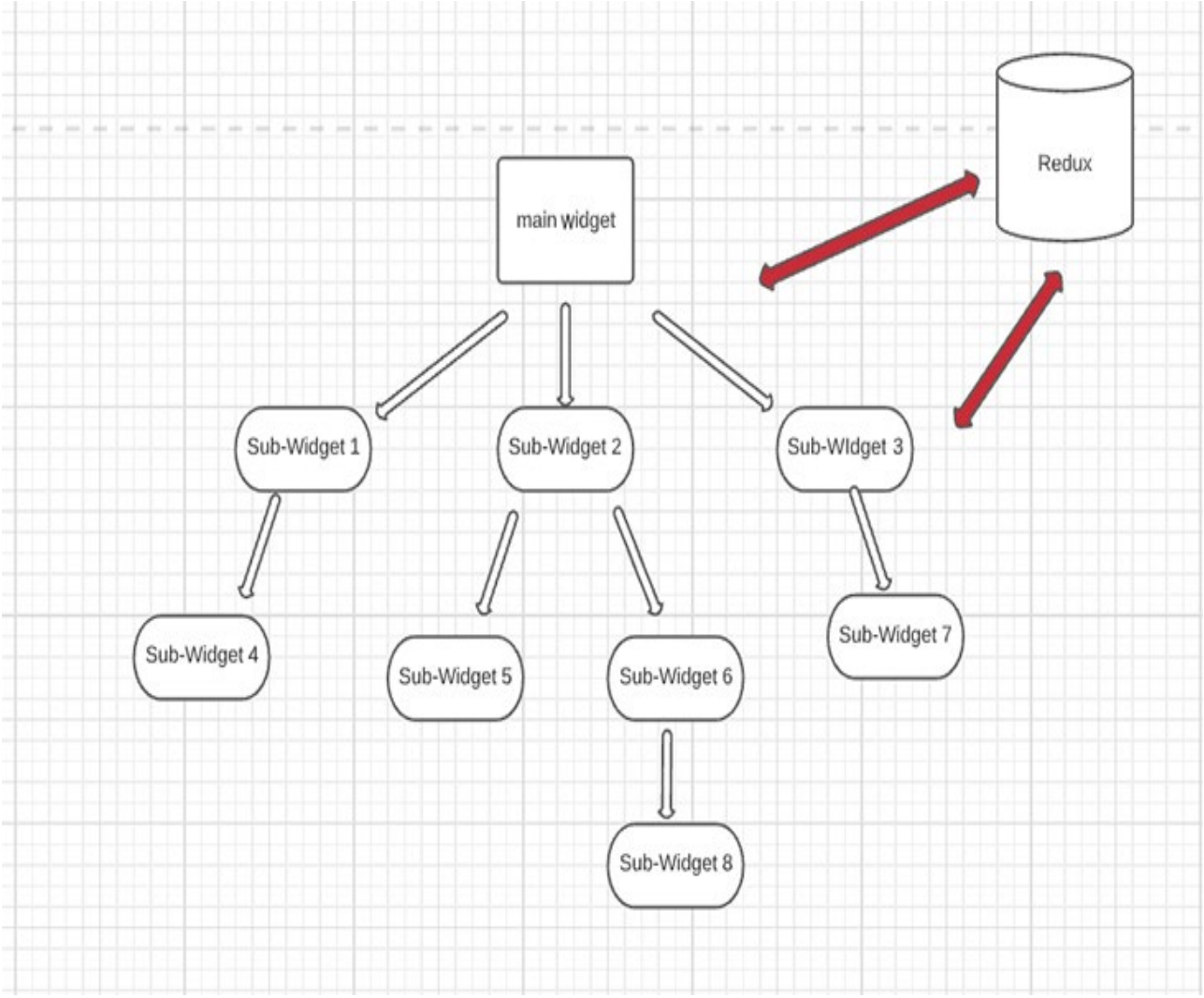

Рис. 2.7 - Взаємодія сховища даних Redux з Flutter віджетами

На рис. 2.7 показано що будь який віджет незалежно від його позиції у дереві віджетів може отримати доступ до сховища даних Redux, хоча це і не бажано.

