Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти перший (бакалаврський)

Кросплатформний застосунок для організації навчального процесу

(тема)

Виконав:
здобувач 4 року навчання,
групи КІУКІ-21-5
Віктор ВОЛОШКО
(власне ім'я, прізвище)

Спеціальність
123 «Комп'ютерна інженерія»
(код і повна назва спеціальності)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма
Комп'ютерна інженерія
(повна назва освітньої програми)

Керівник: ас. Єгор КОРНІЄНКО
(посада, власне ім'я, прізвище)

Допускається до захисту

Завідувач кафедри ЕОМ _____ Андрій КОВАЛЕНКО
(підпис) (власне ім'я, прізвище)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти перший (бакалаврський)

Спеціальність 123 «Комп’ютерна інженерія»
(код і повна назва)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма Комп’ютерна інженерія
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
“____” _____ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Волошку Віктору Валерійовичу
(прізвище, ім’я, по батькові)

1. Тема роботи Кросплатформний застосунок для організації навчального процесу

затверджена наказом по університету від “ 26 ” травня 2025 р. № 424 Ст

2. Термін подання здобувачем роботи до екзаменаційної комісії 17 червня 2025 р.

3. Вхідні дані до роботи 1) API сервісу cist.nure.ua; 2) фреймворк Flutter; 3) цільові платформи: Android та iOS

4. Перелік питань, що потрібно опрацювати у роботі
1) аналіз проблеми та огляд існуючих рішень;
2) вибір технології та засобів розробки;
3) розробка застосунку;
4) тестування застосунку;
5) висновки.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій

Слайд-презентація – 13 слайдів

6. Консультанти розділів роботи (заповнюється за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Аналіз проблеми та огляд існуючих рішень	27.05.25-30.05.25	
2	Вибір технології та засобів розробки	31.05.25-02.06.25	
3	Розробка застосунку	03.06.25-05.06.25	
4	Тестування та відлагодження застосунку	06.06.25-09.06.25	
5	Оформлення матеріалів кваліфікаційної роботи	10.06.25-11.06.25	
6	Подання кваліфікаційної роботи керівникові та	12.06.25-13.06.25	
	її попередній захист		
7	Подання кваліфікаційної роботи на рецензу-	14.06.25-16.06.25	
	вання		

Дата видачі завдання " 26 " травня 2025 р.

Здобувач _____
(підпис)

Керівник роботи _____ ас. Єгор КОРНІЄНКО
(підпис) (посада, власне ім'я, прізвище)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 111 с., 33 рис., 2 дод., 27 джерел.

КАЛЕНДАР, СПИСОК ЗАДАЧ, FLUTTER, DART, ЧИСТА АРХІТЕКТУРА, MVVM.

Метою кваліфікаційної роботи є розробка кросплатформного застосунку для керування навчальним процесом.

У ході виконання кваліфікаційної роботи було проаналізовано проблему та існуючі рішення, на основі яких поставлено задачу стосовно розробки проєкту. У ході його розробки було розглянуто засоби для рішення задачі, розроблено та протестовано кросплатформний застосунок з використанням фреймворку Flutter з дотриманням принципів чистої архітектури та патерну MVVM, що задовольняє поставлену задачу. Результат проаналізований, зроблено висновки.

ABSTRACT

Bachelor’s thesis: 111 pages, 33 figures, 2 appendices, 27 sources.

CALENDAR, TASK LIST, FLUTTER, DART, CLEAN ARCHITECTURE, MVVM.

The major goal of this thesis is the development of a cross-platform application for managing the educational process.

In order to finish qualification thesis, the problem and existing solutions were analyzed, on the basis of which the task regarding the project development was set. During its development, tools for solving the problem were considered, a cross-platform application was developed and tested using the Flutter framework in compliance with the principles of clean architecture and the MVVM pattern, which satisfies the set task. The result was analyzed, and conclusions were drawn.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

ОС – операційна система

СУБД – система управління базами даних

API – Application Programming Interface

DI – Dependency Injection

HTTP – Hypertext Transfer Protocol

JSON – JavaScript Object Notation

KMP – Kotlin Multiplatform

MAUI – Multi-platform App UI

MVC – Model-View-Controller

MVI – Model-View-Intent

MVP – Model-View-Presenter

MVVM – Model-View-ViewModel

SQL – Structured Query Language

UI – User Interface

URL – Uniform Resource Locator

ВСТУП

Наразі, існуючі на ринку програмні рішення для управління навчальним розкладом та задачами не повною мірою задовольняють функціональні потреби користувачів, зокрема студентів ХНУРЕ, що зумовлює актуальність розробки нового, більш функціонального програмного продукту.

Інтеграція мобільних застосунків у повсякденне життя та навчальну діяльність є невід'ємною складовою сучасного життя. Кросплатформна розробка є особливо актуальною зараз [1], оскільки більшість людей мають більш ніж один пристрій та хочуть отримувати доступ до одних й тих самих сервісів та застосунків з якомога більшого числа наявних у них пристроїв. Для розробки таких застосунків чудово підходить Flutter, що наразі підтримує всі популярні платформи: від мобільних ОС Android та iOS та декстопних ОС Windows, Linux та macOS до веб-платформ. Він є актуальним вибором для вирішення поставлених задач [2].

Розробка нового програмного продукту є доцільною лише за умови наявності переваг над існуючими аналогами. Окрім базової функції відображення навчального розкладу, передбачається можливість додання користувацьких подій. Система управління завданнями, окрім стандартних операцій створення, редагування та видалення, повинна включати інноваційну функцію автоматичної генерації актуального переліку завдань на основі даних розкладу.

Також варто врахувати важливість дотримання певної архітектури при створенні застосунку оскільки використання «чистої архітектури» та архітектурного патерну є практично обов’язковим у сучасній розробці [3]. Це є невід'ємною складовою для підтримки застосунку у майбутньому, розширення функціоналу та дозволяє іншим розробникам швидше розібратись у вихідному коді застосунку та доєднатись до розробки.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналіз засобів кросплатформенної розробки

У сучасному світі мобільні застосунки стали невід'ємною частиною повсякденного життя та важливим інструментом для бізнесу. З огляду на різноманітність операційних систем, зокрема iOS та Android, розробники постають перед вибором стратегії створення програмних продуктів. Кросплатформна розробка пропонує підхід, що дозволяє використовувати єдину кодову базу для створення застосунків, здатних функціонувати на різних платформах.

Однією з головних переваг є потенційна економія ресурсів [1]. Замість того, щоб формувати окремі команди та писати код для кожної платформи індивідуально, компанії можуть оптимізувати витрати та час, використовуючи єдиний набір інструментів та спільну кодову базу. Це, в свою чергу, може пришвидшити вихід продукту на ринок, що є критично важливим в умовах високої конкуренції.

Оновлення та підтримка застосунків також спрощуються, оскільки зміни, внесені до спільного коду, автоматично застосовуються до версій для різних операційних систем. Це сприяє узгодженості функціонала та користувацького досвіду на всіх пристроях. Крім того, кросплатформенні рішення дозволяють охопити ширшу аудиторію користувачів з різними пристроями.

Незважаючи на очевидні переваги, кросплатформенна розробка має і певні виклики. Одним із суттєвих аспектів є потенційні проблеми з продуктивністю. Застосунки, створені за допомогою кросплатформенних інструментів, не завжди можуть досягти того ж рівня плавності та швидкодії, що й нативні аналоги, особливо при виконанні складних графічних завдань [2].

Іншим важливим моментом є доступ до нативних функцій пристрою. Хоча сучасні кросплатформенні фреймворки значно покращили цей аспект, іноді можуть виникати складнощі з інтеграцією специфічних можливостей операційної системи, таких як камера або GPS, що може потребувати додаткових рішень або плагінів.

Також існує ризик, що користувацький інтерфейс може виглядати або відчуватися "не зовсім рідним" на тій чи іншій платформі. Кожна операційна система має свої унікальні гайдлайни та патерни взаємодії, і повне їх відтворення в рамках єдиної кодової бази може бути складним завданням. Це може вплинути на загальне сприйняття застосунку користувачами, які звикли до певного "відчуття" своєї платформи.

1.1.1 Фреймворк React Native

React Native – це фреймворк, який дозволяє створювати мобільні застосунки, що виглядають і відчуваються як нативні, використовуючи знайомі багатьом веб-розробникам технології. В його основі лежить популярна JavaScript-бібліотека React, що робить поріг входження для досвідчених фронтенд-розробників відносно невисоким [4].

Однією з ключових переваг є можливість значної частини коду бути спільною між платформами iOS та Android. Це означає, що команди розробників можуть ефективніше використовувати свій час та ресурси, не дублюючи зусилля для кожної операційної системи окремо. Це, в свою чергу, часто призводить до скорочення термінів розробки та, потенційно, до зменшення загальної вартості проєкту.

Швидкість розробки також виграє завдяки таким функціям, як "гаряче перезавантаження" (hot reloading), що дозволяє миттєво бачити зміни в коді без необхідності повної перекомпіляції застосунку. Це значно прискорює ітераційний процес та полегшує експерименти з інтерфейсом та логікою.

Велика та активна спільнота розробників – ще один вагомий плюс.

Наявність численних готових бібліотек, компонентів та інструментів, а також велика кількість навчальних матеріалів та форумів, де можна знайти відповіді на питання, суттєво спрощує процес розробки та вирішення можливих проблем.

Також варто відзначити, що React Native дозволяє створювати інтерфейси, які використовують нативні компоненти операційної системи [5]. Це означає, що користувацький досвід часто буває дуже близьким до того, що пропонують повністю нативні застосунки, оскільки елементи управління виглядають і поводяться звично для користувачів кожної конкретної платформи.

Незважаючи на численні переваги, існують і певні аспекти, які варто враховувати. Хоча React Native прагне забезпечити високу продуктивність, у випадках дуже складних анімацій або інтенсивних обчислювальних завдань, продуктивність може поступатися повністю нативним рішенням [5]. Це пов'язано з тим, що між JavaScript-кодом та нативними компонентами існує так званий "міст", який може вносити певні затримки (рис. 1.1) [6].

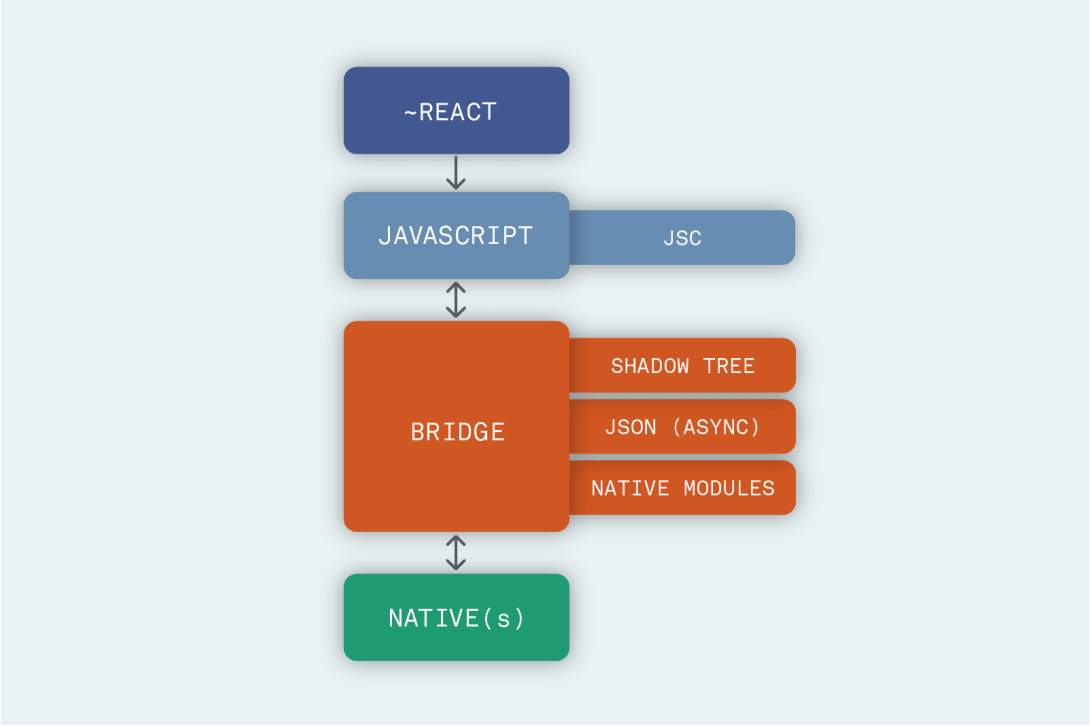


Рисунок 1.1 – Архітектура React Native

Доступ до деяких специфічних нативних функцій або API іноді може вимагати написання додаткового нативного коду (так званих "нативних модулів"). Хоча спільнота пропонує багато готових рішень для популярних функцій, унікальні або новітні можливості платформи можуть потребувати додаткових зусиль для інтеграції.

