Національний університет «Полтавська політехніка імені Юрія Кондратюка»
(повне найменування вищого навчального закладу)
Навчально-науковий інститут інформаційних технологій і робототехніки
(повна назва факультету)
Кафедра комп'ютерних та інформаційних технологій і систем
(повне найменування вищого навчального закладу)

Пояснювальна записка
до диломного проекту (роботи)
магістра
(освітньо-кваліфікаційний рівень)
на тему

Розробка кросплатформного додатку для реалізації функціоналу task-management

Виконав: студент 6 курсу, групи 602-ТН
спеціальності
122 Комп'ютерні науки
(шифр і назва напряму)
Важничий Є.В.
(прізвище та ініціали)
Керівник Фесенко Т.Г.
(прізвище та ініціали)
Рецензент Дорошенко С.М.
(прізвище та ініціали)

Полтава – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
«ПОЛТАВСЬКА ПОЛІТЕХНІКА ІМЕНІ ЮРІЯ КОНДРАТЮКА»

НАВЧАЛЬНО НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ І РОБОТОТЕХНІКИ

КАФЕДРА КОМП'ЮТЕРНИХ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ І СИСТЕМ

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

спеціальність 122 «Комп'ютерні науки»

на тему

«Розробка кросплатформного додатку для реалізації функціоналу task-management»

Студента групи 602-ТН Важничого Єгора Валентиновича

Керівник роботи
кандидат технічних наук,
д.т.н., проф. Фесенко Т.Г.

Консультант
кандидат технічних наук,
доцент Скакаліна О.В.

Завідувач кафедри
кандидат фізико-математичних наук,
Двірна О.А.

Полтава – 2025

РЕФЕРАТ

Кваліфікаційна робота магістра: 58 с., 54 малюнків, 2 додатка, 15 джерел.

Об’єкт дослідження: системи менеджменту персональних задач.

Мета роботи: розроблення крос-платформного додатку для реалізації функціоналу task-management з мінімалістичним та інтуїтивним інтерфейсом.

Методи: проектування та розробка крос-платформного додатку, у вигляді системи, що надає можливість управління персональними задачами у вигляді списку поділеного на виконані, сьогоднішні та майбутні таски.

Ключові слова: крос-платформний додаток, task-management, мінімалізм, список задач.

ABSTRACT

Master's qualification work: 58 p., 54 images, 2 appendices, 15 sources.

Object of research: personal task management systems.

Purpose of work: development of a cross-platform application for implementing task-management functionality with a minimalist and intuitive interface.

Methods: design and development of a cross-platform application in the form of a system that provides the ability to manage personal tasks in the form of a list divided into completed, current and future tasks.

Keywords: cross-platform application, task-management, minimalism, task list.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ 6
ВСТУП 7
РОЗДІЛ 1 АНАЛІТИЧНИЙ ОГЛЯД TASK-MANAGER СИСТЕМ 9
1.1 Аналіз предметної області 9
1.1.1 Загальні відомості 9
1.1.2 Історія становлення та еволюції task-manager систем? 9
1.2 Аналіз конкурентів 12
1.2.1 Things3 12
1.2.2 Notion 13
1.2.3 Obsidian 14
1.2.4 Minimalist 15
1.2.5 Jira 16
1.2.6 Trello 17
1.3 Висновки щодо вимог до сучасного task-manager 17
РОЗДІЛ 2 ПРОЕКТУВАННЯ ДОДАТКУ 19
2.1 Аналіз технологій для реалізації проекту 19
2.1.1 Flutter. 19
2.1.2 Riverpod. 21
2.1.3 Freezed. 24
2.1.4 Isar. 25
2.2 Структура проекту 29
2.3 Проєктування моделей даних 31
РОЗДІЛ 3 РОЗРОБКА ДОДАТКУ 32
3.1 Концепт та особливості додатку 32
3.2 Структура додатку 35
3.3 Кодова база додатку 36
3.3.1 Presentation 37
3.3.2 Domain 48
3.3.3 Data 49
РОЗДІЛ 4 ПЛАН ПОДАЛЬШОГО РОЗВИТКУ ДОДАТКУ 52

ВИСНОВКИ 54
ДЖЕРЕЛА 55
ДОДАТОК А ВЕРСТКА СПИСКІВ ЗАДАЧ ДОДАТКУ 56
ДОДАТОК Б КОНФІГУРАЦІЯ БД ДОДАТКУ 58

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

HTTP – HyperText Transfer Protocol

HTTPS – HyperText Transfer Protocol Secure

SQL – Structured Query Language

NoSQL – non-SQL

UI – User Interface

UX – User Experience

API – Application Programming Interface

Frontend – клієнтська частина додатку

Backend – серверна частина додатку

ПЗ – програмне забезпечення

ОС – операційна система

БД – база даних

СУБД – Система управління базами даних

ПЗ – програмне забезпечення

DAO – Data Access Object

CRUD – Create Read Update Delete

ВСТУП

Що таке кросплатформенний додаток?

Кросплатформенний додаток - це програма, яка працює на декількох операційних системах одночасно. Завдання програміста - написати код, який добре працює на всіх операційних системах.

Універсальний підхід до розробки відповідає двом важливим вимогам: економія часу та грошей. Розробники можуть створювати додатки швидше. Точніше, сам додаток розробляється за той самий час, що й додаток для iPhone чи Android. Але якщо компанії замовляють додатки для різних систем, час розробки подвоюється, витрачається більше грошей.

Але є й недоліки. Крос-платформні додатки не такі гнучкі, як нативні. Магазини додатків мають свої вимоги, які необхідно враховувати під час розробки. Це створює додаткові незручності та труднощі.

Незважаючи на це, крос-платформні додатки дуже популярні та ефективні. Залежно від вашого бізнесу, ви можете створювати унікальні інструменти, з якими користувачі зможуть взаємодіяти. Швидкий старт, широке охоплення користувачів та відносно низькі витрати на розробку дозволяють швидко реалізовувати свої ідеї, пропонувати ринку потужні інструменти та оцифровувати свій бізнес.

Відомими прикладами є React Native, Flutter та Ionic.

Відмінності між нативною та кросплатформною розробкою додатків.

Нативні додатки - це додатки, розроблені для конкретної операційної системи. Вони використовують відповідний стек технологій для вирішення конкретного завдання.

Серед переваг - висока продуктивність, максимальне використання всіх можливостей платформи та хороший користувацький інтерфейс. Такі додатки найкраще демонструвати на воркшопах. Недоліки: створення нативних додатків займає багато часу і значно збільшує витрати. Оскільки вони не підтримуються

в інших операційних системах, додатки для цих систем доводиться розробляти окремо, а це дорого і довго.

Отже, які основні відмінності між нативною та кросплатформною розробкою додатків? Необхідно заздалегідь розуміти, яку функцію додаток буде виконувати в бізнесі, хто його цільові користувачі і чому його потрібно розробляти в першу чергу. Якщо у вас є час і бюджет, вам не потрібно використовувати дві платформи одночасно, пріоритет слід віддати нативним додаткам. Якщо додаток простий і функціональний, а основна увага приділяється ясності завдання і мети, перевага надається крос-платформному рішенню.

Крос-платформна розробка додатків - найкраще рішення для бізнес-завдань. Якщо основна увага приділяється функціональності, а не візуальному дизайну, такий підхід до розробки може заощадити багато часу, скоротити бюджет і створити ефективні додатки, які служать бізнесу [1].

РОЗДІЛ 1
АНАЛІТИЧНИЙ ОГЛЯД TASK-MANAGER СИСТЕМ

1.1Аналіз предметної області

1.1.1 Загальні відомості. Task-manager системи є інструментами, які допомагають структурувати робочий процес і підвищувати продуктивність. Вони дозволяють створювати завдання, встановлювати дедлайни, визначати пріоритети та відповідальних осіб. Завдяки інтегрованим функціям комунікації та обміну файлами, ці системи спрощують співпрацю в командах. Багато з них пропонують автоматизацію процесів, як-от нагадування або шаблони для повторюваних завдань. Прозорість робочих процесів дає можливість керівникам ефективніше контролювати прогрес. Водночас, велика кількість функцій може ускладнити використання для новачків, а надмірна залежність від технологій може стати перешкодою в разі технічних проблем. Завдяки масштабованості, ці системи підходять як для індивідуальних користувачів, так і для великих організацій. Вибір підходящої платформи залежить від потреб користувача, рівня інтеграції з іншими інструментами та бюджету. Хоча деякі рішення можуть бути перевантажені функціоналом, правильно обрана система здатна значно оптимізувати робочі процеси. Загалом, task-manager системи є важливим інструментом у сучасному світі для управління часом і ресурсами.

1.1.2 Історія становлення та еволюції task-manager систем?

Історія становлення та еволюції task-manager систем пов'язана зі зростанням потреби людей у структурованому управлінні своїм часом. Витоки цих інструментів можна знайти ще у рукописних планувальниках, які почали активно використовуватись в XIX столітті. Найпростішою формою були записники та блокноти, що слугували для запису завдань, зустрічей і нотаток. Згодом ці записники еволюціонували в органайзери, такі як Filofax у 1920-х роках, що мали поділ на секції для завдань, контактів і календаря.

Ідея візуалізації задач отримала новий поштовх у 1980-х роках із появою методології Bullet Journal, розробленої для ручного ведення списків. Цей підхід дозволив користувачам створювати систематизовані записи з індексацією, іконками для позначення статусів і структурованими списками завдань. Bullet Journal став популярним серед тих, хто прагнув контролювати свої справи в аналоговій формі, пропонуючи гнучкість у налаштуванні.

З розвитком цифрових технологій у 1990-х роках відбулося перше значне зрушення у сфері управління завданнями. Поява персональних комп'ютерів і програмного забезпечення дозволила створювати електронні органайзери. Одними з перших цифрових додатків стали Lotus Notes та Microsoft Outlook, які інтегрували завдання, календарі та електронну пошту в одну систему. Хоча ці інструменти були орієнтовані на бізнес-користувачів, вони заклали основу для більш сучасних і гнучких рішень.

У 2000-х роках, з розвитком інтернету, почали з'являтися хмарні сервіси управління завданнями. Одним із піонерів у цій сфері стала система Basecamp, запущена в 2004 році, яка пропонувала можливості для командної роботи й організації проектів. Її поява стала поворотним моментом, оскільки це був один із перших інструментів, який дозволяв віддаленим командам співпрацювати в реальному часі. Паралельно з цим з'явилися більш прості інструменти для індивідуального використання, такі як Remember The Milk (2004), що спеціалізувався на списках справ.

Справжній вибух популярності task-manager систем стався у 2010-х роках, коли мобільні технології та смартфони стали невід'ємною частиною повсякденного життя. Програми, такі як Todoist і Wunderlist, запропонували зручність у використанні завдяки кросплатформенності, можливості синхронізації між пристроями та мінімалістичному дизайну. Завдяки цьому користувачі отримали можливість керувати своїми завданнями будь-де і будь-коли.

Інновації в дизайні та функціональності привели до появи більш складних систем, орієнтованих на команди й великі проекти. Наприклад, Trello, запущений

у 2011 році, запропонував унікальний підхід до візуалізації завдань у вигляді Kanban-дошок. Цей метод дав змогу легко організовувати завдання за статусами ("Виконується", "Очікує", "Завершено") й отримав визнання у сфері проектного менеджменту. Інший приклад – Asana, яка з 2008 року почала пропонувати можливості для управління завданнями з акцентом на командну роботу.

З поширенням методологій управління, таких як Agile та Scrum, task-manager системи стали інтегрувати нові функції. Сучасні інструменти, такі як ClickUp, Monday.com і Jira, включають діаграми Ганта, часові лінії, Kanban-дошки й автоматизацію завдань. Ці системи орієнтовані на бізнес-середовище, але залишаються доступними для персонального використання.

На сьогодні найпопулярніші task-manager системи – це універсальні інструменти, які можуть адаптуватися до будь-яких потреб. Серед таких систем можна виділити Notion, який об'єднує функції управління завданнями, ведення нотаток і організації даних. Іншою популярною платформою є Microsoft To Do, яка завдяки простоті інтегрується в екосистему Microsoft 365. Ще одним прикладом є Google Tasks, вбудований у Google Workspace, що спрощує роботу з Gmail і Календарем.

Еволюція task-manager систем не закінчується. Сьогодні активно розробляються системи, які використовують штучний інтелект для автоматизації задач і прогнозування пріоритетів. Наприклад, деякі сучасні інструменти можуть самостійно аналізувати графіки користувача й пропонувати оптимальний розклад завдань. У майбутньому можна очікувати ще більшої персоналізації та інтеграції цих систем у повсякденне життя, зокрема через голосових асистентів і технології доповненої реальності.

Таким чином, task-manager системи пройшли довгий шлях від простих записників до складних цифрових екосистем. Їх популярність зумовлена тим, що вони відповідають сучасним вимогам ефективності, гнучкості та простоти. У майбутньому ці інструменти продовжать розвиватися, інтегруючи нові технології для покращення організації часу і ресурсів.