Будь-який віджет, який потребує додавання, зміни чи отримання даних у стані, керованому сховищем Redux, повинен зробити запит на це з відповідними аргументами.

Так само, для кожної зміни стану, залежні віджети відповідають на зміну або через інтерфейс користувача, або будь-яким іншим налаштованим способом.

З Redux у вашій програмі лише одне сховище інформації. Це не тільки допомагає з налагодженням, але щоразу, коли дані змінюються у вашій програмі, вам легше визначити, де та чому вони змінилися.

Стан вашої програми має бути незмінним, доступ до нього має здійснюватися лише шляхом його читання. Це означає, що якщо ви хочете змінити значення всередині стану, ви повинні повністю замінити стан на новий стан, який містить ваші нові значення.

Це допомагає захистити стан і дозволяє змінювати стан лише за допомогою компоненту, який називається Action. Це також забезпечує прозорість у програмі, оскільки ви завжди можете виявити причину змін у стані та об’єкти, відповідальні за ці зміни.

Зміни стану мають відбуватися лише за допомогою функції. Ці функції, відомі як Reducers, є єдиними сутностями, яким дозволено вносити зміни в стан вашої програми. Кожен Reducer змінює стан певним чином та викликається лише на відповідь на конкретний Action.

2.3.1. Огляд компонентів архітектури Redux

Redux є архітектурою центральною задачею якої є чітке, прозоре зберігання, та маніпулювання станом додатку. Через велику кількість компонентів, та їх специфічні зв’язки краще відійти від класичного поділу на рівень представлення, рівень бізнес логіки, та доменного рівня. Ці зв’язки дещо розподілені між різними компонентами архітектури. Компоненти архітектури Redux краще поділити на такі 3 рівні:

1. Рівень представлення;
2. Рівень Redux;
3. Сервісний рівень;

Store. Це центральний компонент, в якому розташовується стан програми. Store зберігає інформацію про весь стан додатка або будь-який інший окремий стан у кожен момент часу.

Reducer — це сутність, яка повертає новий об’єкт стану на основі Action. Reducer може поєднуйте багато інших Reducer, щоб змінювати менші компоненти стану.

State — це сутність, яка зберігає поточний стан сеансу програми. Це єдине сховище всіх завантажених з інтернету даних, встановлених користувацькими фільтрами, введеними запитами тощо. State є єдиним джерелом правди додатка. State — це незмінний об’єкт, який агрегує менші фрагменти стану про екрани, користувачів та будь які інші дані, сутності які використовуються в додатку. State та будь-який інший менший фрагмент стану можна змінити лише його Reducer. Інтерфейс користувача створюється на основі цього стану. Одна з переваг Redux перед Bloc, це наявність цього компоненту. Він дозволяє гнучким та в той же час простим чином зберігати усі потрібні додатку дані, будь яким зручним для розробника способом. Наприклад, у складному додатку з багатьма запитами до сервера, можна організувати State таким чином, щоб дані про конкретний

екран додатка зберігалися в та мінялися в одному місці, а дані які вигаються з сервера у іншому, та зберігалися у хеш таблиці за ключем ID та значенням, конкретна конкретної сутності. Тобто за потреби State можна розділити на два менших стани: Relational і Modules.

Relational — це просте сховище даних, у якому збираються завантажені дані з парами ключ-значення, які мають використовуватися багатьма частинами програми. Ключ – це ідентифікатор збережених даних, а значення – це самі дані.

Modules. Більш динамічна частина стану, яка зазвичай представляє якийсь екран або функцію. Він містить дані про фільтри користувачів, запити, ідентифікатори даних, які зберігаються в реляційному стані тощо.

Action – це будь-яка подія, що сталася в додатку. Action передається до Store, а потім передається всім Middleware і Reducer. Action може бути надіслана як інтерфейсом користувача, так і Middleware, повідомляючи, наприклад, що завантаження даних завершено або не вдалося.

Middleware є частиною Redux, яка обробляє всі побічні ефекти. Middleware є мостом між рівнями Redux і Service. Middleware приймає в обробку Action, та в залежності від нього звертається до Service, і на основі відповіді Service може відправляти до Store інший Action, щоб вказати, що завантаження даних завершено, не вдалося чи щось інше.