Процес налагодження (дебагінгу) іноді може бути складнішим, ніж у чисто нативних проектах, оскільки проблеми можуть виникати як на рівні JavaScript, так і на рівні нативного коду, а також у взаємодії між ними [5].

Також варто пам'ятати, що оновлення самої платформи React Native, хоча й приносять нові можливості та виправлення, іноді можуть призводити до необхідності адаптації існуючого коду, що вимагає додаткового часу та тестування.

1.1.2 Фреймворк NativeScript

NativeScript представляє собою цікавий підхід до створення мобільних програм, які можуть працювати на різних операційних системах, спираючись на єдину кодову базу. Його особливість полягає в тому, що він дозволяє розробникам використовувати вже знайомі їм веб-технології, такі як JavaScript або TypeScript, а також популярні фреймворки на кшталт Angular або Vue.js, для побудови інтерфейсів та логіки застосунків, які при цьому безпосередньо взаємодіють з нативними компонентами та API платформи [4].

Однією з найважливіших відмінних рис NativeScript є його архітектура (рис 1.2), яка забезпечує прямий доступ до всіх нативних API операційних систем iOS та Android безпосередньо з JavaScript або TypeScript. Це означає, що розробники можуть викликати будь-які функції, доступні для нативних розробників, не вдаючись до складних "мостів" або обгорток, що часто позитивно впливає на продуктивність та можливості застосунку.

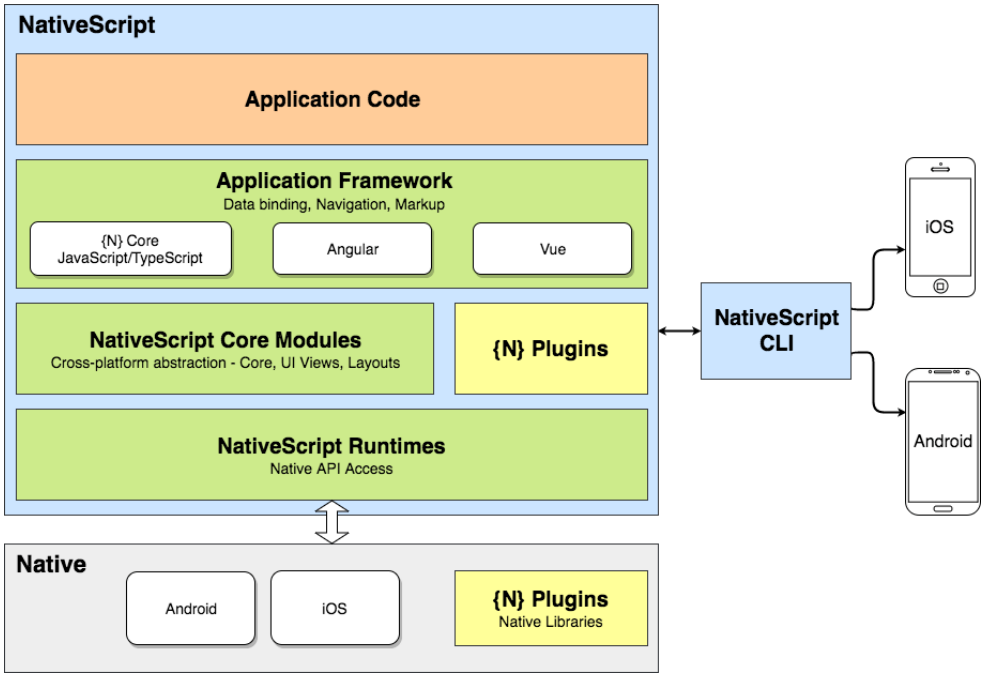


Рисунок 1.2 – Архітектура NativeScript

Використання JavaScript або TypeScript, а також інтеграція з такими фреймворками, як Angular та Vue.js, робить NativeScript привабливим для команд, які вже мають досвід у веб-розробці. Це дозволяє скоротити час на навчання та швидше приступити до створення продукту. Можливість використовувати знайомі інструменти та концепції, такі як компоненти, сервіси та керування станом, також сприяє ефективності [4].

Застосунки, створені за допомогою NativeScript, використовують нативні елементи інтерфейсу користувача. Це означає, що кнопки, списки, поля вводу та інші елементи будуть виглядати та поводитися так, як очікують користувачі кожної конкретної платформи, що сприяє кращому користувацькому досвіду та "рідному" відчуттю застосунку.

Спільнота, хоч і може бути не такою великою, як у деяких інших кросплатформенних рішень, є досить активною та надає підтримку, плагіни та інструменти, що розширюють функціональність фреймворку.

Попри значні переваги, існують і певні моменти, на які варто звернути увагу. Хоча прямий доступ до нативних API є потужною можливістю, він

також може означати, що для реалізації деяких складних або дуже специфічних для платформи функцій розробникам може знадобитися глибше розуміння особливостей роботи iOS та Android.

Поріг входження для розробників, які не знайомі з Angular або Vue.js (якщо обрано відповідний варіант NativeScript), може бути дещо вищим. Хоча можна використовувати і "чистий" JavaScript/TypeScript, використання цих фреймворків часто є рекомендованим для побудови складних застосунків [4].

Розмір спільноти та екосистеми плагінів, хоч і зростає, може бути меншим порівняно з деякими більш масовими кросплатформенними інструментами. Це іноді може означати, що для інтеграції певної рідкісної функціональності доведеться писати власні нативні модулі або шукати менш поширені рішення.

Також, як і з будь-яким кросплатформенним рішенням, оновлення операційних систем можуть іноді вимагати адаптації коду або оновлення самого фреймворку для забезпечення повної сумісності та доступу до нових можливостей.

1.1.3 Фреймворк Ionic

Ionic – це ще один популярний інструментарій для створення мобільних, веб- та десктопних застосунків з єдиної кодової бази. Його основна ідея полягає у використанні стандартних веб-технологій, таких як HTML, CSS та JavaScript (або TypeScript), разом із фреймворками на кшталт Angular, React або Vue.js, для побудови користувацького інтерфейсу [4]. Для доступу до нативних функцій пристрою Ionic традиційно використовував Apache Cordova, а згодом запропонував власне рішення – Capacitor, яке надає більш сучасний підхід до взаємодії з нативними API.

Однією з найбільших переваг Ionic є його низький поріг входження для веб-розробників. Команди, які вже мають досвід роботи з HTML, CSS та

JavaScript, а також з популярними фронтенд-фреймворками, можуть відносно швидко почати створювати мобільні застосунки за допомогою Ionic. Це значно розширює коло потенційних розробників мобільних додатків.

Ionic пропонує велику бібліотеку готових, стилізованих компонентів користувацького інтерфейсу, які адаптуються під вигляд тієї чи іншої платформи (iOS або Android). Це дозволяє швидко створювати візуально привабливі та функціональні інтерфейси без необхідності розробляти кожен елемент з нуля.

Можливість створювати не лише мобільні застосунки, але й прогресивні веб-застосунки (PWA) та навіть десктопні програми (за допомогою Electron) з тієї ж кодової бази є суттєвим плюсом. Це забезпечує гнучкість та дозволяє охопити максимальну кількість платформ з мінімальними зусиллями.

Швидке прототипування та розробка – ще одна сильна сторона. Завдяки використанню веб-технологій та інструментів, таких як "live reload" (миттєве оновлення в браузері або на пристрої при зміні коду), ітераційний процес стає дуже швидким.

Оскільки Ionic-застосунки, по суті, є веб-сторінками, що виконуються всередині нативної оболонки (рис. 1.3), продуктивність у дуже складних, графічно насичених або обчислювально інтенсивних сценаріях може поступатися повністю нативним рішенням або фреймворкам, які компілюють код у нативні компоненти. Хоча сучасні WebView стали значно продуктивнішими, це залишається важливим фактором для певних типів застосунків [4].

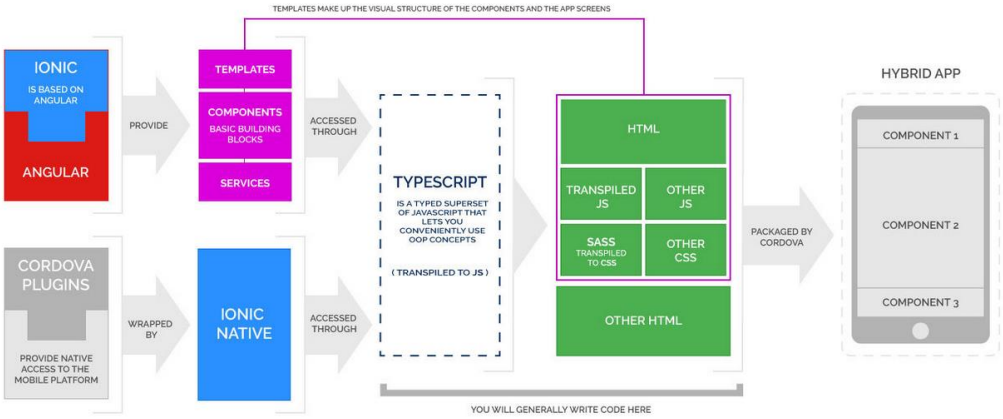


Рисунок 1.3 – Архітектура Ionic

Користувацький досвід, хоч і максимально наближений до нативного завдяки стилізованим компонентам, іноді може видавати своє веб-походження, особливо в дрібних деталях анімацій або поведінці деяких елементів управління. Для проєктів, де абсолютна відповідність "рідному" вигляду та відчуттю є критичною, це може бути недоліком.

Залежність від плагінів (через Cordova або Capacitor) для доступу до більшості нативних функцій пристрою (камера, геолокація, файлова система тощо) означає, що якість та актуальність цих плагінів може впливати на стабільність та функціональність застосунку. Хоча існує велика кількість плагінів, іноді можуть виникати проблеми з їх сумісністю або підтримкою.

1.1.4 Фреймворк .NET MAUI та Xamarin

Xamarin свого часу з'явився як інструмент, що дозволив .NET-розробникам увійти у світ мобільної розробки без необхідності глибокого вивчення абсолютно нових мов програмування, таких як Java/Kotlin для Android чи Objective-C/Swift для iOS. Його ключова ідея полягала в тому, щоб дати можливість писати бізнес-логіку застосунку один раз мовою C#, а потім спільно використовувати її на різних платформах [7]. Однією з головних переваг Xamarin є доступ до нативних API кожної платформи.

Застосунки, розроблені за допомогою Xamarin, компілюються в нативний код, що часто забезпечує високу продуктивність, близьку до тієї, яку мають повністю нативні програми. Це дозволяє використовувати всі можливості пристрою та створювати користувацькі інтерфейси, які виглядають і відчуваються як "рідні" для кожної операційної системи.