1.2 Аналіз конкурентів

1.2.1 Things3

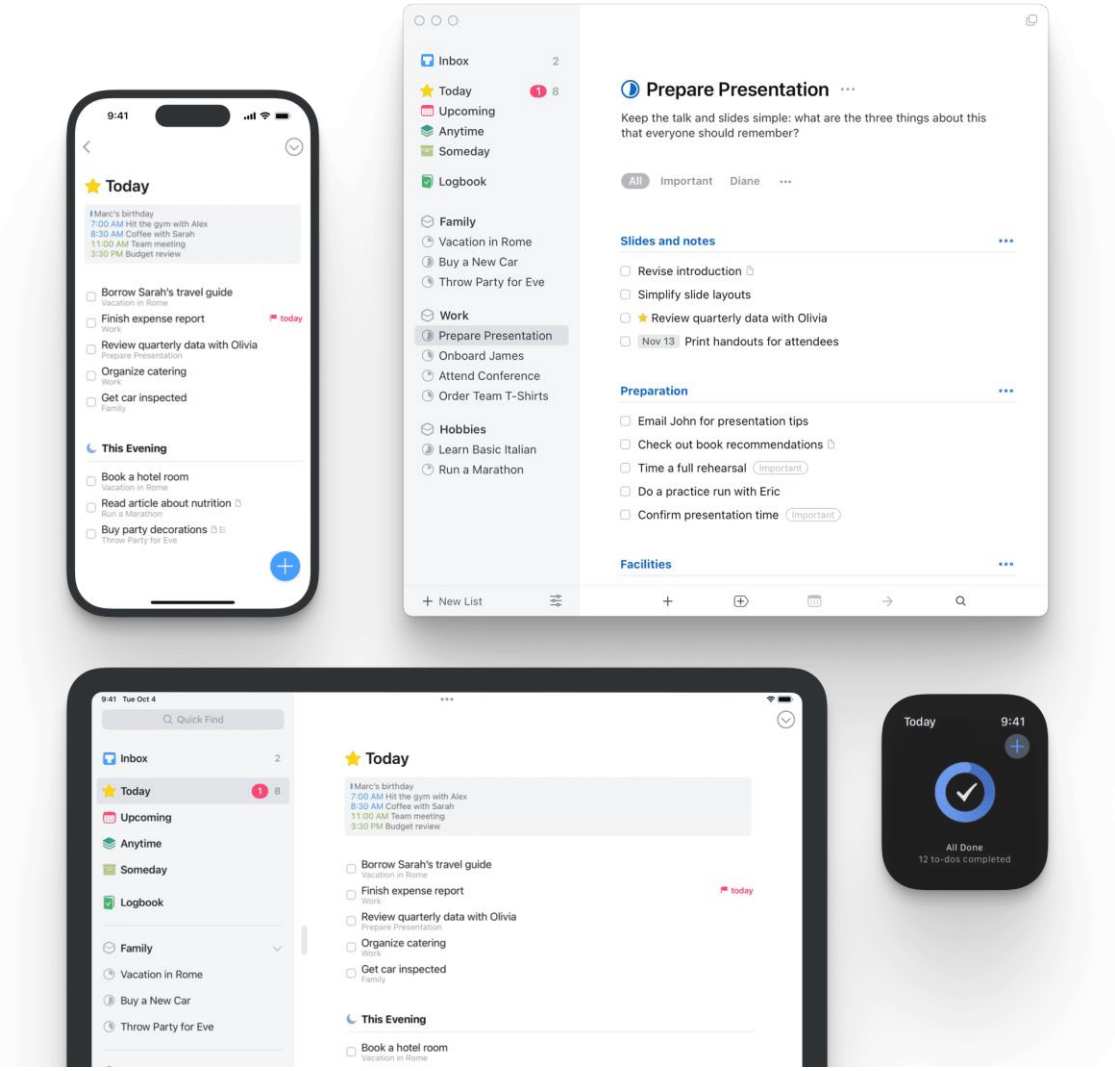


Рисунок 1.1 –інтерфейс додатку Things3 на різних платформах

Things3 відзначається інтуїтивним дизайном і зручним користувацьким інтерфейсом, орієнтованим на користувачів Apple. Його ключова перевага – простота у використанні, що поєднується з потужною функціональністю. Things3 забезпечує глибоку інтеграцію з екосистемою Apple (iPhone, iPad, Mac), включаючи підтримку функцій Siri, віджетів і швидких дій. Основна особливість – логічно побудована система проєктів, тегів та нагадувань, яка дозволяє легко структурувати завдання. Відсутність зайвих функцій робить його ідеальним для

індивідуальних користувачів, які цінують ефективність. Недолік – відсутність кросплатформенності, обмежуючи аудиторію користувачами Apple.

1.2.2 Notion

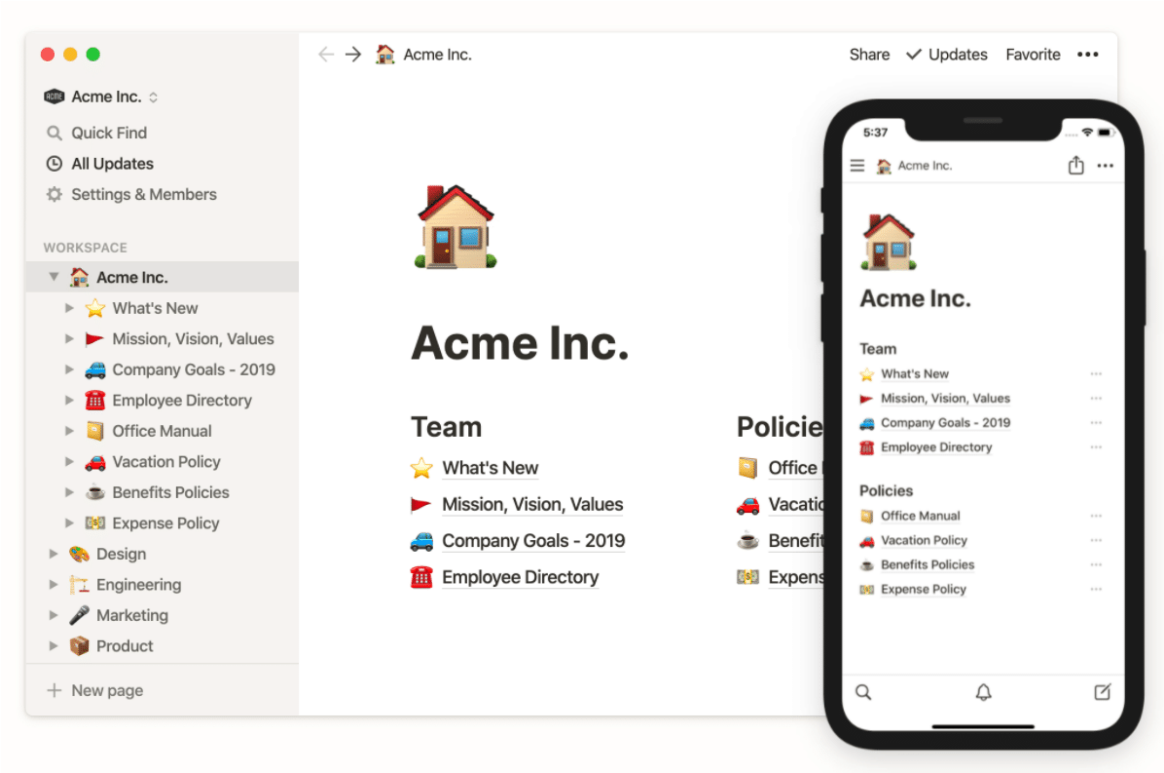


Рисунок 1.2 –інтерфейс додатку Notion на різних платформах

Notion – універсальний інструмент, який поєднує функції управління завданнями, ведення нотаток і баз даних. Основною конкурентною перевагою є його гнучкість – користувачі можуть налаштовувати робочий простір відповідно до своїх потреб. Завдяки кросплатформенності Notion підходить для особистого використання та командної роботи. Серед сильних сторін – вбудовані шаблони для швидкого старту, підтримка інтеграцій і потужні можливості для візуалізації даних. Однак велика кількість функцій може викликати труднощі у новачків, що потребує часу для навчання.

1.2.3 Obsidian

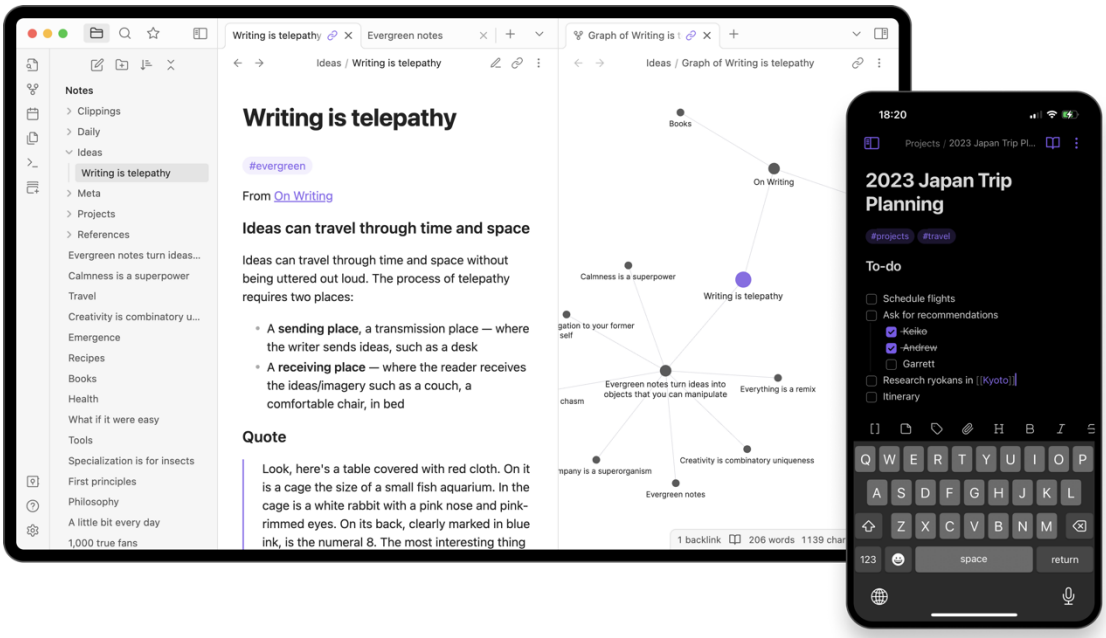


Рисунок 1.3 –інтерфейс додатку Obsidian на різних платформах

Obsidian – це додаток для ведення нотаток, орієнтований на користувачів, які цінують організацію інформації через зв'язки між записами. Його конкурентна перевага – функція графового подання, що дозволяє візуалізувати взаємозв'язки між нотатками. Обсидіан орієнтований на індивідуальну роботу, ідеально підходить для дослідників та письменників. Завдяки локальному збереженню даних користувачі отримують максимальну конфіденційність. Водночас Obsidian не призначений для командної роботи та управління великими проєктами, що обмежує його застосування як повноцінного task-manager.

1.2.4 Minimalist

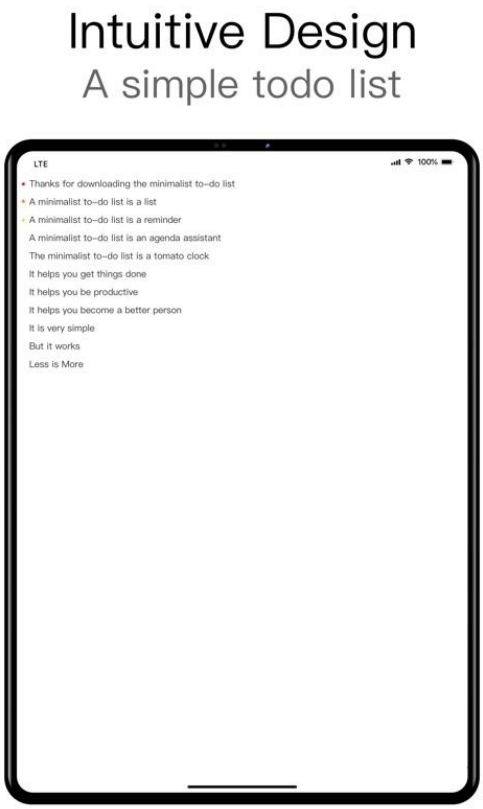


Рисунок 1.4 –інтерфейс додатку Minimalist

Minimalist – простий і стильний додаток, орієнтований на управління особистими завданнями. Його основна перевага – мінімалізм і простота інтерфейсу, які створюють відчуття спокою під час роботи. Додаток фокусується на виконанні основних функцій: створення завдань, встановлення пріоритетів і нагадувань. Його зручність і відсутність зайвих функцій роблять Minimalist привабливим для користувачів, які шукають базовий і естетичний інструмент. Проте відсутність підтримки командної роботи і більш складних функцій обмежує його для особистого використання.