Сервісний рівень керує роботою з серверною частиною, локальним сховищем або будь-якими іншими сторонніми бібліотеками. Це можуть бути прості класи які виконують операції по доступу до сервера, бази даних, тощо. Можуть бути класи які відповідають за навігацію у додатку. Загалом дуже простий рівень в цілому не відрізняючийся від доменного рівня додатку з BLoC архітектурою.

Далі слідує огляд рівня представлення архітектури Redux.

StoreProvider – InheritedWidget, який будує та зберігає Store. Завжди є одним із найвищих віджетів усього додатка.

StoreConnector є мостом між Redux та UI. StoreConnector — це віджет Flutter, який отримує доступ до Store завдяки StoreProvider. StoreConnector будує ViewModel на основі поточного State, створює своє дерево віджетів, частіше за все представляючих один екран додатка і передає йому ViewModel.

ViewModel — це опис інтерфейсу користувача. Створюється StoreConnector на основі State. ViewModel описує стан інтерфейсу користувача та містить необхідні дані, наприклад список покупок на головному екрані додатка-магазина. Також ViewModel містить функції для елементів інтерфейсу користувача, таких як кнопка, при виклику яких Store отримує Action.

ViewModel Має відповідати таким критеріям:

- Бути незмінним об’єктом із з превизначеними equals та hashCode. Flutter оптимізує перебудову інтерфейсу користувача за допомогою цих функцій.
- Опишіть UI таким чином, коли два стани, що виключають один одного, неможливі. Наприклад, ViewModel не може бути помилкою і isLoading одночасно.

StoreConnector спроможний оптимізувати перебудову свого дерева віджетів на основі ViewModel. Якщо при порівнянні нової та старої ViewModel StoreConnector отримує true, він не буде перебудовувати своє дерево віджетів, так як у цьому частіше за все немає ніякого сенсу, тому що ViewModel дає повний опис того, як змінні частини інтерфейсу користувача мають виглядати та що робити у відповідь на дії користувача. Беручи до уваги цей факт, та знаючи про особливості об’єктів мови програмування Dart, які представляють функції. А саме що два об’єкти функцій при порівнянні завжди повертають false. Можна додати ще один простий компонент до рівня презентації – Command. Command це простий клас-обгортка над функцією з визначеними функціями equals та hashCode. Цей

компонент дозволить дуже просто використати оптимізацію інтерфейсу користувача.

2.3.2. Огляд взаємозв’язку компонентів архітектури Redux

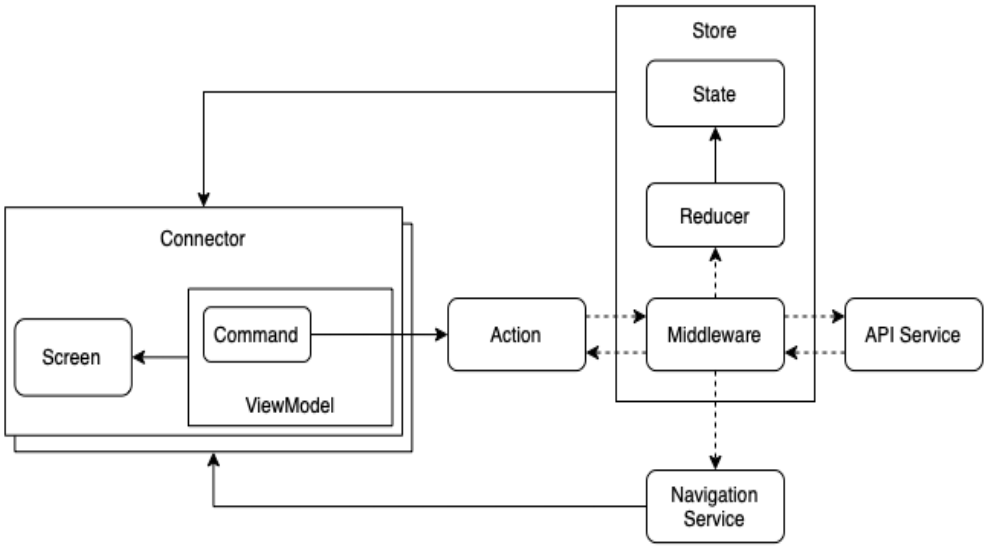

Рис. 2.8 - Схема архітектури Redux

На рис. 2.8 приведена схема взаємодії архітектурних компонентів Redux. Рівень представлення реалізовується за допомогою комбінації Flutter віджетів та пакету “flutter_redux”, який відкриває доступ до віджетів StoreProvider та StoreConnector.