Xamarin пропонував два основні підходи до створення інтерфейсу. Перший, відомий як Xamarin.Native (що включав Xamarin.iOS та Xamarin.Android), передбачав створення окремого користувацького інтерфейсу для кожної платформи, використовуючи специфічні для неї інструменти та бібліотеки, але при цьому зберігаючи спільну бізнес-логіку на C#. Це надавало максимальну гнучкість та контроль над виглядом та поведінкою інтерфейсу на кожній платформі. Другий підхід, Xamarin.Forms, дозволяв створювати єдиний користувацький інтерфейс за допомогою абстрактного шару, який потім транслювався у відповідні нативні елементи управління для кожної платформи. Це значно спрощувало та прискорювало розробку для випадків, де не вимагалася унікальна кастомізація інтерфейсу під кожну ОС, дозволяючи досягти максимального повторного використання коду.

Незважаючи на свої переваги, Xamarin мав і певні виклики. Наприклад, розмір застосунків іноді міг бути більшим через включення необхідних бібліотек .NET. Також, хоча Xamarin.Forms значно спрощував створення UI, іноді досягнення складних, специфічних для платформи дизайнів або поведінки могло вимагати додаткових зусиль або написання кастомних рендерерів [7].

.NET Multi-platform App UI, або .NET MAUI, є наступним кроком у розвитку кросплатформенних рішень від Microsoft, по суті, еволюцією Xamarin.Forms. Головна мета .NET MAUI – ще більше спростити та уніфікувати процес розробки для різних платформ, включаючи не тільки мобільні iOS та Android, але й десктопні Windows та macOS, з єдиного проєкту та єдиної кодової бази (рис 1.4) [8].

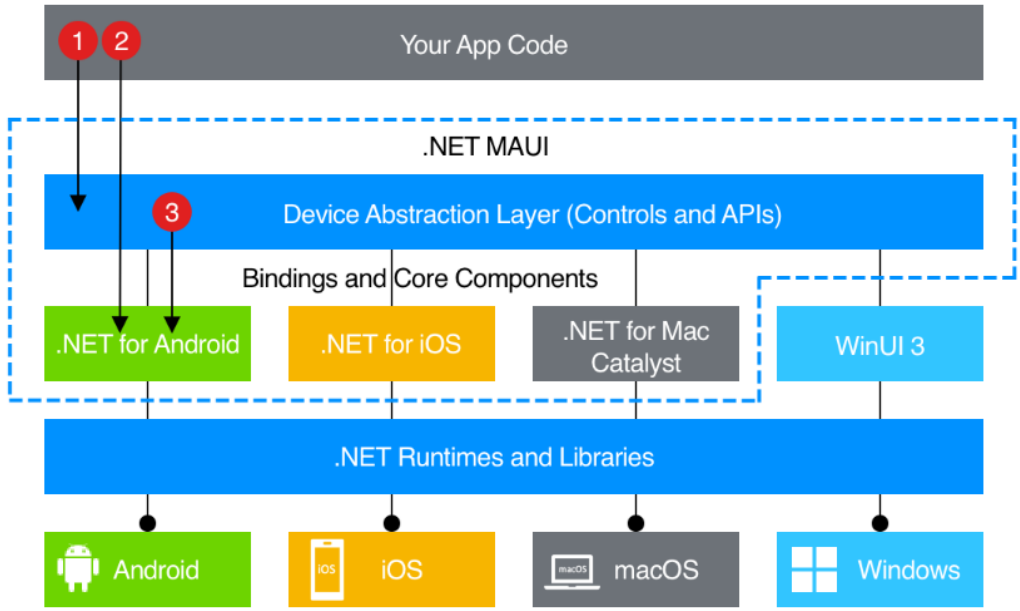


Рисунок 1.4 – Архітектура .NET MAUI

Ключові вдосконалення в .NET MAUI включають, по-перше, єдиний проєкт. Замість кількох проєктів для кожної платформи, як це часто було в Xamarin, .NET MAUI пропонує структуру з єдиним проєктом, що значно спрощує керування кодом та ресурсами. По-друге, це уніфіковані бібліотеки: .NET MAUI використовує переваги .NET 6 (і новіших версій), надаючи доступ до уніфікованого набору бібліотек та інструментів. По-третє, покращена архітектура, яка спрямована на поліпшення продуктивності, гнучкості та розширюваності. По-четверте, графічний стек: .NET MAUI надає можливість використовувати не тільки нативні контролери, а й малювати власні елементи інтерфейсу за допомогою графічних API, що дає більше контролю над виглядом застосунку. І по-п'яте, розширений доступ до нативних функцій, адже продовжується робота над спрощенням доступу до нативних API та функцій пристроїв.

.NET MAUI прагне взяти найкраще від Xamarin.Forms, зокрема, підхід до створення єдиного UI, та покращити його, вирішуючи деякі з попередніх обмежень та надаючи розробникам більш сучасний та продуктивний інструментарій [7].

При виборі .NET MAUI (або при міграції з Xamarin) варто враховувати, що, як і будь-яка відносно нова технологія, її екосистема, що включає сторонні бібліотеки, інструменти, обсяг документації та спільноти, все ще активно розвивається. Хоча вона базується на міцному фундаменті Xamarin, деякі специфічні рішення або підходи можуть потребувати часу на адаптацію чи пошук. Продуктивність застосунків, створених за допомогою .NET MAUI, загалом очікується на високому рівні завдяки компіляції в нативний код та оптимізаціям, проте, як і завжди, фінальна продуктивність залежатиме від складності застосунку та якості написаного коду.

1.1.5 Фреймворк Kotlin Multiplatform

Kotlin Multiplatform, або KMP, є технологією від JetBrains, творців мови програмування Kotlin. Її основна ідея полягає в тому, щоб дозволити розробникам писати та повторно використовувати спільний код, написаний на Kotlin, для різних платформ, включаючи Android, iOS, веб, десктоп та навіть серверні застосунки. Однак, на відміну від деяких інших кросплатформенних рішень, які прагнуть уніфікувати і користувацький інтерфейс, KMP зосереджується переважно на спільному використанні бізнес-логіки, залишаючи створення інтерфейсу нативним для кожної платформи [9].

Ключова концепція Kotlin Multiplatform полягає у можливості визначення спільних модулів, написаних на Kotlin, які містять логіку, що не залежить від конкретної платформи – це можуть бути алгоритми, робота з даними, мережеві запити, керування станом тощо. Потім для кожної цільової платформи (наприклад, Android або iOS) створюються специфічні модулі, які можуть реалізовувати платформо-залежні частини або взаємодіяти з нативними API (рис. 1.5). Таким чином, розробники отримують можливість писати основну, часто найскладнішу частину застосунку, лише один раз, а потім інтегрувати її в нативні проекти для кожної ОС.

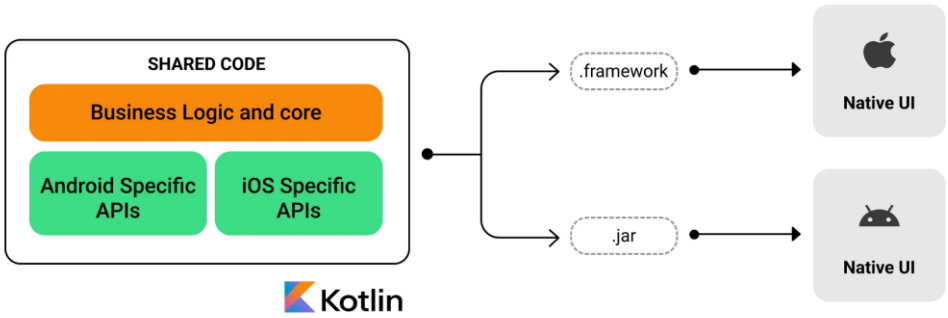


Рисунок 1.5 – Архітектура KMP

Однією з головних переваг такого підходу є те, що користувацький інтерфейс залишається повністю нативним. Це означає, що застосунки, створені з використанням KMP, можуть мати вигляд, відчуття та продуктивність, абсолютно ідентичні тим, що розроблені повністю нативними засобами для Android (з використанням Kotlin або Java та Android SDK) та iOS (з використанням Swift або Objective-C та UIKit/SwiftUI). Це вирішує одну з потенційних проблем деяких інших кросплатформенних інструментів, де інтерфейс може іноді відчуватися "не зовсім рідним".

Використання Kotlin як основної мови для спільного коду також є сильною стороною. Kotlin – це сучасна, виразна та безпечна мова, яка вже є офіційною для розробки під Android. Це робить її природним вибором для Android-розробників та полегшує їм освоєння KMP. Для iOS-розробників Kotlin також компілюється в нативний код, що забезпечує хорошу сумісність та продуктивність.

Можливість поступового впровадження KMP в існуючі проекти є ще одним плюсом. Немає необхідності переписувати весь застосунок одразу. Можна почати з винесення невеликої частини бізнес-логіки в спільний модуль, а потім поступово розширювати його використання.

Однак, варто враховувати й певні аспекти. Оскільки користувацький інтерфейс розробляється нативно для кожної платформи, це все ще вимагає наявності розробників, знайомих з нативними інструментами та підходами

для iOS та Android (або готовності команди вивчати їх). Це може збільшити загальний обсяг роботи порівняно з фреймворками, що пропонують єдиний UI [9].

Екосистема навколо Kotlin Multiplatform, хоч і швидко зростає, може бути ще не такою зрілою, як у деяких більш усталених кросплатформенних рішень, особливо коли йдеться про готові бібліотеки для певних специфічних завдань у спільному коді. Іноді може знадобитися написання власних платформо-специфічних реалізацій для доступу до деяких API.

1.1.6 Фреймворк Flutter

Flutter – це розроблений Google інструментарій (UI toolkit) для створення візуально привабливих та швидко працюючих застосунків для мобільних, веб- та десктопних платформ з єдиної кодової бази. Його ключова особливість полягає в тому, що він не використовує нативні компоненти операційної системи для відображення інтерфейсу, а натомість самостійно малює кожен піксель на екрані за допомогою власного високопродуктивного графічного двигуна Skia або Impeller. Це дає розробникам надзвичайний контроль над зовнішнім виглядом та анімаціями застосунку [5].

Основною мовою програмування для Flutter є Dart, також розроблена Google. Dart – це об'єктно-орієнтована мова, оптимізована для розробки клієнтських застосунків, яка компілюється в нативний ARM-код для мобільних платформ, що забезпечує високу продуктивність.

Однією з найпомітніших переваг Flutter є його здатність створювати надзвичайно гнучкі та кастомні користувацькі інтерфейси. Завдяки підходу "все є віджет", розробники можуть комбінувати та налаштовувати готові віджети або створювати власні з нуля, досягаючи унікального дизайну, який може бути ідентичним на всіх платформах. Це особливо цінно для брендів, які прагнуть мати впізнаваний та послідовний візуальний стиль своїх застосунків [5].

Швидкість розробки – ще одна сильна сторона Flutter. Функція "гарячого перезавантаження" (hot reload) дозволяє розробникам миттєво бачити зміни в коді, не втрачаючи поточного стану застосунку. Це значно прискорює ітераційний процес, експерименти з UI та виправлення помилок. Також багатий набір готових віджетів, що імітують як Material Design (для Android), так і Cupertino (для iOS), дозволяє швидко збирати інтерфейси.

Продуктивність застосунків, створених на Flutter, зазвичай дуже висока, часто не поступаючись повністю нативним рішенням. Це досягається завдяки прямій компіляції в машинний код та використанню потужного графічного двигуна. Плавні анімації та швидка реакція інтерфейсу є характерними рисами Flutter-застосунків [5].