1.2.5 Jira

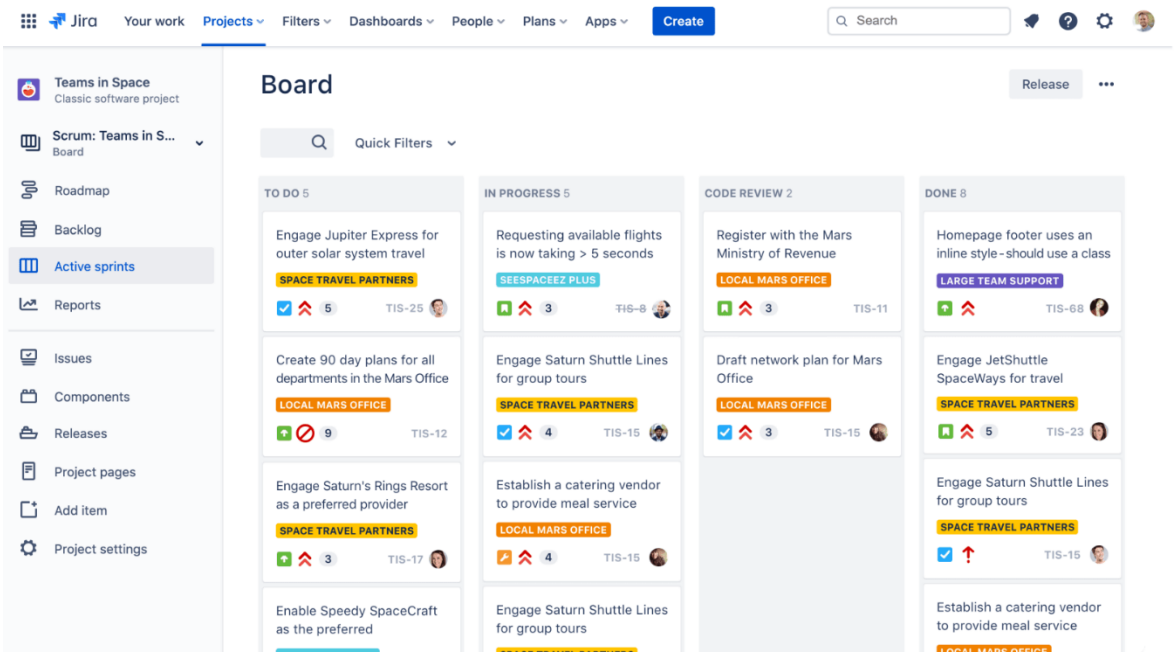


Рисунок 1.5 –інтерфейс додатку Minimalist

Jira – потужний інструмент для управління проєктами, який особливо популярний у сферах програмної розробки та Agile-методологій. Його конкурентна перевага – широкі можливості для налаштування, підтримка Kanban і Scrum-дошок, а також діаграм Ганта. Jira забезпечує детальну аналітику й звітність, що дозволяє відстежувати прогрес проєктів і продуктивність команд. Інтеграція з іншими продуктами Atlassian (Confluence, Bitbucket) додає зручності. Основним недоліком є складність у використанні для новачків і завантаженість інтерфейсу.

1.2.6 Trello

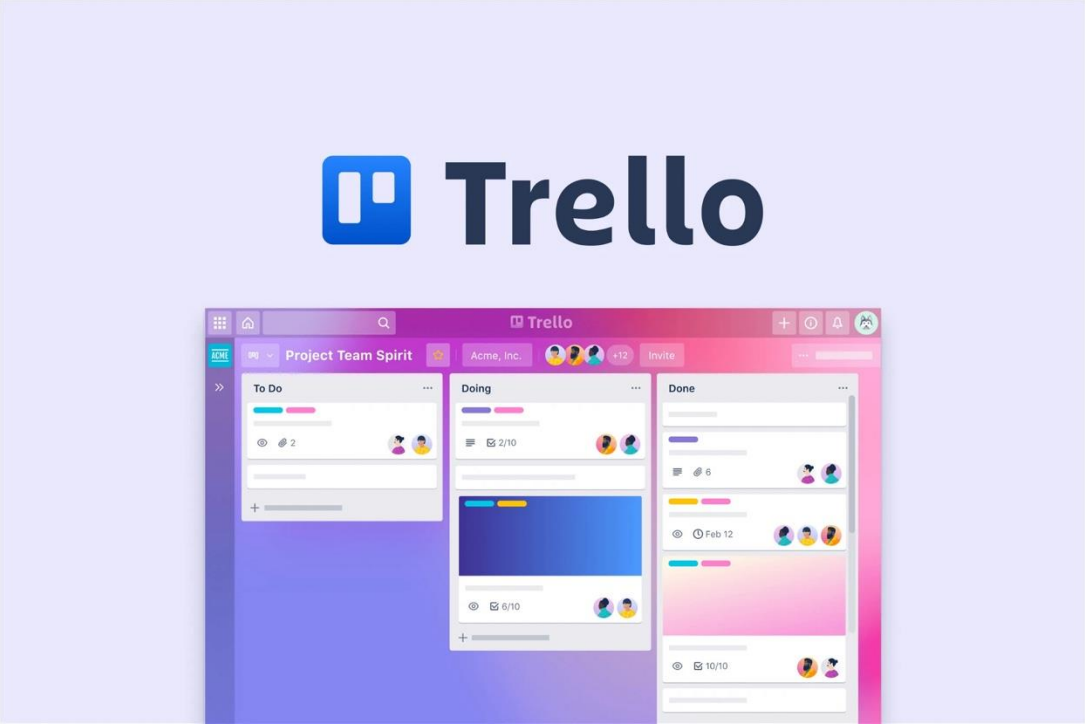


Рисунок 1.6 –інтерфейс додатку Trello

Trello – популярний інструмент для управління завданнями, який виділяється простотою і візуальним підходом. Основна конкурентна перевага – використання Kanban-дошок, що дозволяє швидко організовувати й переміщувати завдання. Trello підходить як для індивідуального, так і для командного використання, пропонуючи інтеграції з різними сервісами (Google Drive, Slack). Завдяки безкоштовному плану додаток став доступним для широкого кола користувачів. Проте обмежені можливості налаштування й аналітики можуть бути недостатніми для великих і складних проєктів.

1.3 Висновки щодо вимог до сучасного task-manager

Сучасні таск-менеджери повинні бути не тільки функціональними, а й мінімалістичними, щоб користувачі могли зосередитися на важливих завданнях без зайвих відволікань. Інтерфейс повинен бути простим і зрозумілим, з чітким акцентом на важливих елементах, таких як створення та управління завданнями,

без надлишкових деталей. Завдання повинні мати можливість встановлення пріоритетів, а також можливість делегування їх серед членів команди.

Механізм розподілу завдань має бути легким та інтуїтивно зрозумілим, що дозволяє швидко призначати відповідальних і встановлювати дедлайни без зайвої складності. Дизайн має бути чистим, із мінімальним використанням кольорів і відсутністю зайвих елементів, що забезпечує зручне користування навіть на мобільних пристроях. Таск-менеджер повинен підтримувати можливість додавання файлів, коментарів та тегів, але ці функції мають бути розміщені у найпростішому вигляді, щоб не перевантажувати користувача. Календар і сповіщення повинні бути візуально лаконічними, не забираючи багато простору на екрані, але в той же час достатньо помітними, щоб користувач завжди був у курсі змін.

Кожен елемент інтерфейсу повинен бути логічно впорядкований, забезпечуючи ефективну роботу без зайвого натискання на кнопки чи перемикання між екранами. Крім того, важливо, щоб додаток мав гарний, але простий дизайн, який не тільки виглядає естетично, але й підтримує швидку навігацію та можливість інтеграції з іншими популярними інструментами та платформами.

РОЗДІЛ 2
ПРОЕКТУВАННЯ ДОДАТКУ

2.1 Аналіз технологій для реалізації проекту

Для реалізації проекту використана платформа Flutter, для розробки кросплатформних додатків. Також зазначу найбільш вагомі з використаних на проекті бібліотек(пакетів): Riverpod для стейт менеджменту додатку, Freezed для кодогенерації найчастіших властивостей моделей для класів, Isar для менеджменту локальної БД додатку.

Рисунок 2.1 – Логотип Flutter

2.1.1 Flutter. Flutter - це портативний інструментарій користувацького інтерфейсу від Google для створення гарно візуалізованихдодатків для мобільних, веб та настільних комп'ютерів наоснові єдиної кодової бази.Flutter працює з існуючим кодом і використовується розробниками та організаціями по всьому світу.

Для кого призначений Flutter? Для користувачів, Flutter створює яскраві та красиві додатки. Для розробників Flutter знижує планку для розробки додатків.

Він прискорює розробку додатків і зменшує вартість і складність створення додатків для різних платформ.

Для дизайнерів Flutter забезпечує основу для створення високоякісного користувацького досвіду Fast Company визнала Flutter однією з найкращих дизайнерських ідей десятиліття.

Це пов'язано з його здатністю перетворювати концепції на дієвий код без компромісів, що накладаються типовими фреймворками. Він також слугує ефективним інструментом прототипування для обміну ідеями з іншими завдяки підтримці CodePen.

Для інженерних менеджерів і підприємств Flutter дозволяє розробникам додатків інтегруватися в єдину команду для мобільних, веб- і настільних додатків і створювати фірмові додатки для різних платформ на основі єдиної кодової бази. Flutter прискорює розробку функцій і синхронізує графіки випусків у різних платформ.

Додатки, які потребують створення фірмового дизайну, особливо підходять для Flutter. За допомогою Flutter ви також можете perfect pixel інтерфейси, сумісні з мовами дизайну Android та iOS.

Екосистема пакетів Flutter включає різноманітне обладнання (камери, GPS, мережу, сховище тощо) та сервіси (платежі Flutter - це проект з відкритим вихідним кодом, в який внесли свій вклад компанії та приватні особи, включаючи Google.

Розробники з Google та за межами Google використовують Flutter для створення чудових нативних додатків для iOS та Android. Ось деякі з них. Відвідайте вітрину, щоб побачити деякі з цих додатків.

Flutter відрізняється від багатьох інших варіантів створення мобільних додатків тим, що він не покладається на технологію веб-браузера або набори віджетів, які поставляються з кожним пристроєм. Замість цього Flutter використовує власний високопродуктивний рушій рендерингу для малювання віджетів.

Крім того, Flutter характеризується тонким шаром коду на C/C++: більша частина системи (макет, жести, анімація, обрамлення, віджети тощо) реалізована на Dart (сучасна і лаконічна об'єктно-орієнтована мова), що дозволяє розробникам легко її читати, модифікувати, змінювати або видаляти. Це дає розробникам великий контроль над системою і може значно зменшити бар'єри доступності для більшості систем.

Рисунок 2.2 – Логотип Riverpod

2.1.2 Riverpod. Riverpod — це бібліотека для управління станом у Flutter, яка позиціонує себе як сучасна, гнучка та безпечна альтернатива іншим популярним підходам, таким як Provider, Redux, Bloc тощо. Вона була створена Ремі Русселе, автором Provider, із врахуванням його обмежень та з метою покращення досвіду розробників.

Основні концепції Riverpod

1. Декларативність і функціональність:

Riverpod будується на функціональному підході до управління станом. У ньому всі провайдери є функціями, які описують, як створюється та оновлюється стан. Це дозволяє уникнути конфліктів контекстів, що є проблемою у Provider.

2. Автоматичний доступ до провайдерів:

Riverpod не потребує контексту Flutter для доступу до провайдера. Це означає, що провайдери можна викликати в будь-якому місці коду, навіть поза виджетами, що значно полегшує написання тестів та рефакторинг.

3. Іммутабельність стану:

Riverpod підтримує роботу з незмінними (immutable) об'єктами, що сприяє створенню передбачуваного й надійного коду. Використання інструментів, таких як freezed, дозволяє легко управляти об'єктами стану.

4. Модульність і повторне використання:

Провайдери в Riverpod легко інтегруються між собою, що дозволяє будувати модульні архітектури. Ви можете визначати провайдери для бізнес-логіки окремо від виджетів і використовувати їх повторно.

5. Типобезпечність:

Riverpod гарантує типобезпечність на етапі компіляції, запобігаючи багатьом помилкам, які можуть виникати під час виконання програми.

6. Кешування та оптимізація:

Riverpod автоматично оптимізує оновлення стану, викликаючи зміни тільки для тих провайдерів, які залежать від зміненої частини. Це підвищує продуктивність додатків.

Основні типи провайдерів у Riverpod

1. Provider для створення простого стану або значень.

2. StateProvider: для управління простим станом, який можна змінювати.

3. FutureProvider для роботи з асинхронними операціями, такими як запити до API.

4. StreamProvider: для роботи зі стрімами (потоками даних).

5. StateNotifierProvider: для більш складних сценаріїв управління станом за допомогою класів, які розширюють StateNotifier.

6. ChangeNotifierProvider: для використання з класами, які реалізують ChangeNotifier (подібно до Provider).

Порівняння Riverpod із іншими бібліотеками

1. Riverpod vs Provider:

Provider є основою для Riverpod, однак останній усуває його основні обмеження. Наприклад, у Riverpod доступ до провайдерів не залежить від контексту BuildContext, що робить його більш зручним для тестування. Riverpod також краще оптимізований для складних додатків завдяки своїй модульності та декларативному стилю.

2. Riverpod vs Redux:

Redux використовує концепцію єдиного глобального стану та обробників (reducers) для оновлення цього стану. Хоча це забезпечує передбачуваність, Redux може бути надто складним і багатослівним для невеликих проєктів. Riverpod, натомість, пропонує простіший функціональний підхід, який легше адаптувати до різних масштабів додатків.

3. Riverpod vs Bloc:

Bloc базується на потоках даних (streams) і забезпечує чітке розділення бізнес-логіки та інтерфейсу. Однак це може призводити до зайвої складності в написанні коду. Riverpod, із підтримкою StreamProvider та StateNotifierProvider, дозволяє досягти подібного розділення з меншою кількістю коду та більшою зручністю.