Рівень Redux реалізовується за допомогою пакету “redux”, та в цілому не потребує фреймворку Flutter, що в теорії дозволяє використовувати цю архітектуру у будь якому додатку написаному на мові програмування Dart. Але рівень презентації буде виглядати дещо інакше.

Сервісний рівень це чисті класи Dart, у яких реалізовується доступ до того чи іншого сховища даних, або виконується інша взаємодія зі сторонніми API.

2.4. Проміжні висновки

На прикладі описаних у першому розділі принципів побудови гарної архітектури та досліджених у розділі 2 архітектурних підходів проведемо аналіз впливу обраної чи побудованої архітектури на складність розробки зважаючи на сильні та слабкі сторони того чи іншого архітектурного рішення. Їх впливу на швидкість та вартість розробки. На складність долучення нових членів команди. На здатність додатка масштабуватись.

РОЗДІЛ 3. АНАЛІЗ ВПЛИВУ ОБРАНОЇ АРХІТЕКТУРИ НА ЖИТТЄВИЙ ЦИКЛ ПРОДУКТУ

Розглянувши поняття архітектури у розробці програмного забезпечення, дослідивши принципи на яких базується добре побудована архітектура мобільного додатку, а саме:

- Розділення відповідальності;
- Керування інтерфейсом користувача за допомогою моделей даних;
- Єдине джерело правди;
- Односпрямований потік даних.

А також дослідивши дві архітектури мобільних додатків для фреймворку Flutter, а саме:

- BLoC;
- Redux.

Розглянемо вплив обраної архітектури на процес розробки та підтримки мобільного додатку. Проведемо оцінку цього впливу на життєвий цикл додатку з точки зору розробки та з точки зору бізнесу.

Розглянемо як архітектура може допомогти у досягненні цілей встановлених у першому розділі:

- Підвищити якість продукту;
- Зменшити вартість розробки та підтримки;
- Полегшити роботу команди розробників;
- Зменшити поріг входження нової команди розробників чи окремих її членів.

Це допоможе продукту бути життєздатним впродовж довгого часу, полегшить його підтримку, масштабування та удосконалення, що в свою чергу допоможе продукту бути ефективним з точки зору бізнесу.

3.1. Вплив обраної архітектури на розробку та підтримку продукту

Розробку мобільного додатка можна умовно поділити на декілька частин:

- Написання каркасу додатка;
- Розробка функціоналу;
- Підтримка.

У межах першого пункту робиться вибір на користь тієї чи іншої архітектури, написання її кодової бази. Інтеграція у цю архітектуру потрібних залежностей від сторонніх сервісів, підключення локального сховища даних, чи сервера.

На другому етапі, по заданому шаблону ведеться розробка узгодженого функціоналу. Для прикладу розглянемо екран логування у додаток. Створюються компоненти кожного рівня для цього функціоналу, а саме:

- Компоненти для відображення інтерфейсу користувача;
- Компоненти відповідальні за бізнес логіку;
- Компоненти відповідальні за роботу зі сховищами даних.

Від обраної архітектури буде залежати кількість цих компонентів, межі їх відповідальності, тощо. Проведемо аналіз кількості компонентів потрібних для реалізації поставленої задачі для архітектур BLoC та Redux, розглянем об’єм роботи для реалізації задачі з використанням кожної з архітектур та подивимося як кожна з архітектур покаже себе з часом.

3.1.1. Аналіз життєвого циклу функції логування у додаток з використанням архітектури BLoC

Перше що нам знадобиться це компонент для відображення інтерфейсу користувача. З використанням фреймворку Flutter описується інтерфейс користувача для логування за допомогою e-mail та паролю.

Друге, це компоненти для роботи з даними. Компонент для виконання запитів на сервер, який зможе отримати запит на логування з e-mail та паролем, та повернути токен користувача або помилку. Компонент для локального збереження токена користувача у зашифрованій пам’яті телефону.