Спільнота навколо Flutter активно зростає, що призводить до збільшення кількості доступних пакетів (плагінів та бібліотек), навчальних матеріалів та готових рішень для різноманітних завдань. Це полегшує інтеграцію з різними сервісами та доступ до нативних функцій пристрою через так звані "платформні канали".

Однак, існують і певні аспекти, які варто враховувати. Розмір застосунків, створених на Flutter, іноді може бути дещо більшим порівняно з повністю нативними аналогами, оскільки вони включають власний графічний двигун та необхідні бібліотеки (рис. 1.6). Хоча розробники Flutter постійно працюють над оптимізацією, це може бути фактором для деяких проєктів [10].

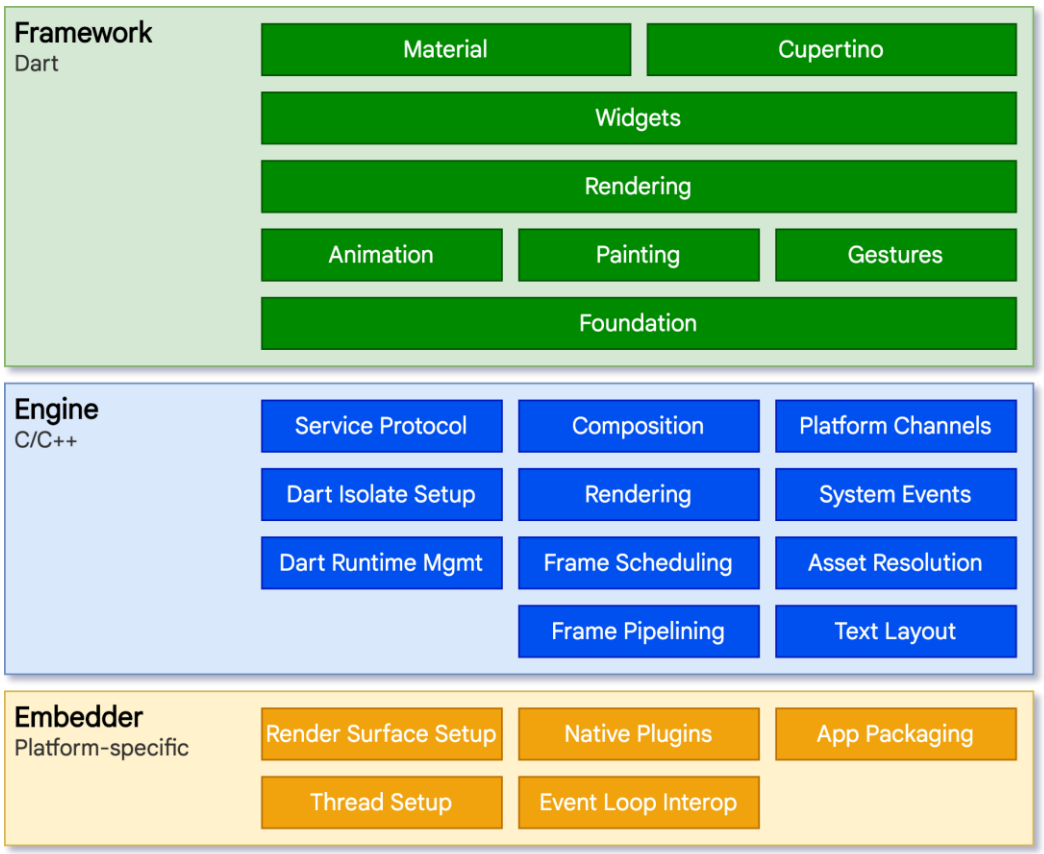


Рисунок 1.6 – Архітектура Flutter

Мова Dart, хоч і є сучасною та потужною, може вимагати певного часу на освоєння для розробників, які раніше з нею не працювали. Це може бути додатковим бар'єром для команд, які вже мають великий досвід з іншими мовами.

Хоча екосистема пакетів стрімко розвивається, для деяких дуже специфічних або новітніх нативних функцій може ще не існувати готового, добре протестованого плагіна, що може вимагати написання власного платформо-специфічного коду.

Також, оскільки Flutter малює власний UI, іноді можуть виникати дуже дрібні відмінності у поведінці або зовнішньому вигляді деяких елементів порівняно з їхніми строго нативними аналогами, хоча Flutter надає великий набір інструментів для максимального наближення до "рідного" вигляду та відчуття.

1.2 Архітектурні та дизайн-патерни

Архітектурні та дизайн-патерни – це, по суті, набір перевірених часом рішень для типових проблем, що виникають під час розробки програмного забезпечення. У контексті створення мобільних застосунків, які мають працювати на різних операційних системах, їхня роль стає ще вагомішою.

Коли розмова йде про архітектурні патерни, мається на увазі загальна структуру застосунку. Вони допомагають організувати код таким чином, щоб він був логічним, легко підтримувався та масштабувався. У кросплатформеній розробці це особливо важливо, адже потрібно враховувати особливості різних операційних систем, зберігаючи при цьому єдину кодову базу наскільки це можливо. Грамотно обрана архітектура дозволяє розділити відповідальність між різними частинами програми, що спрощує розробку, тестування та подальшу модифікацію. Це як закласти міцний фундамент для будинку – якщо він надійний, то й сам будинок буде стояти довго та витримає будь-які навантаження.

Дизайн-патерни, у свою чергу, пропонують рішення для більш конкретних завдань всередині програми. Вони стосуються взаємодії об'єктів та компонентів. Використання дизайн-патернів робить код більш гнучким, зрозумілим для інших розробників та менш схильним до помилок. В умовах кросплатформенності, де часто доводиться працювати з різними нативними можливостями пристроїв, дизайн-патерни допомагають створювати уніфіковані інтерфейси для взаємодії з цими можливостями, приховуючи складність реалізації «під капотом».

1.2.1 «Чиста архітектура»

«Чиста архітектура» (Clean Architecture), запропонована Робертом Мартіном, представляє собою набір архітектурних принципів, спрямованих на розробку програмних систем, що характеризуються високим ступенем

гнучкості, тестованості, підтримуваності та незалежності від зовнішніх технологічних залежностей, таких як фреймворки, системи управління базами даних (СУБД) та користувацькі інтерфейси (UI) [11].

Центральним елементом концепції є організація програмного коду у вигляді концентричних шарів, де кожен шар інкапсулює специфічний рівень абстракції та відповідальності. Фундаментальним правилом є принцип інверсії залежностей, згідно з яким усі залежності спрямовані виключно до внутрішніх шарів. Це означає, що внутрішні компоненти системи не мають жодної інформації про зовнішні.

Типова структура шарів (рис. 1.7), рухаючись від ядра до периферії, виглядає наступним чином:

- Entities (сутності): цей шар формує ядро системи та містить об'єкти, що інкапсулюють фундаментальні бізнес-концепції та найбільш загальні бізнес-правила, що не залежать від конкретного застосування;
- Use Cases: даний шар містить логіку, специфічну для конкретного застосування, та керує потоком даних до сутностей;
- Interface Adapters: цей шар виконує роль посередника між доменом бізнес-логіки та інфраструктурними компонентами (UI чи база даних), компоненти цього шару відповідають за трансформацію даних між форматом, що використовується у внутрішніх шарах, та форматом, необхідним для зовнішніх систем;
- Frameworks and Drivers: найзовнішніший шар, що містить конкретні технологічні реалізації: веб-фреймворки, UI-фреймворки, бази даних, зовнішні сервіси та компоненти користувацького інтерфейсу і є найбільш залежним від деталей реалізації та найбільш схильним до змін.

Ключовим аспектом є строге дотримання напрямку залежностей: вони можуть бути спрямовані лише всередину. Зовнішні шари залежать від внутрішніх, але не навпаки. Це досягається шляхом застосування принципу інверсії залежностей, де внутрішні шари визначають інтерфейси, а зовнішні шари надають їх реалізації [12].

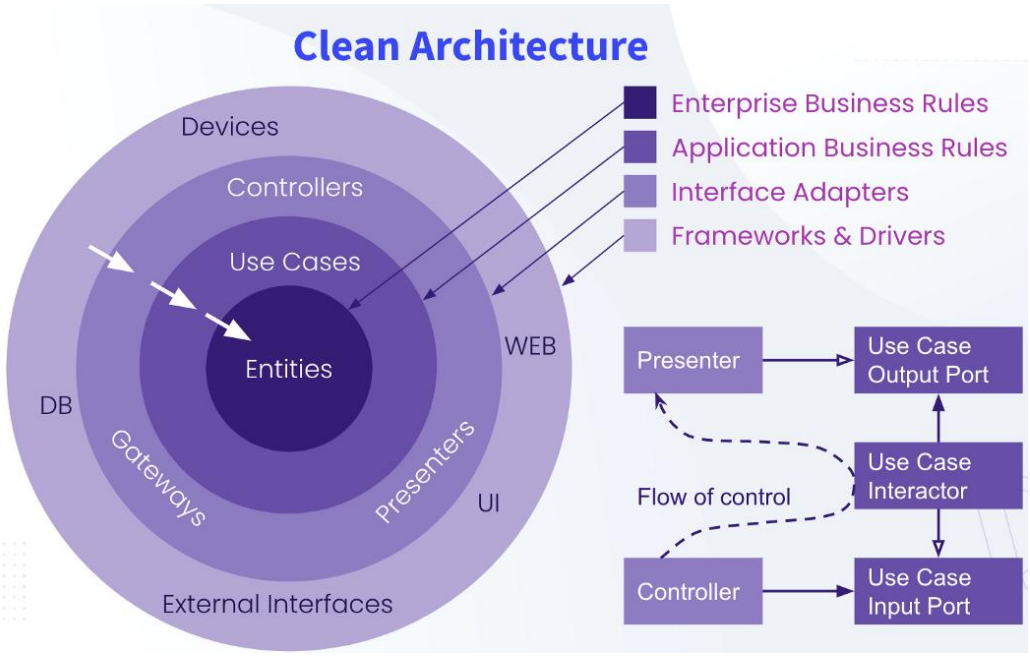


Рисунок 1.7 – Узагальнена схема «чистої архітектури»

1.2.2 Патерн MVC

MVC (Model-View-Controller) – це один із найвідоміших та найширше використовуваних архітектурних патернів. Його основна ідея полягає в тому, щоб розділити відповідальності в застосунку на три окремі, але взаємопов'язані компоненти [13].

Model відповідає за дані та бізнес-логіку. Вона нічого не знає про те, як ці дані будуть відображатися користувачеві, чи як користувач буде з ними взаємодіяти. Її завдання – зберігати дані, обробляти їх, виконувати обчислення, перевіряти правила та повідомляти про зміни свого стану.

View відповідає за відображення даних, отриманих від моделі. Він не містить жодної логіки обробки цих даних, а просто показує їх у зручному для користувача форматі. Може мати різні форми: веб-сторінка, екран мобільного застосунку, вікно програми, тощо.

Controller виступає посередником. Він отримує вхідні дані від користувача (наприклад, натискання кнопки, введення тексту), обробляє їх і вирішує, що робити далі. Контролер може звернутися до Model, щоб оновити

дані або отримати їх, а потім передати ці дані View для відображення. Тобто, контролер реагує на дії користувача та оновлює дані та їх відображення.

1.2.3 Патерн MVP

MVP (Model-View-Presenter), як і MVC, прагне розділити відповідальності в застосунку, але робить це дещо інакше, намагаючись усунути деякі неоднозначності та покращити тестовність, яка іноді стає проблемою в класичному MVC [13].

Model – це все ще постачальник даних та бізнес-логіки. Вона не знає ні про View, ні про Presenter. Її завдання – керувати даними, чи то з бази даних, мережі, чи просто з внутрішньої логіки. Важливо, що Model залишається повністю незалежною від інтерфейсу користувача.