4. Riverpod vs GetX:

GetX — це легка бібліотека для управління станом і навігацією, яка відома своєю простотою. Водночас, GetX іноді критикують за відсутність явної структури коду та меншу типобезпечність. Riverpod забезпечує кращу масштабованість і чистоту коду завдяки декларативному підходу та модульності.

5. Riverpod vs MobX:

MobX забезпечує реактивне управління станом із використанням спостерігачів (observers). Хоча це дозволяє створювати дуже динамічні додатки, MobX може бути важко тестувати та дебажити. Riverpod пропонує більш передбачуваний і декларативний підхід, що спрощує тестування.

Підсумуємо:

Riverpod є потужним, сучасним і зручним інструментом для управління станом у Flutter-додатках. Завдяки функціональному підходу, модульності, типобезпечності та легкій інтеграції з існуючими архітектурними підходами, він виступає як гідна альтернатива іншим бібліотекам. Вибір Riverpod стає особливо обґрунтованим у проєктах, де важливі продуктивність, передбачуваність та масштабованість.

Рисунок 2.3 – Freezed став учасником циклу відео від Google Package of the Week

2.1.3 Freezed. У контексті Flutter, freezed — це популярний пакет для автоматичного створення класів із властивостями, які підтримують іммутабельність, копіювання, серіалізацію та паттерн матчинг. Він значно спрощує роботу з об'єктами даних, які є незмінними (immutable), і усуває необхідність вручну писати багато повторюваного коду.

Основні можливості freezed:

1. Іммутабельність: Класи, створені за допомогою freezed, є незмінними, що відповідає кращим практикам роботи з даними в Flutter/Dart.

2. Зручне копіювання: Пакет автоматично генерує методи copyWith, які дозволяють створювати нові об'єкти на основі існуючих із зміненими полями.

3. Робота з union-класами (sealed класами): Підтримує створення класів із кількома варіантами, що корисно для опису станів (наприклад, у Bloc чи Riverpod).

4. Підтримка порівняння: Генерує методи == і hashCode для коректного порівняння об'єктів.

5. Серіалізація: Має інтеграцію з пакетом json_serializable, що дозволяє автоматично генерувати методи для конвертації об'єктів у JSON і назад.

Isar Database

Super Fast Cross-Platform Database for Flutter

Рисунок 2.4 – Логотип БД Isar

2.1.4 Isar. Isar — це сучасна, високопродуктивна база даних, яка розроблена спеціально для використання у Flutter та Dart. Завдяки своїй продуктивності, простоті використання та багатому функціоналу, Isar є

популярним вибором для локального зберігання даних у мобільних додатках. Цей пакет забезпечує швидке читання та запис великих обсягів даних без значного навантаження на систему.

Основні характеристики Isar:

1. Продуктивність: Isar — це одна з найшвидших баз даних для Flutter. Вона оптимізована для роботи як із невеликими, так і з великими обсягами даних, забезпечуючи швидке виконання запитів і мінімальне споживання пам’яті.

2. Підтримка нативних платформ: Isar працює безпосередньо з рідними платформами (Android, iOS, Windows, macOS, Linux), не використовуючи проміжні шари, як, наприклад, SQLite. Це зменшує час доступу до даних.

3. NoSQL модель: Isar використовує документно-орієнтовану модель, що дозволяє зберігати дані у форматі об’єктів. Це значно спрощує інтеграцію з Dart-кодом, оскільки немає необхідності конвертувати дані в таблиці.

4. Підтримка зв’язків між об’єктами: База даних підтримує як односпрямовані, так і двонаправлені зв’язки, дозволяючи розробникам ефективно моделювати складні структури даних.

5. Асинхронна робота: Isar дозволяє виконувати всі операції з базою даних асинхронно. Це знижує ризик блокування головного потоку додатка та забезпечує плавність роботи.

6. Вбудована підтримка індексів: Індексація полів у Isar автоматизована, що забезпечує швидке виконання складних запитів. Розробники можуть вручну задавати індекси для полів, які часто використовуються у фільтрах.

7. Інтеграція зі структурами Flutter: Isar підтримує реактивне оновлення UI при зміні даних у базі. Завдяки цьому, зміни в базі автоматично відображаються у відповідних виджетах.

Основні переваги Isar:

1. Простота використання: Isar легко налаштовується і має інтуїтивно зрозумілий API.
2. Кросплатформеність: Однаково добре працює на Android, iOS і настільних системах.
3. Типобезпечність: Підтримує Dart'ові типи даних, що забезпечує точність і передбачуваність коду.
4. Мінімальне навантаження на ресурси: Завдяки своєму оптимізованому механізму, Isar ідеально підходить для додатків із високими вимогами до продуктивності.

Порівняння Isar із іншими базами даних

Isar vs SQLite:

- SQLite — це класична реляційна база даних, яка вимагає використання SQL-запитів. Isar, натомість, використовує NoSQL-підхід, що спрощує роботу з даними;
- Isar має вищу продуктивність при читанні та записі великих обсягів даних;
- У SQLite потрібно вручну обробляти зв'язки між таблицями, тоді як Isar підтримує зв'язки на рівні об'єктів.

Isar vs Hive:

- Hive — це також NoSQL база даних для Flutter, але вона поступається Isar у продуктивності та функціональності;
- Isar підтримує складні запити з використанням фільтрів та індексів, чого бракує Hive;
- Hive не підтримує реактивне оновлення UI, у той час як Isar дозволяє автоматично оновлювати інтерфейс при зміні даних.

Isar vs Moor/Drift:

- Drift (раніше Moor) — це обгортка над SQLite, яка надає зручний Dart API для роботи з реляційними базами даних;

- Isar перевершує Drift у продуктивності завдяки прямій роботі з платформою;
- Drift краще підходить для додатків, які потребують складних реляційних операцій, тоді як Isar оптимальний для об’єктно-орієнтованих даних.

Рисунок 2.5 – Порівняння швидкості Isar з конкурентами

Підсумуємо

Isar є потужною, простою та ефективною базою даних для Flutter-додатків, яка ідеально підходить для проєктів, що потребують високої продуктивності та простоти інтеграції. Завдяки своїм особливостям, таким як підтримка зв’язків між об’єктами, індексація та реактивність, Isar забезпечує сучасний підхід до

локального зберігання даних. Порівняно з іншими базами даних, Isar виділяється своєю продуктивністю та зручністю використання, що робить її чудовим вибором для більшості Flutter-додатків.

2.2 Структура проекту

У проекті я прийняв рішення використати орагнізацію папок та файлів додатку у стилі feature-first з розбиттям feature частин відповідно до clean architecture

Feature-first — це підхід до організації проекту, який базується на розділенні коду за функціональними частинами (features), а не за технічними аспектами (наприклад, поділ на модулі для даних, UI чи логіки). Основна ідея полягає в тому, щоб всі частини, необхідні для реалізації певної функціональності, були згруповані в одному місці. Такий підхід забезпечує більшу ізольованість функціональних модулів, спрощує тестування та обслуговування проекту, а також покращує масштабованість.

Переваги feature-first підходу:

1. **Ізоляція компонентів**: Всі елементи, пов'язані з певною функціональністю (наприклад, моделі, віджети, логіка бізнесу), зберігаються в одній папці, що мінімізує залежності між модулями.

2. **Прозорість структури**: Нові розробники можуть швидше зрозуміти структуру проекту, оскільки кожна функція є окремим модулем.

3. **Зручність у масштабуванні**: Легше додавати нові функції, не зачіпаючи існуючий код.

Наприклад, структура проекту, побудована за feature-first підходом, може виглядати так:

Рисунок 2.6 – Зображення прикладу feature-first структури папок проекту з використанням clean architecture

Clean Architecture — це підхід до проектування програмного забезпечення, який спрямований на створення систем із чітким поділом відповідальностей. Головною метою є побудова архітектури, яка є легкою для обслуговування, тестування та розширення. У Flutter Clean Architecture можна реалізувати з використанням state management бібліотеки Riverpod, яка забезпечує декларативне управління станом та високу продуктивність.

Основні принципи Clean Architecture:

1. Поділ на шари: Кожен шар відповідає за певну частину системи. Шари зазвичай поділяються на presentation, domain і data.

2. Залежності спрямовані всередину: Нижчі шари (domain) не повинні залежати від вищих (presentation).

3. Висока тестованість: Завдяки чіткому поділу обов'язків кожен шар можна тестувати окремо.

2.3 Проєктування моделей даних

Для збереження даних тасків було створено просту але ефективну модель даних, яка завдяки кодогенерації Isar та Freezed залишилась доволі лаконічною.

```
part ...

@freezed
@Collection(ignore: {'copyWith', 'toJson', 'fromJson'})
class TaskEntity with _$TaskEntity {
  factory TaskEntity({
    required int id,
    @Default('') String title,
    @Default('') String description,
    @Default(false) bool complete,
    DateTime? updatedAt,
    DateTime? completedAt,
    DateTime? date,
  }) = _TaskEntity;
}

```

Рисунок 2.7 – Зображення моделі для збереження таску

Freezed розширив методи порівняння моделей цього класу, додав можливість копіювання з частковою зміною полей та додав методі парсингу з/у JSON формат.

Isar у свою чергу згенерував колекцію на основі моделі з скріншота вище, що дає можливість взаємодіяти з колекцією через DAO інтерфейс taskEntitys.

РОЗДІЛ 3
РОЗРОБКА ДОДАТКУ

3.1 Концепт та особливості додатку

Основним концептом додатку є мінімалістичний та інтуітивний інтерфейс. Пішовши від початкової концепції списку завдань, пріорітет визначається положенням завдання у списку, вищий пріорітет – вище положення. Хотілось зберегти вигляд єдиного списку але поділ також був потрібен, тому була розроблено наступний вигляд: списки вертикально з’єднані і наче продовжують один одного, навігація між ними йде за допомогою скролу/натискання на панелі управління внизу списку. Порядок списків наступний: виконані завдання, сьогоднішні(також він є стартовим при запуску додатку), майбутні.

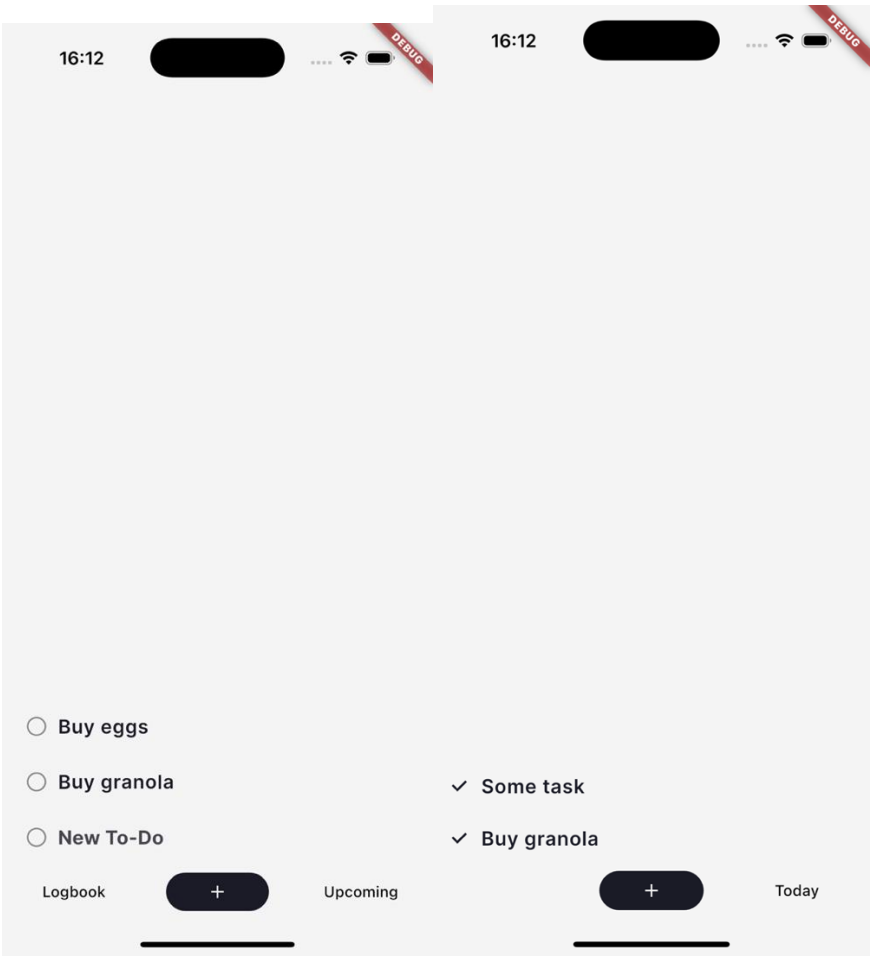


Рисунок 3.1, 3.2 – Список сьогоднішніх/виконаних задач

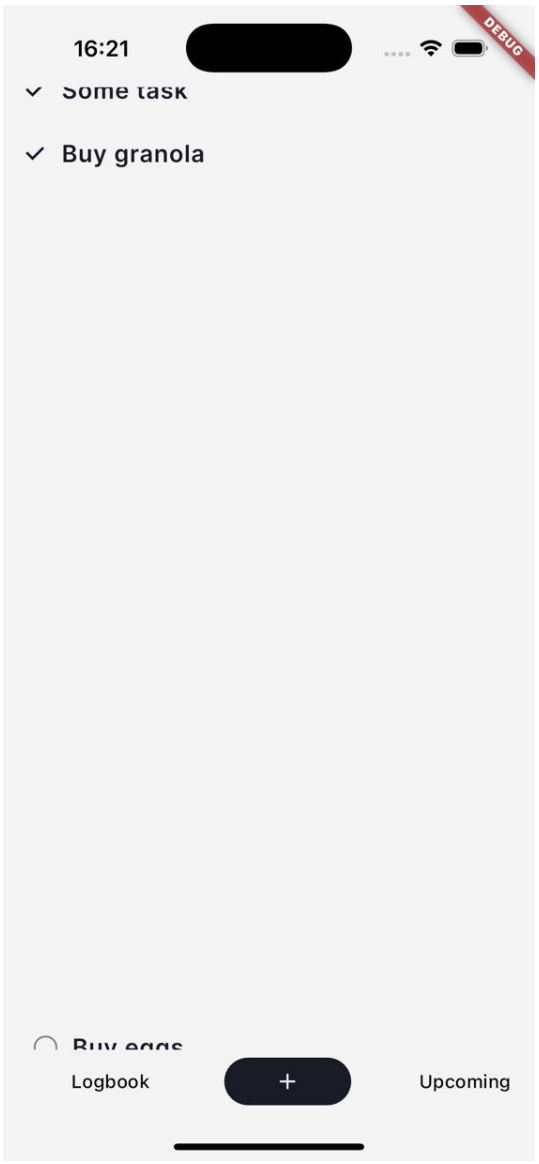


Рисунок 3.3 – Приклад скролу між списками

На малюнку вище можна побачити як списки фактично продовжують один одного під час скролу.