Третє, компонент відповідальний за бізнес-логіку екрану, назвемо його LoginCubit. Це буде компонент архітектури BLoC який приймає події з інтерфейсу користувача через функції, та віддає поточний стан екрану через потік даних.

Цей компонент буде отримувати від користувача e-mail, пароль, зберігати їх у своїх змінних, які будуть представляти стан додатка. Із цим e-mail-ом та паролем робити запит до сервера на логування користувача, установлювати поточний стан додатка у завантаження, на основі відповіді від сервера, або зберігати токен користувача та відправляти його на головний екран додатку, або установлювати поточний стан додатка у помилку. Та передавати цей стан до компоненту інтерфейсу користувача.

Виглядає досить простою. Зрозуміло чим займається кожний з компонентів, та межі їх відповідальності. Невелика кількість коду у кожному з компонентів та невелика кількість цих компонентів робить таке рішення досить привабливим. Але є декілька деталей які слід брати до уваги, щоб не опинитись у ситуації коли це рішення вже не буде привабливим, а стане проблемою. Про це далі у розділі 3.1.3.

3.1.2. Аналіз життєвого циклу функції логування у додаток з використанням архітектури Redux

Перші три компоненти(інтерфейс користувача, компонент для запитів до сервера, та компонент для збереження токена локально) будуть мало чим відрізнятись від аналогічних компонентів у архітектурі BLoC.

Далі, для реалізації того ж самого функціоналу нам знадобляться такі компоненти:

- State для зберігання стану додатка (введені e-mail та пароль, наявність помилок, індикатор завантаження);
- Reducer для зміни першого компоненту;
- Middleware для виконання потрібних запитів до сервера та локального сховища;
- Connector для перетворення стану додатка у опис інтерфейсу користувача;
- Набір компонентів Actions для тригеру подій у додатку, таких як виконання запиту до сервера, чи відповідь з сервера.

Користувач введе e-mail та пароль, тим самим змінив компонент State, натисне кнопку Login, тим самим давши завдання компоненту Reducer на встановлення стану додатку у завантаження, та компоненту Middleware на виконання запиту. Виконавши запит, компонент Middleware за допомогою Action змусить компонент Reducer змінити поточний з стану завантаження у стан успіху або помилки.

Виглядає дещо складно для однієї кнопки та двох елементів для вводу даних. Така декомпозиція і розділення відповідальностей виглядає занадто, і , мабуть, так і є. Але не завжди.

3.1.3. Порівняння підходів BLoC та Redux для реалізації екрану логування

Підхід BLoC виглядає простіше та зрозуміліше компоненти поділені на рівні, їх легко читати та писати. Здається що треба обирати BLoC для зменшення кількості так званого boilerplate коду, присутнього у Redux.

Але якщо представити ситуацію, що через деякий час замовник захоче додати логування через номер телефону, сервіси Google та Apple, і ще декілька соціальних мереж, декомпозиція архітектури Redux вже не виглядає надлишковою. Стан зберігається окремо, змінюється окремо. Запити виконуються також окремо. Тепер ми чітко знаємо де що шукати, та де що писати.

З підходом BLoC усі ці операції будуть зберігатися разом. Одна строка коду встановлює стан додатку у завантаження, а вже наступна іде до компоненту для запиту на сервер. І так декілька разів на один компонент бізнес-логіки. Його об’єм виросте, код буде виглядати намішано та занадто складно.

Отже можна зробити висновок, що якщо у додатку є перспектива розвитку та розширення, з потребою постійно масштабуватись, то мабуть, слід віддати перевагу більш складній архітектурі, яка на перших етапах може виглядати як перебір.

Але якщо додаток планується невеликим, з конкретною ціллю та без перспектив або планів розвитку та масштабування кодової бази, слід віддати перевагу менш складній архітектурі.

3.2. Вплив обраної архітектури на продукт з точки зору бізнесу

Беручи до уваги висновки зроблені у попередньому розділі, проаналізуємо вплив обраної архітектури з точки зору бізнесу. Правильна

оцінка продукту та його перспектив, добре продумані плани на майбутнє слід обирати архітектуру в залежності від наявності перспектив чи планів для розвитку та масштабування. Добре обрана архітектура заощадить час на розробку та підтримку продукту.