View ж у MVP стає максимально пасивним. Це означає, що він практично не містить жодної логіки, окрім тієї, що стосується безпосереднього відображення елементів інтерфейсу та передачі дій користувача презентеру. View у MVP — це, по суті, набір інструкцій "покажи це тут", "зроби це активним/неактивним". Він не приймає рішень. Усі події користувача (кліки, введення тексту) він просто передає презентеру.

Presenter – це ключова фігура в MVP і головна відмінність від Controller в MVC. Presenter виступає як диригент між Моделлю та Виглядом. Двостороння комунікація з View: Presenter отримує події від View (наприклад, "користувач натиснув кнопку") і, у свою чергу, дає команди Вигляду (наприклад, "покажи індикатор завантаження", "відобрази ці дані у списку"). Це відрізняється від деяких MVC-архітектур, де Контролер може бути менш тісно пов'язаний з конкретним View, або де View сам отримує дані з моделі після сповіщення про зміни.

Наявне повне керування логікою представлення: вся логіка, пов'язана з тим, як дані мають бути підготовлені та коли вони мають бути відображені, знаходиться саме в презентері. Він отримує чисті дані від моделі, обробляє їх

(форматує, фільтрує тощо) і передає View вже у готовому для показу вигляді.

1.2.4 Патерн MVVM

MVVM (Model-View-ViewModel) – це архітектурний патерн що дуже популярний там, де є необхідність відокремити логіку інтерфейсу від бізнес-логіки, і він часто використовується з фреймворками, що підтримують прив'язку даних (data binding) [13].

Model відповідає за дані та бізнес-логіку застосунку. Це може бути доступ до бази даних, робота з API, виконання складних обчислень тощо. Модель нічого не знає про ViewModel чи View. Вона просто надає дані та функціонал для роботи з ними.

View – це те, що бачить і з чим взаємодіє користувач – користувацький інтерфейс (UI). У MVVM View відповідає за відображення даних, отриманих від ViewModel, та за передачу команд користувача (наприклад, натискання кнопок) до ViewModel.

ViewModel виступає посередником між View та моделлю. Вона готує дані з Model для відображення у View. Це може включати форматування даних, об'єднання даних з різних джерел, застосування логіки відображення (наприклад, визначення, чи має бути видимим певний елемент). Вона містить стан та події. Стан – це дані, які View відображає (наприклад, текст, список елементів, прапорець активності кнопки). Події – це дії, які View може ініціювати (наприклад, "Зберегти дані", "Завантажити список"). Вигляд прив'язується до цих властивостей та команд. ViewModel нічого не знає про конкретну реалізацію View. Вона просто надає дані та команди, а View вже сам вирішує, як їх відобразити та як прив'язатися до них. Це робить ViewModel легкою для тестування.

Ключова особливість MVVM – це те, що Вигляд часто використовує механізми прив'язки даних (data binding) для синхронізації з ViewModel. Це означає, що зміни у ViewModel автоматично відображаються у View, без

написання великої кількості шаблонного коду для оновлення UI. Вигляд знає про ViewModel (зазвичай має посилання на неї), але не містить складної логіки відображення чи обробки даних.

1.2.5 Патерн MVI

MVI (Model-View-Intent), – це архітектурний патерн, який набирає популярності, особливо в контексті реактивного програмування та управління станом (state management). Він прагне зробити потік даних в застосунку ще більш передбачуваним та керованим [13].

У MVI Model – це не просто дані та бізнес-логіка, як у MVC/MVP/MVVM. Тут Model уособлює стан користувацького інтерфейсу. Це єдиний, незмінний об'єкт, який містить всю інформацію, необхідну для відображення поточного стану UI. Коли щось змінюється, створюється нова копія стану з оновленими даними.

View відповідає за відображення моделі та за отримання намірів користувача. Він спостерігає за змінами в моделі і оновлює UI відповідно. Коли користувач виконує якусь дію (натискає кнопку, вводить текст), View перетворює цю дію на намір.

Intent – це об'єкт, що представляє намір користувача виконати якусь дію. Наприклад, "оновити дані", "додати елемент до списку", "відкрити деталі". Це просто структура даних, яка описує, що користувач хоче зробити. View створює наміри та передає їх далі для обробки. Intent обробляється компонентом, який можна назвати обробником, або частиною ViewModel, залежно від реалізації. Цей компонент може взаємодіяти з бізнес-логікою (наприклад, завантажити дані з мережі, зберегти в БД) та на основі Intent та поточного стану створює новий стан.

1.2.6 Підсумок аналізу архітектурних патернів

При виборі архітектурного патерну для розробки проєкту, MVVM є найкращим рішенням, що обумовлене низкою його фундаментальних характеристик, які ефективно вирішують проблеми, що виникають при розробці кросплатформних мобільних додатків [1].

Центральною перевагою MVVM є максимальне відокремлення логіки представлення, інкапсульованої у ViewModel, від специфіки користувацького інтерфейсу конкретної платформи. ViewModel функціонує як абстрактний шар, що містить стан представлення та операції над ним, не маючи при цьому прямих залежностей від конкретних UI-компонентів цільових платформ. Такий підхід суттєво підвищує коефіцієнт переносимості коду порівняно, наприклад, з класичним MVC, де Controller часто тісно інтегрований з життєвим циклом платформи, або навіть з MVP, де Presenter, хоч і взаємодіє з View через інтерфейс, все ж передбачає управління ним.

Механізм прив'язки даних (data binding), що є невід'ємною частиною багатьох реалізацій MVVM, додатково підсилює його ефективність у кросплатформенному контексті [2]. Він забезпечує декларативний зв'язок між View та ViewModel, автоматизуючи синхронізацію даних. Зміни у властивостях ViewModel рефлектуються в UI без необхідності написання імперативного коду для оновлення конкретних візуальних елементів. Це не тільки мінімізує обсяг шаблонного коду, специфічного для кожної платформи, але й зменшує ймовірність помилок, пов'язаних з ручним управлінням станом UI. На відміну від MVP, де Presenter зазвичай імперативно викликає методи View для оновлення, декларативна природа прив'язки даних в MVVM спрощує розробку та підтримку інтерфейсів.

Високий ступінь тестованості ViewModel є ще одним критичним аргументом на користь MVVM. Оскільки ViewModel не має залежностей від UI-фреймворків та конкретних елементів інтерфейсу, її логіка може бути всебічно перевірена за допомогою юніт-тестів в ізольованому середовищі. Це є надзвичайно важливим для кросплатформних проєктів, де забезпечення якості на всіх цільових платформах вимагає значних ресурсів. Здатність

ефективно тестувати спільну логіку представлення значно підвищує надійність кінцевого продукту [11].

Чітке розмежування відповідальностей між Model (дані та бізнес-логіка), View (відображення) та ViewModel (логіка представлення та стан) сприяє кращій структуризації проєкту, полегшує його розуміння, модифікацію та паралельну розробку окремих компонентів. У кросплатформенних командах це дозволяє ефективно розподіляти завдання між розробниками, що спеціалізуються на UI для конкретних платформ, та тими, хто працює над спільною ViewModel та моделлю [13].

Зрештою, широка підтримка MVVM у сучасних кросплатформних фреймворках та наявність розвинених екосистем інструментів для його реалізації знижують поріг входження та забезпечують доступ до перевірених архітектурних рішень [10]. Хоча альтернативні патерни, такі як MVI, пропонують строгий односпрямований потік даних і можуть бути переважними для певних сценаріїв зі складним управлінням станом, MVVM часто розглядається як більш збалансований підхід з точки зору співвідношення між складністю впровадження та отримуваними перевагами, особливо щодо переносимості коду та декларативного управління UI в кросплатформенних застосунках.

1.2.7 Патерн Dependency Injection

Dependency Injection (DI) є фундаментальним дизайн-патерном проектування в об'єктно-орієнтованому програмуванні, що реалізує принцип інверсії управління. Основна мета DI полягає у зменшенні зв'язності між програмними компонентами шляхом передачі залежностей об'єкту ззовні, замість того, щоб об'єкт сам створював або шукав свої залежності що робить код більш універсальним та підвищує легкість його перевикористання [14].

Замість того, щоб компонент-споживач інстанціював свої залежності напряму, він декларує потребу в них (через конструктор, методи-сеттери або

інтерфейси). Відповідальність за створення та надання цих залежностей покладається на зовнішній механізм, який "впроваджує" необхідні екземпляри в компонент-споживач [15].

Дуже часто фреймворки надають певні контейнери що відповідають за створення та надання доступу до якихось залежностей, наприклад, репозиторіїв, надаючи розробнику можливість отримати та передати такий об’єкт як залежність в будь-яку частину програми.

1.3 Аналіз існуючих рішень

Аналіз існуючих застосунків перед розробкою власного має фундаментальне значення. Він забезпечує розуміння пропозиції конкурентів, дозволяючи ідентифікувати їхні переваги та недоліки, а також специфіку цільової аудиторі. Таке дослідження сприяє уникненню прямого дублювання функціоналу та визначенню унікальної позиції.

Крім того, аналіз допомагає виявити стандарти та сформовані очікування користувачів щодо функціональності та елементів інтерфейсу. Це є ключовим для створення інтуїтивно зрозумілого та комфортного у використанні продукту.

Вивчення успішних рішень дозволяє ідентифікувати "найкращі практики" у дизайні, функціональності та технічній реалізації, а також відстежити актуальні тренди та інноваційні підходи. Не менш важливим є уникнення потенційних помилок шляхом аналізу недоліків та негативного користувацького досвіду в існуючих продуктах.

Зрештою, такий попередній аналіз сприяє формуванню чіткої унікальної ціннісної пропозиції, визначаючи, чим саме майбутній застосунок буде вигідно відрізнятися та яку специфічну користь він надасть кінцевому користувачеві. Він також слугує джерелом натхнення та генерації нових ідей для функціоналу або вдосконалення існуючих концепцій. Розробка програмного продукту без попереднього дослідження ринку та

конкурентного середовища суттєво збільшує ризик створення неконкурентоспроможного або незатребуваного рішення.

1.3.1 Сервіс cist.nure.ua

cist.nure.ua (рис 1.8) – це сервіс надає актуальний розклад за групою, викладачем або кімнатою. Будь-який додаток що отримує розклад базується на API цього сервісу. Хоча його й можна використовувати напряму, інтерфейс є сильно застарілим та незручним, особливо на мобільних пристроях, фунціонал сильно обмежений та доступний тільки перегляд розкладу.

Тим не менш, даний сервіс є єдиним способом отримати актуальний розклад, тому складно переоцінити його значимість в рамках як даного дипломного проєкту, так і інших застосунків.

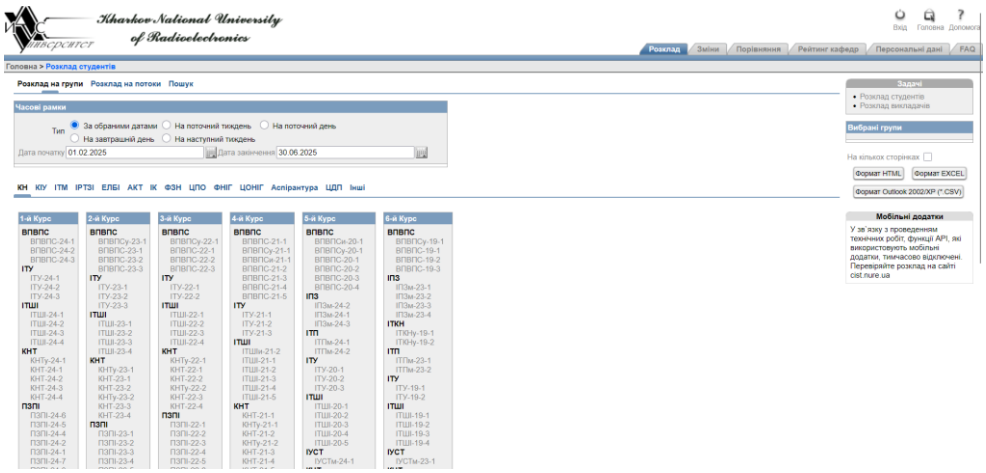


Рисунок 1.8 – Сервіс cist.nure.ua

1.3.2 Застосунок NureTimetable

NureTimetable (рис 1.9) – це мобільний застосунок для ОС Android, створений за допомогою нині застарілої технології Xamarin. Функціонально практично ідентичний до сервісу cist.nure.ua, але є гарно адаптованим для використання на мобільних пристроях за що й отримав популярність. Має

можливість зберігати та автоматично оновлювати розклад що допомагає не тільки завжди мати актуальний розклад, але й переглядати його у разі якщо сервіс cist.nure.ua недоступний.