Також у списку приховано видалення та призначення дати виконання таску. Вони доступні через свайп таску вліво та вправо відповідно.

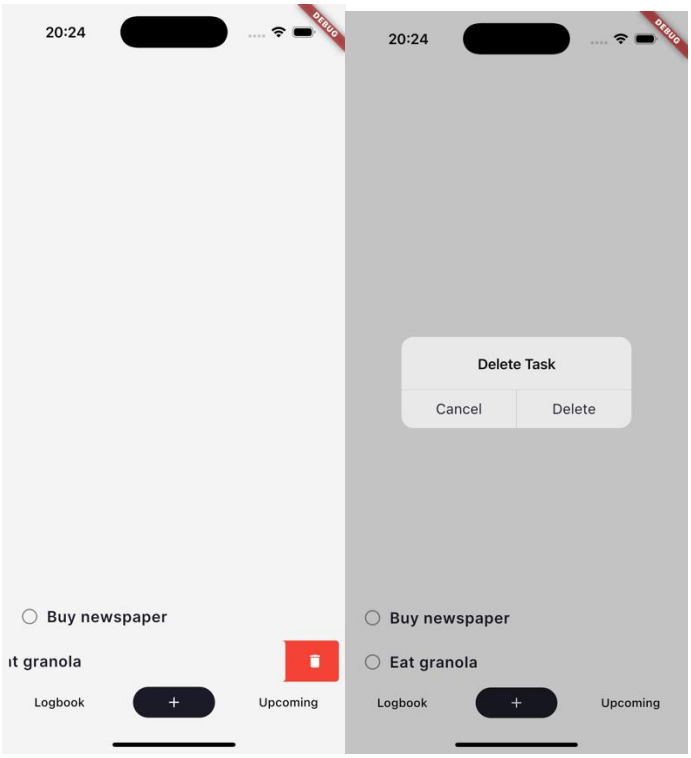


Рисунок 3.4, 3.5 – Видалення таску

Список майбутній тасок ще у розробці тому на ньому поки використовуємо екран-заглушку

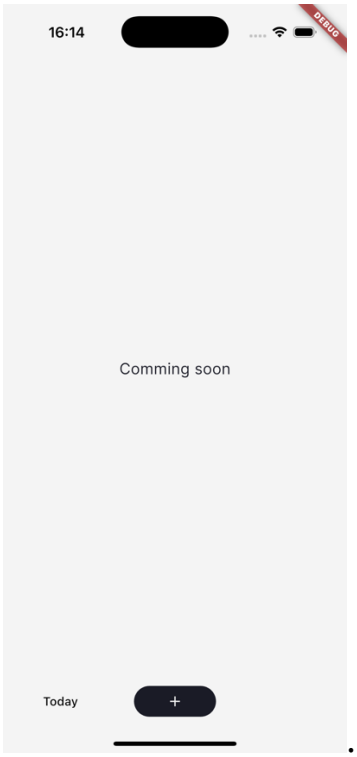


Рисунок 3.6 – Список майбутніх тасків у розробці

3.2 Структура додатку

Фактична структура поточної версії додатку виглядає наступним чином

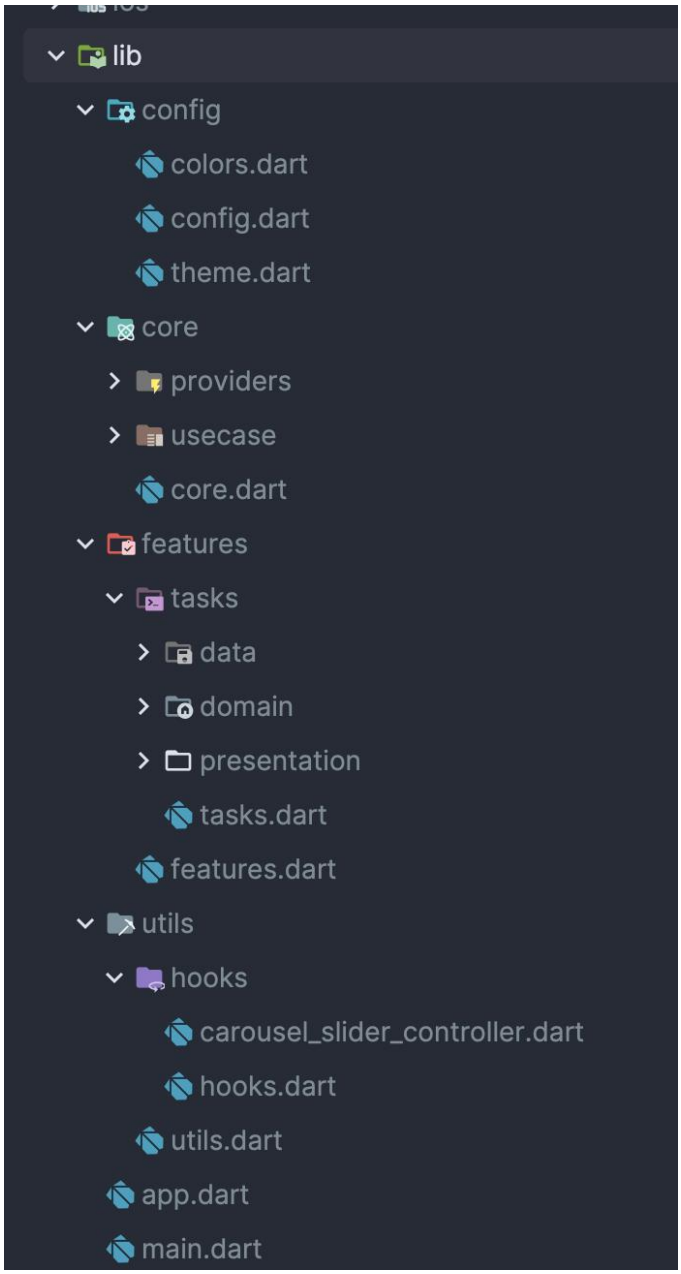


Рисунок 3.7 – Зображення моделі для збереження таску

На верхньому рівні знаходяться наступні директорії:

- config – конфігураційні файли проєкту, теми, кольори і тд;
- core – базові та глобальні сутності проекту. Наприклад provider з конфігурацією БД Icap;
- features – поділені за функціональністю частини/модулі проекту;

- utils – додаткові сервіси, модулі що не залежать від проєкту, але використовуються у ньому.

3.3 Кодова база додатку

Основна кодова магія відбувається у фічі tasks, яку ми і розглянемо у цьому підрозділі. Виходячи з назви фічі тут знаходиться лише код що відноситься до функціоналу тасок. У подальшому також будуть додані нові фічі (наприклад авторизація), але на данний момент tasks, є єдиним представником фіч.

Окремо розглянемо кожну з секцій tasks.

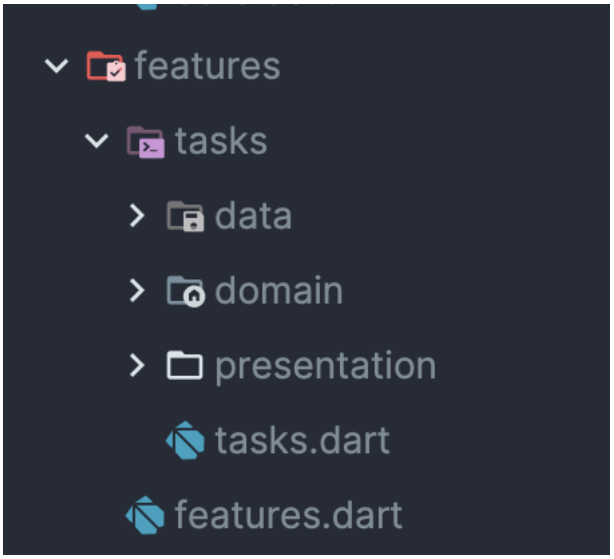


Рисунок 3.8 – Директорія features

3.3.1 Presentation

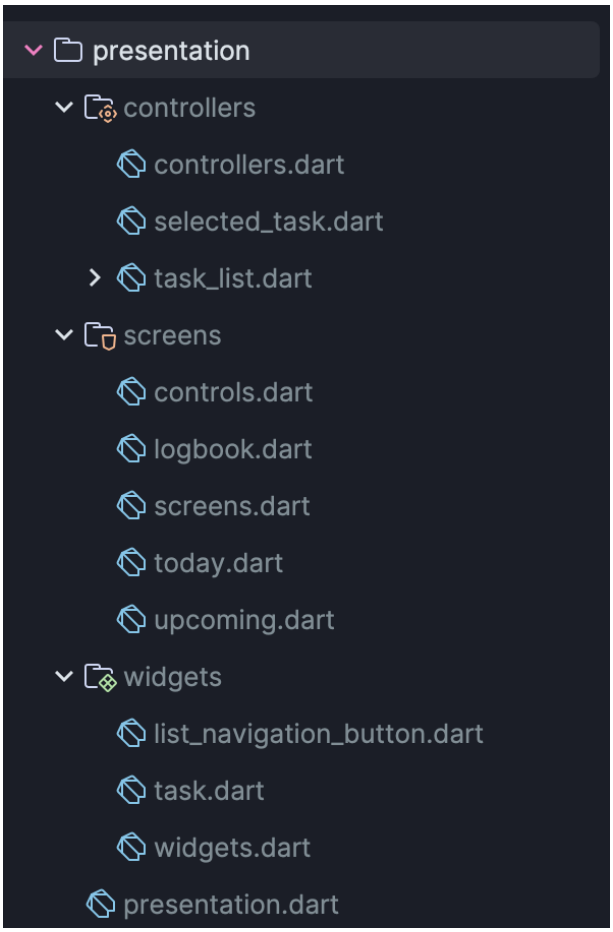
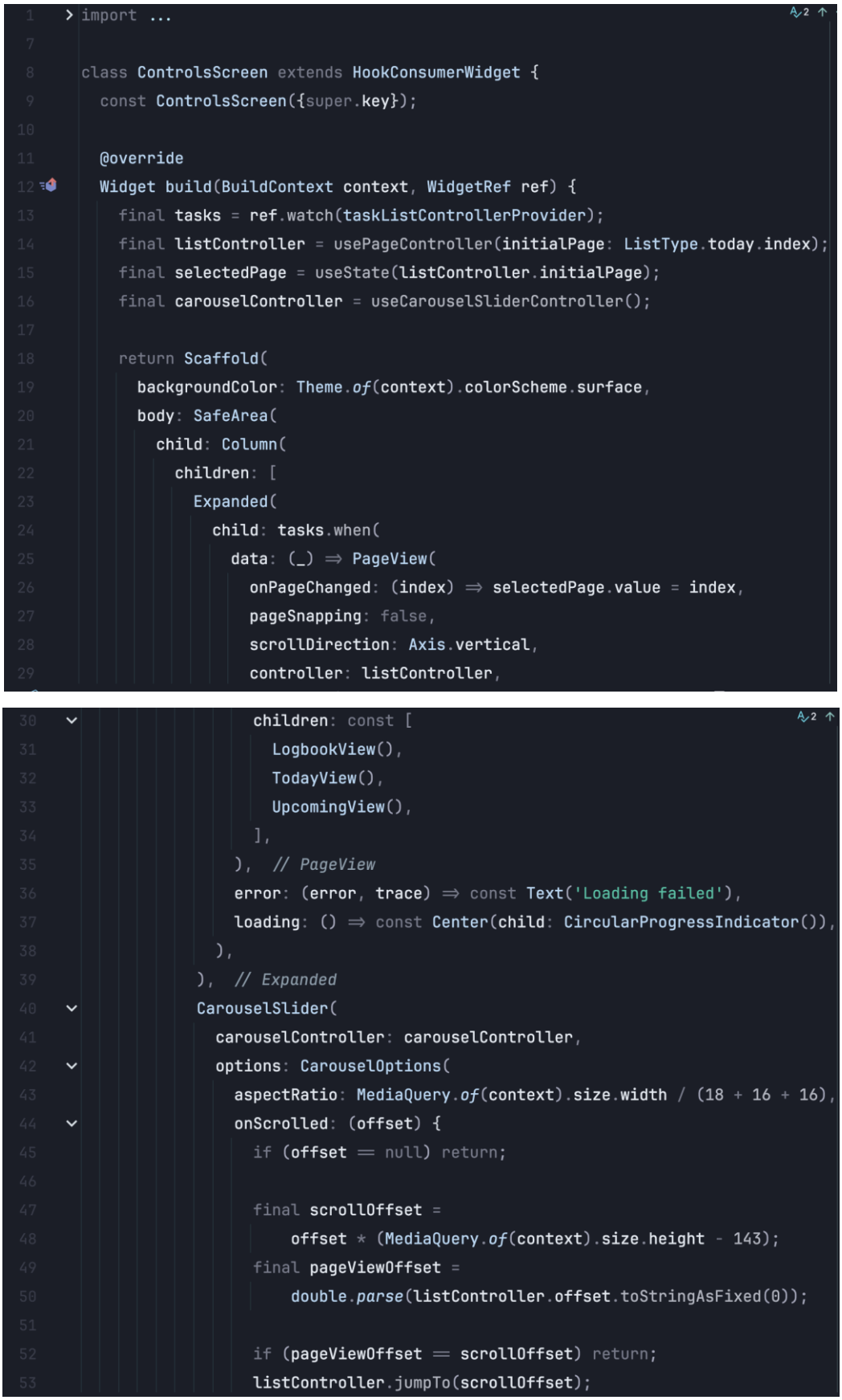