ВИСНОВКИ

У ході виконання дипломної роботи було розглянуто поняття архітектури програмного забезпечення, проведено аналіз впливу наявності архітектури на здатність продукту масштабуватись. Проведено аналіз впливу архітектури на вартість та складність розробки та підтримки мобільних додатків.

Виконано дослідження архітектур мобільних додатків написаних з використанням фреймворку Flutter. Досліджено архітектури BLoC та Redux. Оцінена складність кожної з цих архітектур, виявлені їх сильні та слабкі сторони.

На основі досліджених архітектур проведено аналіз впливу архітектури на життєздатність продукту, його складність та вартість реалізації. Та описано критерії та принципи за якими слід обирати архітектуру.

СПИСОК ЛІТЕРАТУРИ

1. Роберт Мартін. Чиста архітектура: Мистецтво розроблення програмного забезпечення. «Ранок», Фабула. 2020. с. 368. ISBN 978-617-09-5286-8.
2. Фрэнк Заметти. Flutter на практике. ДМК Пресс. с. 98. ISBN: 9785970608081.
3. List of state management approaches. [Електронний ресурс]. – Режим доступу: https://docs.flutter.dev/development/data-and-backend/state-mgmt/options.
4. Bloc State Management Library. [Електронний ресурс]. – Режим доступу: https://bloclibrary.dev/#/gettingstarted.
5. How to manage States in Flutter using Redux. [Електронний ресурс]. – Режим доступу: https://medium.com/@rajeswari3699/flutter-redux-state-management-with-redux-in-flutter-7a6a13515f69.
6. Марк Р. Ричардс. Fundamentals of Software Architecture. O'Reilly Media, Inc. с. 122. ISBN: 9781492043454.
7. Chris Buckett. Dart in Action. Manning. с. 384. ISBN: 978-1617290862.
8. Роберт Мартін. Чистий код: Створення і рефакторинг за допомогою agile. «Ранок», Фабула. 2020. с. 448. ISBN 978-617-09-5285-1.

ДОДАТОК А

Лістінг програми прикладу використання Inherited віджетів

```
import 'package:flutter/material.dart';

class AppDataProvider  extends InheritedWidget {
  final AppData appData;
  final Widget child;
  const AppDataProvider({Key? key, required this.appData, required this.child}): super(key:
key, child: child);

  static AppDataProvider? of (BuildContext context) =>
context.dependOnInheritedWidgetOfExactType<AppDataProvider>();

  @override
  bool updateShouldNotify(covariant AppDataProvider oldWidget) {
    return true;
  }
}

class AppData {
  int count;
  Color backgroundColor;
  AppData({required this.count, required this.backgroundColor});

  incrementCount() {
    count++;
  }
  changeBackgroundColor (Color bgColor) {
    backgroundColor = bgColor;
  }
}

void main() {
```

```
  runApp(
    AppDataProvider(appData: AppData(count: 5, backgroundColor: Colors.black) , child:
MyApp() )
  );
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Inherited Widget',
      home: HomePage() ,
    );
  }
}

class HomePage extends StatefulWidget {
  @override
  _HomePageState createState() => _HomePageState();
}

class _HomePageState extends State<HomePage> {
  @override
  Widget build(BuildContext context) {
    AppDataProvider? appDataProvider = AppDataProvider.of(context);
    return Scaffold(
      body: SafeArea(
        child: Container(
          height: MediaQuery.of(context).size.height,
          color: appDataProvider?.appData.backgroundColor,
          child: Center(
            child: Column(
              children: [
                Text( appDataProvider!.appData.count.toString(), style: TextStyle(
```

```
                color: Colors.white, fontSize: 24
              ),  ),
              ElevatedButton(onPressed: () {
                setState(() {
                  appDataProvider.appData.incrementCount();
                });
              } ,
                child: Text('Increment')),
              ElevatedButton(onPressed: () {
                setState(() {
                  appDataProvider.appData.changeBackgroundColor(Colors.purple);
                });
              } ,
                child: Text('Change Color'))
            ],
          ),
        ),
      ),
    ) ,
  ) ;
 }
}
```