На жаль, незважаючи на те, що даний застосунок є мобільним, він не надає такого функціоналу, як повідомлення користувачу. Також великою проблемою є той факт що розклади є незмінними, тобто, не можна додати свої події до календаря, що може бути дуже корисним у певних випадках.

Незважаючи на це, даний додаток все ще є чудовим вибором для студентів, який наразі широко використовується. Він став натхненням під час виконання даного проєкту та виправлення його недоліків є важливою метою проєкту.

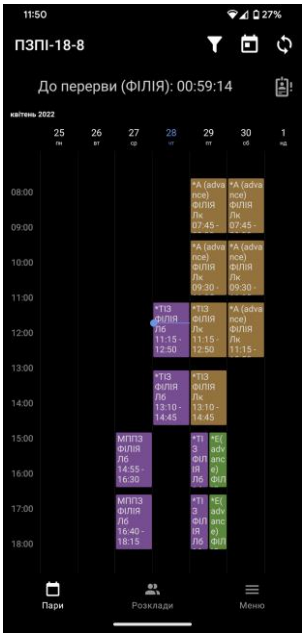


Рисунок 1.9 – Застосунок NureTimetable

1.3.3 Веб-застосунок nure.date

nure.date (рис. 1.10) – це відносно новий застосунок що є інтерактивним календарем. Також використовує сервіс cist.nure.ua для отримання розкладу. Його головною перевагою є можливість додавати свої події як у звичайному календарі та синхронізація даних за допомогою акаунту Google.

Проте, у застосунку є ряд недоліків. По-перше, він є виключно веб-застосунком і керування ним на мобільних пристроях доволі незручне. По-друге, для роботи треба обов’язково авторизуватись у за допомогою акаунту Google, що незручно, так як це необхідно робити кожного разу. Також не має жодних повідомлень.

Загалом через свою незручність саме на мобільних пристроях даний застосунок не набрав популярності, необхідність робити багато зайвих дій при кожному відкритті також є критичним фактором, але можливість додавати власні події є унікальною для застосунків що працюють з сервісом cist.nure.ua і даний функціонал має бути наявним у даному проєкті через свою важливість.

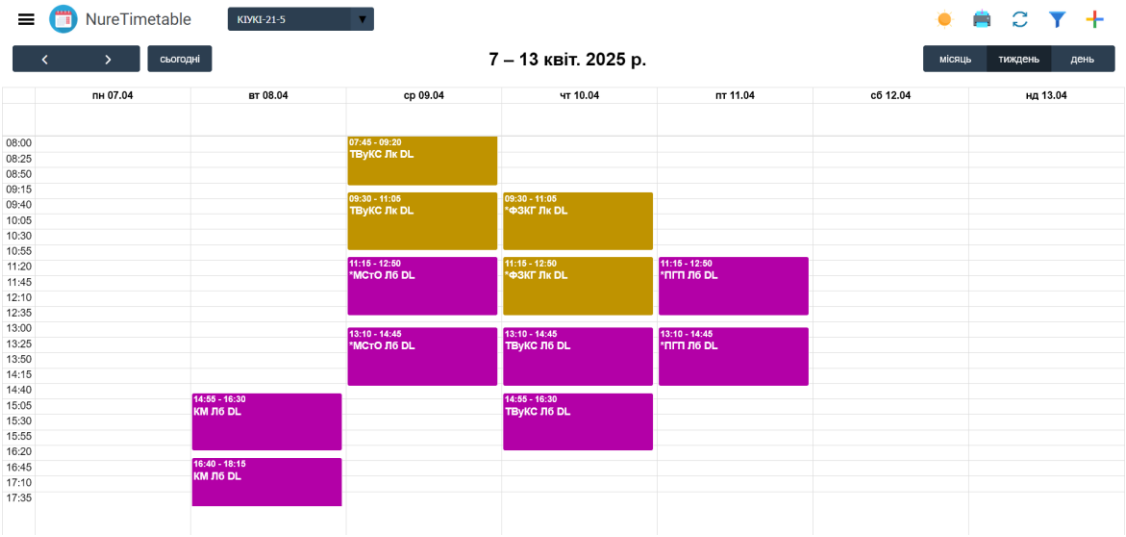


Рисунок 1.10 – Веб-застосунок nure.date

1.3.4 Застосунок Google Календар

Google Календар (рис. 1.11) – це дуже популярний застосунок від компанії Google. Є фактично стандартом інтерактивних календарів. Найбільшою перевагою є гарна інтеграція з іншими сервісами компанії Google, наприклад, поштою, Tasks та Meet. Також наявна синхронізація та підтримка фактично усіх платформ – мобільні застосунки для Android та iOS,

веб-застосунок для інших платформ.

Великим недоліком є відсутність інтеграції з сервісом cist.nure.ua. Розклад занять користувачу доведеться переносити вручну. Незважаючи на можливість імпорту подій до календаря, жодної простої реалізації такого функціоналу наразі не існує.

Застосунок має зручний інтерфейс користувача який можна взяти за основу. Він потребує малої кількості дій для отримання доступу до практично всіх функцій застосунку.

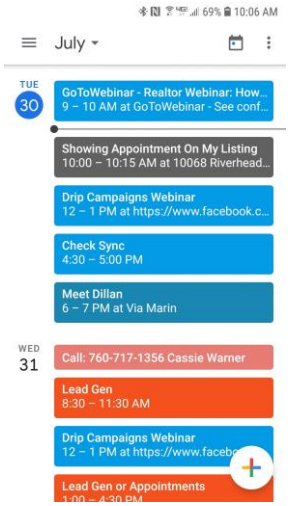


Рисунок 1.11 – Застосунок Google Календар

1.3.5 Застосунок Google Tasks

Google Tasks (рис. 1.12) – це популярний застосунок від Google. Головною перевагою є зручна інтеграція з іншими сервісами Google, зокрема, Google Календарем, але не має жодної інтеграції з сервісом cist.nure.ua, через що студенту доведеться власноруч вносити завдання до застосунку.

Зручному інтерфейс користувача дозволяє з легкістю налаштовувати завдання, він є дуже інтуїтивним для будь-якого користувача, що надихнуло на дизайн даного проєкту.

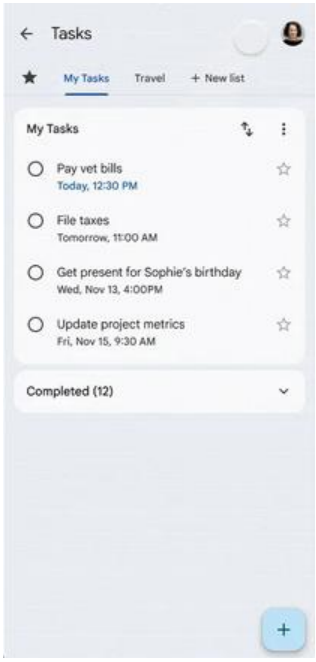


Рисунок 1.12 – Застосунок Google Tasks

1.3.6 Застосунок Microsoft To Do

Microsoft To Do (рис. 1.13) – це застосунок від Microsoft, що є багатофункціональним прикладом реалізації такого сервісу. Дуже схожий з Google Tasks за функціоналом, головна відмінність: замість акаунту Google, використовується акаунт Microsoft.

Має приємний дизайн, але перевантажений певним функціоналом, який більшості людей може ніколи не знадобитись. Також не має інтеграції з сервісами ХНУРЕ, що ускладнює використання його для керування навчанням.

Рисунок 1.13 – Застосунок Microsoft To Do

1.3.7 Застосунок NureTool Legacy

NureTool Legacy (рис. 1.14) – це новий застосунок створений для вирішення проблеми відсутності інтеграції існуючих рішень з сервісами ХНУРЕ. Має простий інтерфейс, який, однак, має місце для багатьох поліпшень, але має дуже малий набір функціоналу та багато невдалих рішень стосовно інтерфейсу користувача. Також немає можливості створення власних завдань, має дуже обмежені налаштування задач, що зазвичай наявний у подібних затосунках.

Знання, які були отримані у ході розробки даного застосунку, а також відгуки користувачів стали основою ідей, реалізованих у рамках цього проєкту.

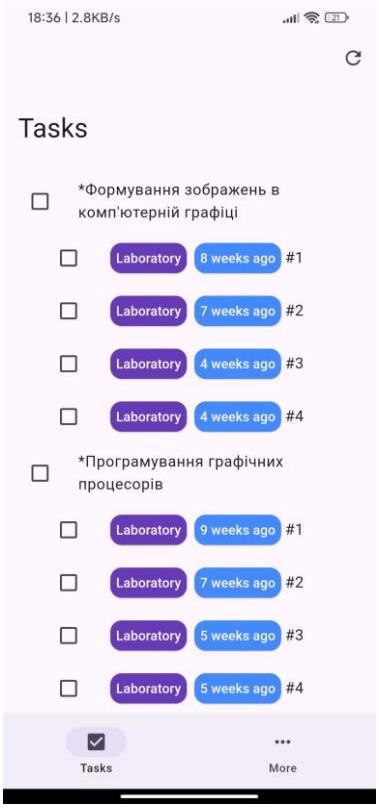


Рисунок 1.14 – Застосунок NureTool Legacy

1.3.8 Висновок аналізу наявних рішень

Загалом рішення можна розбити на дві категорії: створені спеціально для взаємодії з сервісами ХНУРЕ та загальні рішення, не напрямлені на використання конкретною організацію.

Застосунки для ХНУРЕ мають проблему з функціональністю: зазвичай їх функціонал зводиться лише до відображення інформації що надає база даних університету, функціонал для додавання власної інформації до застосунку відсутній або дуже примітивний.

Загальні рішення не мають проблем з функціоналом, але страждають від відсутності інтеграції з сервісами університету. Це призводить до значного ускладнення їх використання оскільки ручне внесення розкладу чи задач є дуже довгим та кропітким процесом що може легко зайняти декілька годин, а неможливість оновлення даних може призвести до непередбачуваних проблем.

Також варто відмітити що відсутні рішення «все в одному». Застосунки що надають функціонал календаря, необхідний для зберігання та відображення розкладу, та застосунки що надають список задач є двома незалежними рішеннями що, частіше за все, є недоліком.

1.4 Постановка задачі

На основі аналізу існуючих рішень можна зробити висновки щодо таких задач для виконання: необхідно зробити застосунок що поєднує функціонал календаря та списку задач щоб зменшити кількість витраченого часу та об’єднати всю необхідну інформацію в одному зручному застосунку. При чому, застосунок має мати інтеграцію з сервісами ХНУРЕ, надаючи користувачу можливість переглянути актуальний розклад чи отримати список задач без зайвих клопіт. Застосунок має надавати функціонал зберігання розкладів декількох груп, викладачів та аудиторій.