Рисунок 3.9 – Директорія presentation

У цій директорії зберігаються код відповідальний за відображення екранів фічі tasks.

Секції screens зберігає екрани цієї фічі: controls – у даному випадку єдине виключення, бо він є екраном-обгорткою з елементами управління як от навігація; logbook, today, upcoming – відповідно до своїх назв відповідають за списки виконаних, сьогоднішніх та майбутніх задач.



```dart
import ...

class ControlsScreen extends HookConsumerWidget {
  const ControlsScreen({super.key});

  @override
  Widget build(BuildContext context, WidgetRef ref) {
    final tasks = ref.watch(taskListControllerProvider);
    final listController = usePageController(initialPage: ListType.today.index);
    final selectedPage = useState(listController.initialPage);
    final carouselController = useCarouselSliderController();

    return Scaffold(
      backgroundColor: Theme.of(context).colorScheme.surface,
      body: SafeArea(
        child: Column(
          children: [
            Expanded(
              child: tasks.when(
                data: (_) => PageView(
                  onPageChanged: (index) => selectedPage.value = index,
                  pageSnapping: false,
                  scrollDirection: Axis.vertical,
                  controller: listController,
                  children: const [
                    LogbookView(),
                    TodayView(),
                    UpcomingView(),
                  ],
                ),  // PageView
                error: (error, trace) => const Text('Loading failed'),
                loading: () => const Center(child: CircularProgressIndicator()),
              ),
            ),  // Expanded
            CarouselSlider(
              carouselController: carouselController,
              options: CarouselOptions(
                aspectRatio: MediaQuery.of(context).size.width / (18 + 16 + 16),
                onScrolled: (offset) {
                  if (offset == null) return;

                  final scrollOffset =
                      offset * (MediaQuery.of(context).size.height - 143);
                  final pageViewOffset =
                      double.parse(listController.offset.toStringAsFixed(0));

                  if (pageViewOffset == scrollOffset) return;
                  listController.jumpTo(scrollOffset);
```

Рисунок 3.10 3.11 – Controls screen

```
54                  },
55                  viewportFraction: 1 / 3,
56                  enableInfiniteScroll: false,
57                  initialPage: 1,
58                ),  // CarouselOptions
59                items: [
60                  ListNavigationButton(
61                    label: 'Logbook',
62                    selected: selectedPage.value == ListType.logbook.index,
63                    onPressed: () => ref
64                        .read(taskListControllerProvider.notifier)
65                        .addTask(TaskEntity(id: 0, complete: true)),
66                    onSelected: () => carouselController.animateToPage(
67                      ListType.logbook.index,
68                      duration: const Duration(milliseconds: 200),
69                      curve: Curves.easeInOut,
70                    ),
71                  ),  // ListNavigationButton
72                  ListNavigationButton(   Vergo, 08.12.2024, 18:57 • implemented ca
73                    label: 'Today',
74                    selected: selectedPage.value == ListType.today.index,
75                    onPressed: () => ref
76                        .read(taskListControllerProvider.notifier)
77                        .addTask(TaskEntity(id: 0)),
```

```
78                    onSelected: () => carouselController.animateToPage(
79                      ListType.today.index,
80                      duration: const Duration(milliseconds: 200),
81                      curve: Curves.easeInOut,
82                    ),
83                  ),  // ListNavigationButton
84                  ListNavigationButton(
85                    label: 'Upcoming',
86                    selected: selectedPage.value == ListType.upcoming.index,
87                    onPressed: () => ref
88                        .read(taskListControllerProvider.notifier)
89                        .addTask(TaskEntity(id: 0)),
90                    onSelected: () => carouselController.animateToPage(
91                      ListType.upcoming.index,
92                      duration: const Duration(milliseconds: 200),
93                      curve: Curves.easeInOut,
94                    ),
95                  ),  // ListNavigationButton
96                ],
97              )  // CarouselSlider
98            ],
99          ),  // Column
100       ),  // SafeArea
101     );  // Scaffold
```

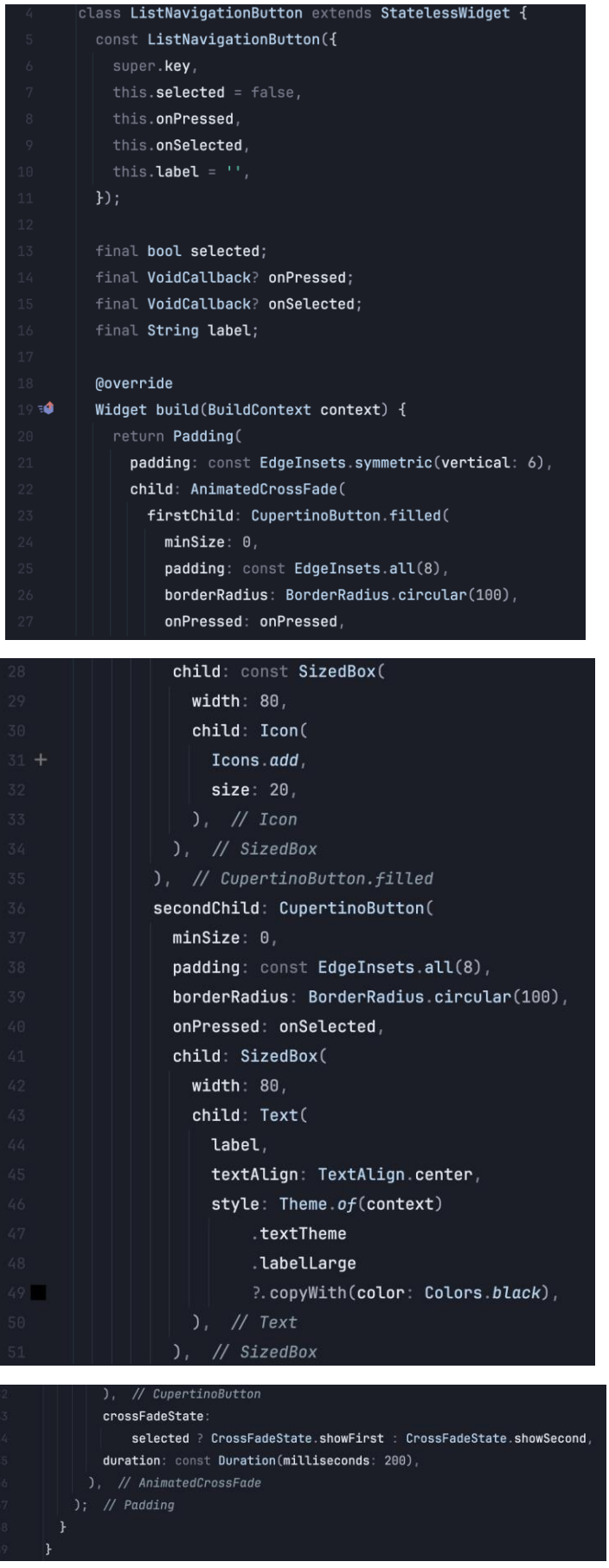
Рисунок 3.12, 3.13 – Controls screen

```dart
import ...

class TodayView extends HookConsumerWidget {
  const TodayView({super.key});

  @override
  Widget build(BuildContext context, WidgetRef ref) {
    final tasks = ref.watch(taskListControllerProvider).value!;

    return ListView.separated(
      padding: const EdgeInsets.symmetric(horizontal: 8),
      reverse: true,
      separatorBuilder: (context, index) => const SizedBox(height: 4),
      itemBuilder: (context, index) => Task(
        tasks.current[index],
        key: Key('task_${tasks.current[index].id}'),
      ),  // Task
      itemCount: tasks.current.length,
    );  // ListView.separated
  }
}
```

Рисунок 3.14 – Верстка списка сьогоднішніх задач

Інші списки схожі на список сьогоднішніх задач і прикріплені у ДОДАТКУ А

У віджетах компоненти, що є будівними блоками цієї фічі. Тут є компоненти що відповідають за кнопки навігації між списками (list_navigation_button), та відображення задач (task). Таск також є виключенням з загальної структури де загалом один віджет на файл. Так як віджет доволі комплексний то він розбитий на Task, DismissibleWrapper(обгортка для виклику модалю видалення через свайп вліво), _DeleteTaskModal (що викликається через парну до нього функцію showDeleteTaskModal), та поки що закоментовану частину з експерементальним описом таску.



```dart
class ListNavigationButton extends StatelessWidget {
  const ListNavigationButton({
    super.key,
    this.selected = false,
    this.onPressed,
    this.onSelected,
    this.label = '',
  });

  final bool selected;
  final VoidCallback? onPressed;
  final VoidCallback? onSelected;
  final String label;

  @override
  Widget build(BuildContext context) {
    return Padding(
      padding: const EdgeInsets.symmetric(vertical: 6),
      child: AnimatedCrossFade(
        firstChild: CupertinoButton.filled(
          minSize: 0,
          padding: const EdgeInsets.all(8),
          borderRadius: BorderRadius.circular(100),
          onPressed: onPressed,
          child: const SizedBox(
            width: 80,
            child: Icon(
              Icons.add,
              size: 20,
            ),  // Icon
          ),  // SizedBox
        ),  // CupertinoButton.filled
        secondChild: CupertinoButton(
          minSize: 0,
          padding: const EdgeInsets.all(8),
          borderRadius: BorderRadius.circular(100),
          onPressed: onSelected,
          child: SizedBox(
            width: 80,
            child: Text(
              label,
              textAlign: TextAlign.center,
              style: Theme.of(context)
                  .textTheme
                  .labelLarge
                  ?.copyWith(color: Colors.black),
            ),  // Text
          ),  // SizedBox
        ),  // CupertinoButton
        crossFadeState:
            selected ? CrossFadeState.showFirst : CrossFadeState.showSecond,
        duration: const Duration(milliseconds: 200),
      ),  // AnimatedCrossFade
    );  // Padding
  }
}
```

Рисунок 3.15-3.17 – ListNavigationButton віджет

```dart
class Task extends HookConsumerWidget {
  const Task(this.task, {this.onComplete, this.margin, super.key});

  final TaskEntity task;
  final VoidCallback? onComplete;
  final EdgeInsets? margin;

  @override
  Widget build(BuildContext context, WidgetRef ref) {
    final titleController = useTextEditingController()..text = task.title
    final descriptionController = useTextEditingController()
      ..text = task.description;
    final ticker = useSingleTickerProvider();
    final fadeController = useAnimationController(
      initialValue: 1,
      vsync: ticker,
      duration: const Duration(milliseconds: 300),
    );
    final fadeAnimation = CurvedAnimation(
      parent: fadeController,
      curve: Curves.easeIn,
    );
    final complete = useState(task.complete);

    onChecked(bool? value) {
      if (value == null) return;

      complete.value = value;
      final fadeDuration = fadeController.duration!.inMilliseconds;
      Future.delayed(
        Duration(milliseconds: 1000 - fadeDuration),
        () {
          if (complete.value != task.complete) {
            fadeController.reverse();
            Future.delayed(
              Duration(milliseconds: fadeDuration),
              () => ref.read(taskListControllerProvider.notifier).updateTask(
                    task.copyWith(complete: complete.value),
                  ),
            ); // Future.delayed
          }
        },
      ); // Future.delayed
    }

    return SizeTransition(
      sizeFactor: fadeAnimation,
      child: Padding(
```