Оскільки необхідність дізнатись розклад чи переглянути статус виконання своїх завдань може з’явитись в будь-який момент, головними платформами для подібних застосунків є мобільні платформи: Android та iOS. Іноді користувачу може знадобитись функціонал застосунку коли він працює за комп’ютером. Мати повноцінну програму з функціоналом, що поміщається у кишені – не найпрактичніша ідея. Тому такі застосунки часто мають веб-версії. Отже, цільовою платформою є саме мобільні платформи, також підтримка веб-платформ була б бажаною.

Оскільки розклад та завдання, що стосуються університету, це не єдині події у житті студента, застосунок також має підтримувати створення власних подій. Це позбавить необхідності використовувати інші додатки задля внесення своїх власних повсякденних задач.

Отже, проаналізувавши проблему та засоби її рішення, сформовано такі етапи реалізації поставленої мети:

а) спроєктувати користувацький UI, визначивши кількість сторінок та

їх призначення;

б) визначити формати зберігання даних:

1) критичних даних, таких як розклад та задачі користувача;

2) некритичних даних, таких як налаштування зовнішнього вигляду застосунку;

в) реалізувати логіку для отримання та збереження даних та надавання їх іншим частинам застосунку;

г) реалізувати бізнес-логіку для обробки даних, необхідної в рамках проєкту;

ґ) реалізувати інтерфейс користувача;

д) протестувати роботу застосунку при виконанні реальних задач.

2 АНАЛІЗ ПРОГРАМНИХ ЗАСОБІВ

2.1 Мова програмування Dart

Dart – це об'єктно-орієнтована мова програмування, розроблена компанією Google. Вона була представлена як структурована мова для веб-розробки, маючи на меті потенційно замінити JavaScript у сфері розробки кросплатформенних рішень, вирішуючи проблеми розширюваності, продуктивності та розробки складних застосунків [16]. Синтаксис Dart подібний до Java та C#, що полегшує його освоєння розробниками, знайомими з цими мовами [17]. Dart підтримує строгу типізацію, що означає, що кожна змінна та функція має визначений тип даних, хоча явне визначення типів не завжди є обов'язковим.

Dart надає значні переваги завдяки своїй об'єктно-орієнтованій природі, що сприяє розробці масштабованого та добре структурованого коду, а також можливості компіляції як у JavaScript для веб-застосунків, так і в ефективний нативний код для мобільних та настільних платформ, забезпечуючи високу продуктивність. Вбудована підтримка асинхронного програмування за допомогою async та await істотно спрощує роботу з операціями вводу-виводу та паралельними завданнями, що є критичним для сучасних інтерактивних систем. Ключовою силою Dart є його тісна інтеграція з фреймворком Flutter, що дозволяє створювати кросплатформенні застосунки з єдиної кодової бази з нативною продуктивністю та візуальною привабливістю [17], а його знайомий C-подібний синтаксис полегшує освоєння для розробників з досвідом в інших популярних мовах.

2.2 Система управління базами даних SQLite

Дані про розклад занять є доволі комплексними. Головними є заняття,

які для зручності в рамках розробки проєкту буде названо подіями. У кожної події є час у який вона відбудеться, предмет, до якої ця подія належить, викладачі, які будуть її проводити, список груп студентів, що мають її відвідати та кімната, у якій вона буде проведена. При чому, предмети, викладачі, групи та кімнати не просто текстові дані: кожен з цих елементів сам по собі є об'єктом, який має свій ідентифікатор та параметри.

Зберігання таких взаємопов'язаних даних зазвичай потребує бази даних SQL. Їх основним призначенням є саме збереження пов'язаних між собою певними відношеннями даних, завдяки чому вони ідеально підходять для поставлених задач [18]. Використання SQL бази даних у застосунку дозволить зберігати дані у форматі дуже близькому до того що використовується сервісом що спрощує розробку.

SQLite є реляційною системою управління базами даних, реалізованою у вигляді вбудованої бібліотеки. На відміну від традиційних клієнт-серверних СУБД, таких як MySQL або PostgreSQL, SQLite не вимагає окремого серверного процесу, конфігурації або адміністрування. Вся база даних, включаючи схему, таблиці, індекси та дані, зберігається в єдиному файлі на диску, що забезпечує високу портативність та простоту розгортання [18].

2.3 Бібліотека менеджменту стану Bloc

Бібліотека Bloc (Business Logic Component) є популярним архітектурним шаблоном та пакетом для управління станом у застосунках, розроблених на Dart, особливо в екосистемі Flutter [19]. Основна мета Bloc – відокремити бізнес-логіку від UI, що сприяє покращенню тестування, підтримуваності та масштабованості коду [20].

Бібліотека Bloc функціонує на принципі обробки вхідних подій (Events), які представляють дії користувача або системи, спеціальними компонентами, що містять бізнес-логіку, – блоками, або їх спрощеною

версією – к'юбітами. У відповідь на ці події, блок або к'юбіт виконує необхідні операції та випромінює нові незмінні стани, які відображають поточний стан частини застосунку. Ці стани передаються до UI через потоки, на які UI підписується, дозволяючи йому реактивно перебудовуватися при кожній зміні стану. Таким чином, Bloc забезпечує чіткий односпрямований потік даних та відокремлює логіку від відображення [21], що є його ключовою особливістю (рис. 2.1).

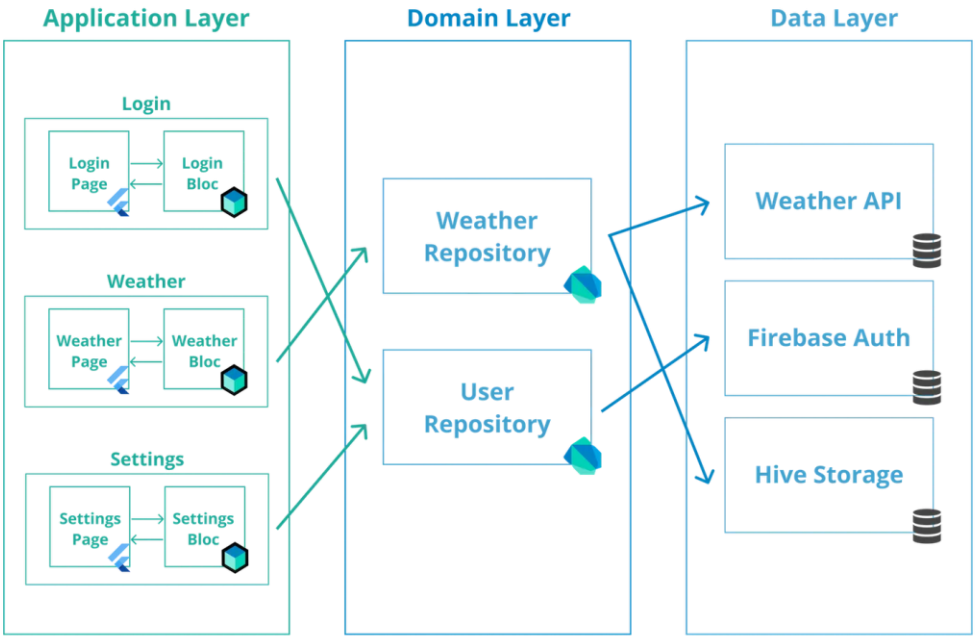


Рисунок 2.1 – Приклад архітектури додатку, створеного за допомогою бібліотеки Bloc

2.4 Бібліотека Drift

Для роботи з SQLite різні мови програмування використовують бібліотеки. Вони відрізняються здебільшого функціоналом: якісь є лише мінімальним інтерфейсом для взаємодії з базою даних, інші ж надають додатковий функціонал для, наприклад, перевірки типів. Серед рішень для мови програмування Dart можна виділити три: sqlite3, sqflite та Drift. З них усіх саме остання є найцікавішою для розробника.

Drift (раніше відомий як Moor) є реактивною бібліотекою для Dart та Flutter, розробленою для ефективної роботи з локальними базами даних SQLite [22]. Вона надає розробникам потужний інструментарій для взаємодії з базами даних, комбінуючи гнучкість SQL з безпекою типів Dart. Drift дозволяє визначати схему бази даних за допомогою Dart-класів, генеруючи на їх основі код для виконання запитів, вставки, оновлення та видалення даних. Ключовою особливістю є компіляційна перевірка SQL-запитів, що дозволяє виявляти помилки на етапі розробки, а не під час виконання програми, на відміну від згаданих вище альтернатив.

Також важливою особливістю Drift є надання можливості використання підходу реактивного програмування для розробників, що є особливо актуальним при розробці клієнтських застосунків [23].

2.5 Бібліотека shared_preferences

Бібліотека shared_preferences є інструментом для збереження простих даних у Flutter, надаючи легкий спосіб управління налаштуваннями програми та іншими даними на стороні клієнта у вигляді пар "ключ-значення" [24].

Основне призначення shared_preferences полягає у зберіганні невеликих обсягів даних, таких як налаштування користувача (наприклад, тема оформлення, мова), стан авторизації, прості прапорці або токени сесії. Вона не призначена для зберігання великих наборів даних або складних реляційних даних, але дозволяє легко читати та записувати дані за допомогою декількох рядків коду. Підтримуються базові типи даних: int, double, bool, String та List<String>, але якщо є можливість серіалізації то зберігати можна практично все.

Дані, збережені за допомогою shared_preferences, залишаються доступними навіть після закриття та повторного запуску програми. Однак, при видаленні даних програми або деінсталяції програми, збережені дані будуть втрачені. Операції читання та запису даних є асинхронними, що

запобігає блокуванню основного потоку інтерфейсу користувача. Важливо зазначити, що немає гарантії негайного запису даних на диск після виконання операції, тому цей плагін не слід використовувати для зберігання критично важливих даних.

2.6 Бібліотека RxDart

RxDart є бібліотекою для мови Dart, яка суттєво розширює стандартні можливості роботи з потоками даних, керуючись принципами реактивного програмування [25]. Вона надає розробникам потужні інструменти для ефективного управління асинхронними потоками даних та подіями, що в результаті призводить до більш читабельного, структурованого та легкого в керуванні коду, особливо при роботі зі складними асинхронними операціями.

В основі RxDart лежить парадигма реактивного програмування, де центральне місце займають потоки даних та їхні трансформації. Замість безпосередньої роботи зі змінами даних, розробники оперують потоками, що значно спрощує обробку асинхронних подій. Важливо зазначити, що RxDart не замінює стандартні потоки Dart, а доповнює та розширює їх функціональність, вводячи нові класи потоків, відомі як Observables, та велику кількість операторів. Observables в RxDart, по суті, є потоками з розширеними можливостями.

Однією з ключових переваг RxDart є багатий набір операторів. Ці оператори дозволяють трансформувати, фільтрувати, комбінувати та керувати потоками даних різноманітними способами. Наприклад, оператор map дозволяє перетворювати дані в потоці, filter – відфільтровувати непотрібні елементи, debounceTime – ігнорувати події, що відбуваються занадто часто, а combineLatest – об'єднувати декілька потоків в один.

RxDart також вводить спеціальні типи потоків, які називаються Subjects, наприклад, BehaviorSubject, PublishSubject та ReplaySubject. BehaviorSubject, наприклад, має корисну властивість зберігати останнє

значення, що надійшло в потік, і віддавати його новим підписникам. Subjects можуть одночасно виступати і як джерело подій, і як потік для їх прослуховування, що робить їх надзвичайно корисними для управління станом додатку [25].

Бібліотека чудово підходить для роботи з асинхронними операціями, такими як запити до мережі, взаємодія з базою даних або обробка користувацьких подій. Хоча RxDart не є виключно бібліотекою для управління станом, її інструменти, зокрема Subjects та оператори, часто використовуються для реалізації складних сценаріїв управління станом у Flutter-додатках. RxDart добре інтегрується з Flutter, дозволяючи створювати реактивні користувацькі інтерфейси з мінімальними зусиллями.