Рисунок 3.18, 3.19 – Task віджет

```
padding: margin ?? EdgeInsets.zero,
child: FadeTransition(
  opacity: fadeAnimation,
  child: Container(
    decoration: const BoxDecoration(
      color: Colors.red,
      borderRadius: BorderRadius.all(Radius.circular(4)),
    ),  // BoxDecoration
    child: DismissibleWrapper(
      key: Key('${task.id}-dismissible'),
      onDelete: () => showDeleteTaskModal(
        context,
        onDelete: (deleted) {
          if (deleted == false) return;
          ref
              .read(taskListControllerProvider.notifier)
              .removeTask(task);
        },
      ),
      child: Container(
        decoration: BoxDecoration(
          borderRadius: BorderRadius.circular(4),
          color: Theme.of(context).colorScheme.primary,
        ),  // BoxDecoration
        padding: const EdgeInsets.symmetric(horizontal: 8),
        child: Column(
          children: [
            Row(
              children: [
                Checkbox(
                  value: complete.value,
                  onChanged: onChecked,
                ),  // Checkbox
                const SizedBox(width: 4),
                Expanded(
                  child: TextFormField(
                    decoration: const InputDecoration(
                      border: InputBorder.none,
                      hintText: 'New To-Do',
                    ),  // InputDecoration
                    controller: titleController,
                    onChanged: (title) {
                      ref
                          .read(taskListControllerProvider.notifier)
                          .updateTask(
                            task.copyWith(title: title),
                          );
                    },
                    onTapOutside: (_) =>
                        FocusManager.instance.primaryFocus?.unfocus(),
                  ),  // TextFormField
                ),  // Expanded
```

Рисунок 3.20-3.22 – Task віджет

```dart
class DismissibleWrapper extends HookWidget {
  const DismissibleWrapper({
    required this.child,
    required Key key,
    this.onDelete,
  })  : taskKey = key,
        super(key: key);

  final Widget child;
  final Key taskKey;
  final VoidCallback? onDelete;

  @override
  Widget build(BuildContext context) {
    final actionTriggered = useState(false);

    final offsetToAction = 56 / MediaQuery.of(context).size.width;

    return Container(
      decoration: const BoxDecoration(
        color: Colors.red,
        borderRadius: BorderRadius.all(Radius.circular(4)),
      ),  // BoxDecoration
      child: Dismissible(
        onUpdate: (details) {
          final triggerAction = details.progress > offsetToAction;
          if (triggerAction != actionTriggered.value) {
            HapticFeedback.lightImpact();
            actionTriggered.value = triggerAction;
          }
        },  Vergo, 11.01.2025, 18:59 • Fixes and improvements
        confirmDismiss: (dismissed) async {
          print(dismissed);
          onDelete?.call();
          return false;
        },
        direction: DismissDirection.endToStart,
        background: const Icon(Icons.delete),
        secondaryBackground: Row(
          children: [
            const Spacer(),
            SizedBox(
                width: 56,
                child: Icon(
                  Icons.delete,
                  size: actionTriggered.value ? 20 : 16,
                  color: Colors.white,
                )),  // Icon, SizedBox
          ],
        ),  // Row
        key: taskKey,
        dismissThresholds: {DismissDirection.endToStart: offsetToAction},
        child: child,
      ),  // Dismissible
    );  // Container
  }
}
```

Рисунок 3.23-3.25 – DismissibleWrapper віджет

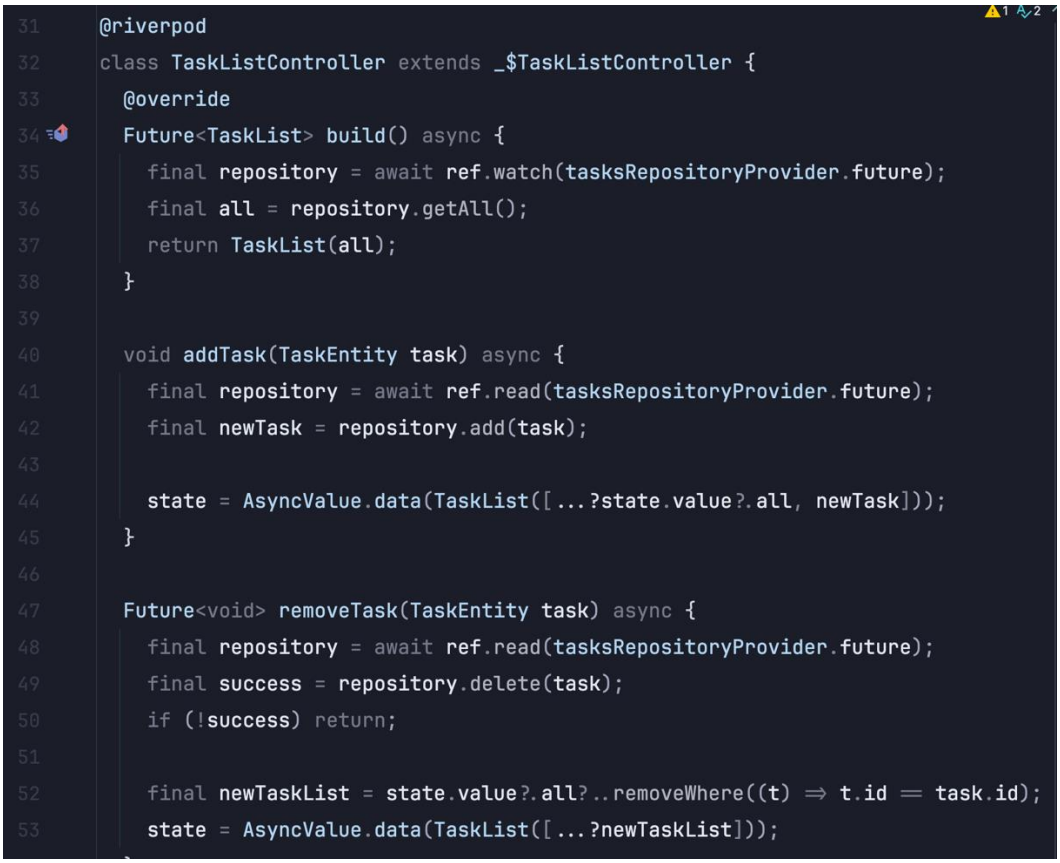
```dart
showDeleteTaskModal(BuildContext context,
    {required void Function(bool deleted) onDelete}) {
  return showCupertinoDialog(
      context: context, builder: (_) => _DeleteTaskModal(onDelete));
}

class _DeleteTaskModal extends StatelessWidget {
  const _DeleteTaskModal(this.onDelete);

  final void Function(bool deleted) onDelete;

  @override
  Widget build(BuildContext context) {
    return CupertinoAlertDialog(
      title: const Text('Delete Task'),
      actions: [
        CupertinoDialogAction(
          onPressed: () {
            Navigator.of(context).pop();
            onDelete(false);
          },
          child: const Text('Cancel'),
        ),  // CupertinoDialogAction
        CupertinoDialogAction(
          onPressed: () {
            Navigator.of(context).pop();
            onDelete(true);
          },
          child: const Text('Delete'),
        ),  // CupertinoDialogAction
      ],
    );  // CupertinoAlertDialog
  }
}
```

Рисунок 3.26, 3.27 – DeleteTaskModal віджет

Остання секція у presentation яку ми не розглянули, це controllers. Вони відповідають за зберігання стану тасків. task_list віповідає за зберігання станів списків (завантаження, успіх, помилка), а також при успішному завантаженні віддає самі списки тасків (Рисунок 3.10). Також через контроллери можна ініціювати дії над даними списку типу додавання/видалення/завантаження/оновлення тасків. selected_task, буде відповідати за збереження поточного активного таску у майбутньому, поки ця частина у розробці.

```dart
class TaskList {
  final List<TaskEntity> all;

  TaskList(this.all);

  List<TaskEntity> get complete =>
      all.where((task) => task.complete).toList().reversed.toList();

  List<TaskEntity> get current =>
      all.where((task) => !task.complete).toList().reversed.toList();

  List<TaskEntity> get upcoming => all.where((task) {
        if (task.date == null) return false;
        return _afterTomorrow(task.date!);
      }).toList();

  bool _afterTomorrow(DateTime date) {
    final now = DateTime.now();
    if (date.year > now.year) return true;
    if (date.month > now.month) return true;
    if (date.day > now.day) return true;
    return false;
  }
}
```

Рисунок 3.28 – дані що віддає task list controller при успішному завантаженні



```dart
@riverpod
class TaskListController extends _$TaskListController {
  @override
  Future<TaskList> build() async {
    final repository = await ref.watch(tasksRepositoryProvider.future);
    final all = repository.getAll();
    return TaskList(all);
  }

  void addTask(TaskEntity task) async {
    final repository = await ref.read(tasksRepositoryProvider.future);
    final newTask = repository.add(task);

    state = AsyncValue.data(TaskList([...?state.value?.all, newTask]));
  }

  Future<void> removeTask(TaskEntity task) async {
    final repository = await ref.read(tasksRepositoryProvider.future);
    final success = repository.delete(task);
    if (!success) return;

    final newTaskList = state.value?.all?..removeWhere((t) => t.id == task.id);
    state = AsyncValue.data(TaskList([...?newTaskList]));
```

```dart
  Future<void> removeTask(TaskEntity task) async {
    final repository = await ref.read(tasksRepositoryProvider.future);
    final success = repository.delete(task);
    if (!success) return;

    final newTaskList = state.value?.all?..removeWhere((t) => t.id == task.id);
    state = AsyncValue.data(TaskList([...?newTaskList]));
  }

  Future<void> updateTask(TaskEntity task) async {
    final repository = await ref.read(tasksRepositoryProvider.future);
    final newTask = repository.update(task);

    final index = state.value?.all.indexWhere((t) => t.id == task.id);
    print(index);
    if (index == null || index < 0) return;

    List<TaskEntity> newTaskList =
        List<TaskEntity>.from(state.value?.all ?? []);
    newTaskList[index] = newTask;

    state = AsyncValue.data(TaskList(newTaskList));
  }
}
```

Рисунок 3.29, 3.30 – TaskListController

3.3.2 Domain

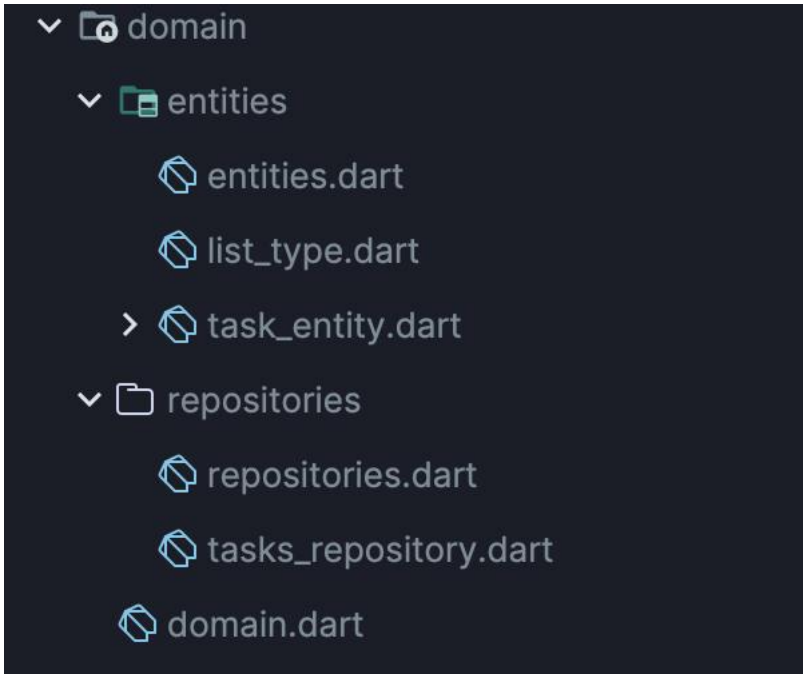


Рисунок 3.31 – Директорія domain

У цій директорії знаходяться файли що зв’язують між собою data та presentation шари наче міст. Entities містять базові сутності тасків такі як enum list_type та основу моделі таску TaskEntity, яку ми вже могли бачити раніше у розділі проектування

```
1  enum ListType {
2    logbook,
3    today,
4    upcoming,
5  }
```

Рисунок 3.32 – ListType enum

Директорія repositories у свою чергу містить репозиторій як абстрактий клас інтерфейс доступу до отримання данних, який зумовлює імплементацію фактичного репозиторія на рівні data.

```dart
import 'package:graphite/features/features.dart';

abstract class TaskRepository {
  TaskRepository();

  List<TaskEntity> getAll();

  TaskEntity add(TaskEntity task);

  TaskEntity update(TaskEntity task);

  bool delete(TaskEntity task);
}
```

Рисунок 3.33 – TaskRepository

3.3.3 Data

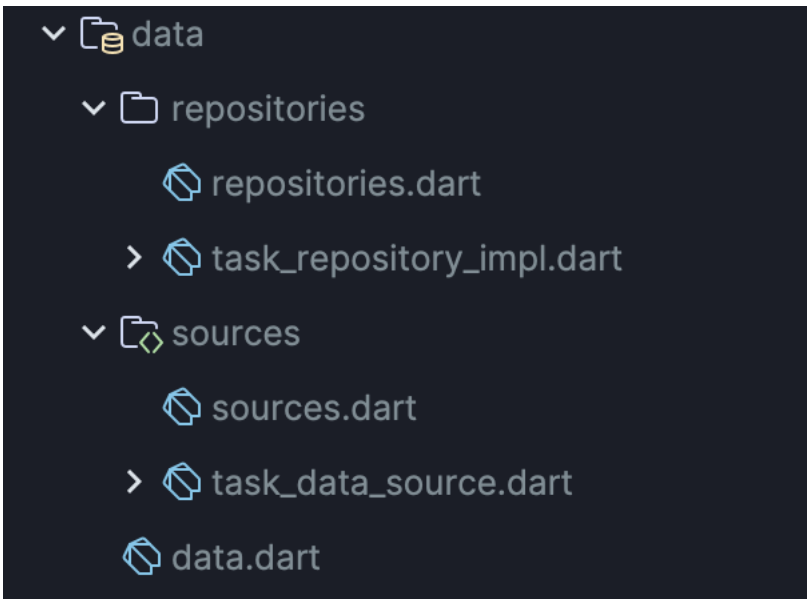


Рисунок 3.34 – Директорія data

На data рівні відбувається отримання даних у sources, та їх парсинг і передача даних через імплементацію репозиторіїв. У нас дані є локальними тому парсинг на даному етапі розробки нам не потрібен, але при подальшій розробці коли буде імплементована синхронізація через запити, нам ще знадобиться, тому подібне розбиття не є оверінжинірингом.

```dart
class TaskRepositoryImpl implements TaskRepository {
  final TaskDataSource taskDataSource;

  TaskRepositoryImpl(this.taskDataSource);

  @override
  TaskEntity add(TaskEntity task) => taskDataSource.add(task);

  @override
  bool delete(TaskEntity task) => taskDataSource.delete(task);

  @override
  List<TaskEntity> getAll() => taskDataSource.getAll();

  @override
  TaskEntity update(TaskEntity task) => taskDataSource.update(task);
}

@riverpod
Future<TaskRepository> tasksRepository(Ref ref) async {
  final taskDataSource = await ref.read(taskDataSourceProvider.future);
  return TaskRepositoryImpl(taskDataSource);
}
```

Рисунок 3.35 – TaskRepositoryImpl; імплементація TaskRepository

На рисунку 3.35 можна помітити функцію tasksRepository що разом з генерованим кодом riverpod допомагає отримати потім tasksRepository вже у вигляді імплементації на рівні контроллера з шару presentation через dependency injection.

```dart
class TaskDataSource {
  final Isar isar;

  TaskDataSource(this.isar);

  List<TaskEntity> getAll() => isar.taskEntitys.where().findAll();

  TaskEntity add(TaskEntity task) => isar.write((isar) {
        final newTask = task.copyWith(id: isar.taskEntitys.autoIncrement());
        isar.taskEntitys.put(newTask);
        return newTask;
      });

  TaskEntity update(TaskEntity task) => isar.write((isar) {
        isar.taskEntitys.put(task);
        return task;
      });

  bool delete(TaskEntity task) =>
      isar.write((isar) => isar.taskEntitys.delete(task.id));
}
```

```dart
@riverpod
Future<TaskDataSource> taskDataSource(Ref ref) async {
  final isar = await ref.read(isarProvider.future);
  return TaskDataSource(isar);
}
```

Рисунок 3.36, 3.37 – TasksDataSource

Аналогічно з імплементацією репозиторію тут можна побачити функцію що разом з riverpod допомагає нам зробити dependency injection, а також передає TaskDataSource з core рівня з конфігурацією поточної БД додатку Isar через provider згенерований riverpod. Код з конфігурацією БД прикріплений у ДОДАТКУ Б

РОЗДІЛ 4
ПЛАН ПОДАЛЬШОГО РОЗВИТКУ ДОДАТКУ

Проект ще знаходиться у активній фазі дослідження, експериментів та розробки тому це далеко не кінцева версія додатку.

Подальший розвиток додатку буде поділено на фази які у процесі будуть випробовуватись та поєднуватись між собою щоб надати користувачу додатку унікальний досвід взаємодії, та також монетизувати. Поточний додаток є початковим концептом, що поступово переходить у першу фазу

Фаза 1 – поточний беклог

- Upcoming список: ділення на розділи списку;
- Розробка зручного календаря для вибору дня виконання;
- Logbook ділення списку на часові проміжки;
- Task: покращення режиму редагування таску, прибирання зайвих елементів під час редагування;
- Перетягування тасків у локальних групах, для зміни пріорітетності.

Фаза 2 – покращення унікальності додатку

- покращення скролу між списками: при скролі пустий простір між завданнями та сам вигляд скролу буде з навгіцією по датам включаючі минулі та майбутні дні разом з поточним;
- анімація переміщення таску між списками в залежності від його модифікації;
- локальний скрол через елементи навігації між списками за допомогою незначного перетягування елементу контролю;
- shake to complete: можливість не переміщувати таски з поточного списку доти, доки юзер не «струсне» їх щоб вони потрапили у відповідні списки (ця функція буде опціональною і може бути змінена у налаштуваннях);
- сторінка налаштувань додатку;

- Focus/Zen Mode: показує лише один поточний таск при жесті зуму. Також є ідея додати таймер помодоро для виконання таску та трекінг часу виконання, для подальшого аналізу;
- тестовий релізу у маркети, за успіху монетизація шляхом разової покупки додатку.

Фаза 3 – Нарощування платних функцій

- створення підписок;
- Collaborative lists: спільні списки, де таски та дії різних юзерів відрізняються за кольором. Хост списку буде мати початкову монохромну тему, та обиратиме кольори для команди (доступні лише за наявності підписки);
- синхронізація та хмарне зберігання задач. На даному етапі у першу чергу задачі зберігаються локально, але для підтримки кросплатформної синхронізації потрібно буде додати можливість зберігання тасків у хмарі. Також це полегшить міграцію користувачів на новий гаджет. (доступна лише за наявності підписки).

Фаза 4 – Подальший розвиток можливостей додатку

- Retrospective: можливість зробити аналіз прогресу за поточний список;
- пошук по таскам;
- кольорові теми (платні, або доступні при підписці);
- віджети додатку;

Фаза 5 – Десктоп версія додатку

- створення нового додатку на базі мобільного для десктопів;
- розширення можливостей адміністрування collaborative lists.

ВИСНОВКИ

Мета дипломної роботи полягала в розробці кросплатформного додатку для реалізації функціоналу task-management. У результаті було розроблено концептну версію додатку, що демонструє базову структуру додатку. Також закладена архітектура дає можливість легко масштабувати додаток та змінювати альтернативні технології.

Поточна версія додатку являє собою групу списків задач у хронологічній послідовності з можливостю виконання базових CRUD операцій, та унікальної навігації між даними списками, інтерфейс якої дає відчуття цілостності та нерозривності списку.

Також було зроблено аналіз переваг конкурентів та можливих напрямів розвитку додатку, з прицілом на подальшу монетизацію та реліз у App Store та Google Play. Результати аналізу були викладені у 4 розділі та розбиті на 4 окремих фази розвитку.

ДЖЕРЕЛА

1. Кросплатформна розробка додатків. URL: https://wezom.com.ua/ua/blog/krossplatformennaya-razrabotka-prilozhenij

2. Еволюція програмного забезпечення для управління проектами. URL: https://pmsoftware.com.ua/istoriya-rozrobok

3. Bullet Journal: Історія та основи . URL: https://bulletjournal.com/pages/learn

4. Історія розвитку хмарних сервісів . URL: https://cloudservices.com.ua/history

5. Інновації в управлінні завданнями. URL: https://techinnovations.org.ua/articles/taskmanagers

6. Things3: офіційний сайт. URL: https://culturedcode.com/things/

7. Notion: огляд функціональності. URL: https://www.notion.so/product

8. Obsidian: переваги та особливості. URL: https://obsidian.md/

9. Minimalist: опис додатку . URL: https://minimalistapp.com/

10. Jira: інструмент управління проектами. URL: https://www.atlassian.com/software/jira

11. Trello: використання Kanban-дошок. URL: https://trello.com/

12. What is Flutter. URL: https://docs.flutter.dev/resources/faq

13. Сторінка пакету freezed. URL: https://pub.dev/packages/freezed

14. Документація бібліотеки Riverpod [Електронний ресурс]: [Веб-сайт] – https://riverpod.dev/

15. Документація пакету Isar. URL: https://isar.dev/

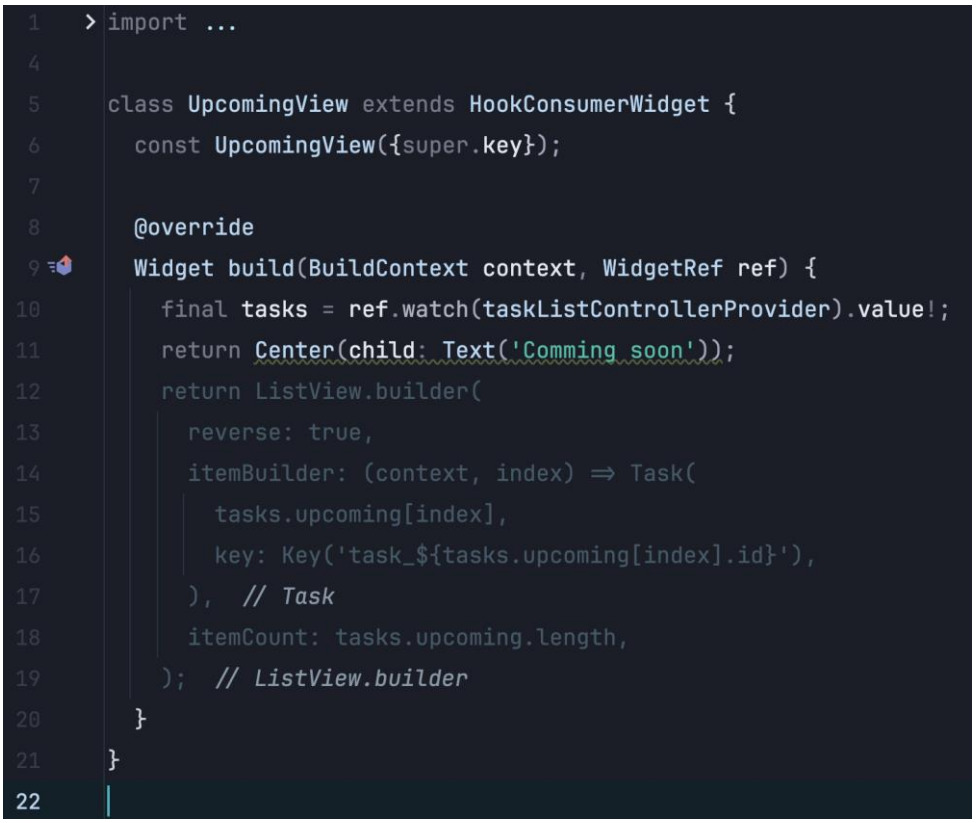
ДОДАТОК А
ВЕРСТКА СПИСКІВ ЗАДАЧ ДОДАТКУ

```dart
import ...

class LogbookView extends HookConsumerWidget {
  const LogbookView({super.key});

  @override
  Widget build(BuildContext context, WidgetRef ref) {
    final tasks = ref.watch(taskListControllerProvider).value!;

    return ListView.builder(
      reverse: true,
      itemBuilder: (context, index) => Task(
        tasks.complete[index],
        key: Key('task_${tasks.complete[index].id}'),
      ),  // Task
      itemCount: tasks.complete.length,
    );  // ListView.builder
  }
}
```

Рисунок А.1 – Список виконаних задач

```dart
import ...

class TodayView extends HookConsumerWidget {
  const TodayView({super.key});

  @override
  Widget build(BuildContext context, WidgetRef ref) {
    final tasks = ref.watch(taskListControllerProvider).value!;

    return ListView.separated(
      padding: const EdgeInsets.symmetric(horizontal: 8),
      reverse: true,
      separatorBuilder: (context, index) => const SizedBox(height: 4),
      itemBuilder: (context, index) => Task(
        tasks.current[index],
        key: Key('task_${tasks.current[index].id}'),
      ),  // Task
      itemCount: tasks.current.length,
    );  // ListView.separated
  }
}
```

Рисунок А.2 – Список поточних задач



```dart
import ...

class UpcomingView extends HookConsumerWidget {
  const UpcomingView({super.key});

  @override
  Widget build(BuildContext context, WidgetRef ref) {
    final tasks = ref.watch(taskListControllerProvider).value!;
    return Center(child: Text('Comming soon'));
    return ListView.builder(
      reverse: true,
      itemBuilder: (context, index) => Task(
        tasks.upcoming[index],
        key: Key('task_${tasks.upcoming[index].id}'),
      ),  // Task
      itemCount: tasks.upcoming.length,
    );  // ListView.builder
  }
}
```

Рисунок А.3 – Список майбутніх задач

ДОДАТОК Б
КОНФІГУРАЦІЯ БД ДОДАТКУ

```dart
import ...

part 'isar.g.dart';

@riverpod
Future<Isar> isar(Ref ref) async {
  final dir = await getApplicationDocumentsDirectory();
  return Isar.open(schemas: [TaskEntitySchema], directory: dir.path);
}
```

Рисунок Б.3 – Список майбутніх задач