Використання RxDart приносить значні переваги, такі як спрощення роботи з асинхронним кодом завдяки потужним операторам, покращення читабельності коду через реактивний підхід, зменшення кількості шаблонного коду та надання гнучкості й потужності для маніпулювання потоками даних.

2.7 Інші бібліотеки

equatable – це бібліотека, яка спрощує реалізацію порівняння об'єктів на рівність та генерацію hashCode в Dart. Замість того, щоб вручну перевизначати ці методи для кожного класу, достатньо успадкувати клас від Equatable та вказати список властивостей, які повинні враховуватися при порівнянні. Це допомагає уникнути помилок та зменшити кількість шаблонного коду, роблячи класи більш чистими та легкими для тестування, особливо при роботі з незмінними об'єктами та в управлінні станом [26].

json_serializable – це генератор коду, який автоматизує процес перетворення об'єктів Dart у JSON-формат (серіалізація) та навпаки – з JSON у об'єкти Dart (десеріалізація). Розробник визначає структуру класу та додає анотації, а json_serializable генерує необхідний код для методів toJson() та

fromJson(). Це значно зменшує рутинну роботу, мінімізує ймовірність помилок при ручному написанні логіки серіалізації/десеріалізації та полегшує роботу з мережевими запитами та локальним зберіганням даних у форматі JSON [27].

2.8 Середовище розробки Visual Studio Code

Visual Studio Code (рис.2.2) – це безкоштовний редактор коду та, практично, середовище розробки, розроблений компанією Microsoft. Він поєднує в собі простоту легкового текстового редактора з потужними інструментами розробки, такими як налагодження, інтеграція з системами контролю версій, автодоповнення коду, підсвічування синтаксису та можливість розширення функціональності за допомогою великої екосистеми розширень. VSCode побудований на базі фреймворку Electron, що дозволяє йому працювати на операційних системах Windows, macOS та Linux.

Цей редактор має чудову підтримку мови програмування Dart та фремйворку Flutter завдяки великій кількості якісних розширень а також інтеграції інструментів для встановлення, оновлення та роботи з інструментами розробника Dart/Flutter [28].

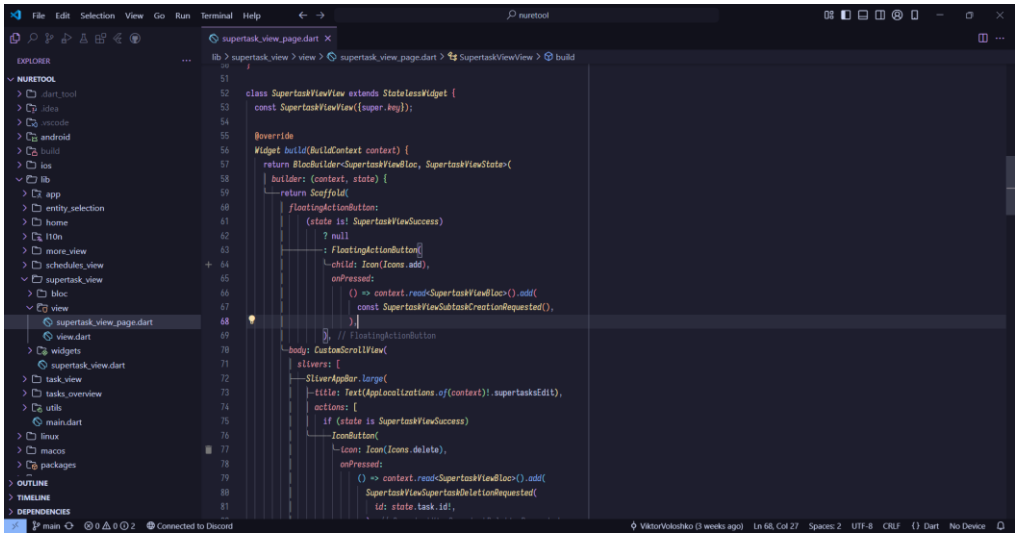


Рисунок 2.2 – Зовнішній вигляд середовища розробки VSCode

3 ОПИС ВИКОНАННЯ РОБОТИ

3.1 Data Layer

Сенс Data Layer в архітектурі додатку полягає в створенні абстракції та централізації всієї логіки, що стосується отримання, зберігання, управління та перетворення інформації.

Цей шар приховує від інших частин програми, таких як бізнес-логіка чи інтерфейс користувача, конкретні джерела даних, будь то мережевий API, локальна база даних, файли або кеш. Замість цього, він надає єдиний та уніфікований інтерфейс для доступу до необхідної інформації. Уся специфічна логіка, що включає виконання запитів, розбір відповідей, обробку помилок підключення, кешування або міграції бази даних, зосереджується саме в цьому шарі.

Такий підхід забезпечує незалежність: зміни у способі зберігання або отримання даних, наприклад, перехід на інший API або іншу базу даних, впливатимуть лише на Data Layer, не зачіпаючи решту кодової бази застосунку. Це, в свою чергу, значно покращує тестованість, оскільки логіку роботи з даними можна перевіряти ізольовано, а для тестування інших компонентів системи реалізацію шару даних легко замінити на тестові заготовки.

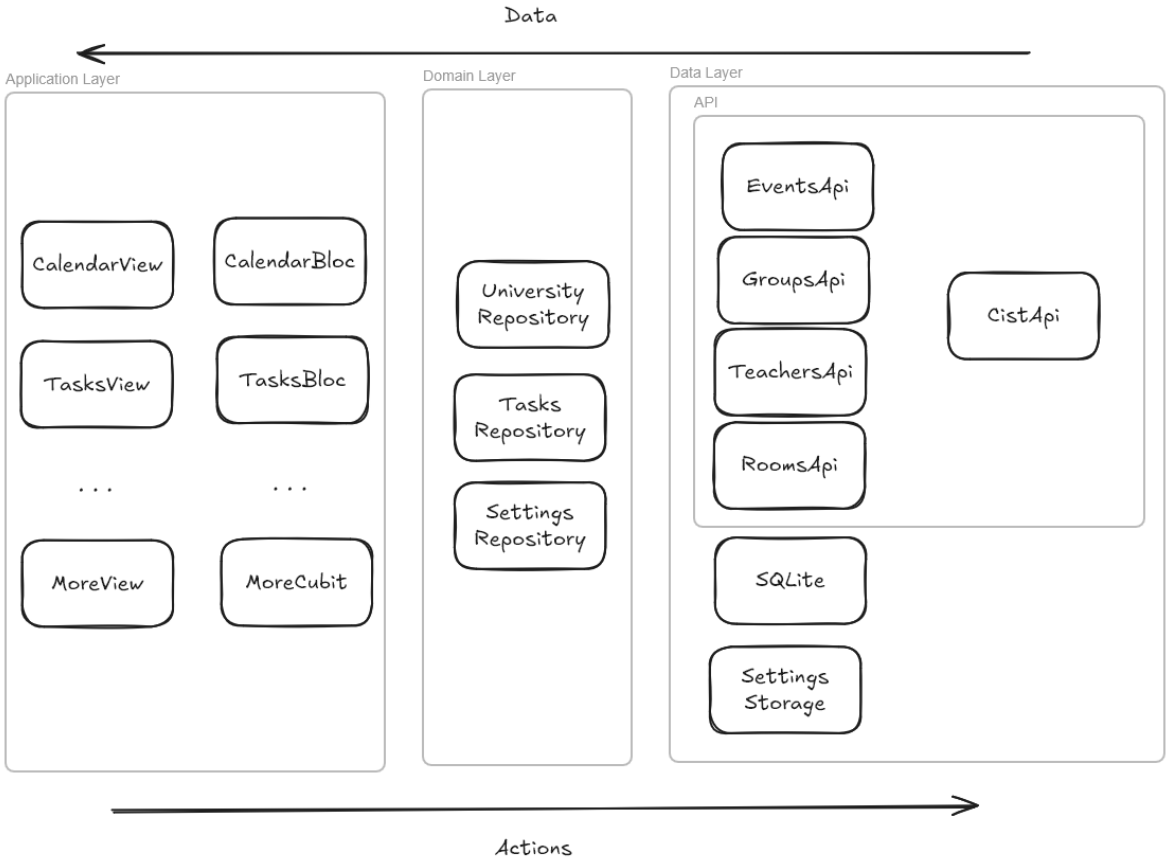


Рисунок 3.1 – Архітектура застосунку

3.1.1 API-клієнт cist.nure.ua

Клієнт розбито на дві частини: інтерфейси для груп, викладачів, кімнат та розкладу та конкретну реалізацію цих інтерфейсів. Це зроблено задля уникнення тісної прив’язки до реалізації, яка може змінюватись. До того ж можна використовувати будь-який клас за умови що він буде реалізовувати всі необхідні інтерфейси.

GroupsApi (лістинг 3.1) є абстрактним інтерфейсом, який визначає контракт для отримання даних про групи. Він оголошує метод fetchGroups(), що повертає Future<List<Faculty>>. Це означає, що будь-який клас, який реалізує цей інтерфейс, зобов'язаний надати спосіб асинхронного отримання списку факультетів, які, в свою чергу, рекурсивно містять інформацію про групи. Сам інтерфейс не містить логіки отримання даних, а лише описує, що повинно бути зроблено.

Лістинг 3.1 – Інтерфейс GroupsApi

```
abstract interface class GroupsApi {
  const GroupsApi();

  Future<List<Faculty>> fetchGroups();
}

class GroupsNotFoundFailure implements Exception {}
```

TeachersApi є абстрактним інтерфейсом, аналогічним GroupsApi, але призначеним для роботи з даними про викладачів. Він визначає контракт, вимагаючи реалізації методу fetchTeachers(), який асинхронно повертає список факультетів, що містять інформацію про викладачів.

RoomsApi – це ще один абстрактний інтерфейс, що визначає контракт для отримання інформації про аудиторії. Він вимагає реалізації методу fetchRooms(), який асинхронно повертає список будівель, що рекурсивно містять дані про всі аудиторії.

GroupsRequestFailure, TeachersRequestFailure та RoomsRequestFailure – це класи винятку, що успадковується від Exception. Вони призначені для сповіщення про невдачу під час виконання запиту на отримання даних про відповідні об’єкти або під час їх декодування. Використання специфічних винятків дозволяє більш точно обробляти помилки, пов'язані саме з операціями над групами.

GroupsNotFoundFailure, TeachersNotFoundFailure та RoomsNotFoundFailure також є класами винятку. Вони використовуються для позначення ситуації, коли API успішно обробив запит, але не повернув порожні дані. Так як така ситуація вважається практично неможливою, виникнення цього винятку може свідчити про неочевидну проблему, яка не була виявлена на попередніх етапах обробки відповіді.

EventsApi (лістинг 3.2) є абстрактним інтерфейсом, який визначає контракт для отримання даних, пов'язаних з подіями (наприклад, розкладом занять). Він не містить конкретної реалізації, а лише описує, які методи має

надати клас, що його реалізовуватиме.

Лістинг 3.2 – Інтерфейс EventsApi

```
import 'models/models.dart';

abstract interface class EventsApi {
  const EventsApi();

  Future<({List<Event> events, List<Subject> subjects,
List<EventType> types})>
  fetchEventsForGroup(int groupID, int fromTimestamp, int
toTimestamp);

  Future<({List<Event> events, List<Subject> subjects,
List<EventType> types})>
  fetchEventsForTeacher(int teacherID, int fromTimestamp, int
toTimestamp);

  Future<({List<Event> events, List<Subject> subjects,
List<EventType> types})>
  fetchEventsForRoom(int roomID, int fromTimestamp, int
toTimestamp);
}

class EventsRequestFailure implements Exception {}
```

Методи fetchEventsForGroup, fetchEventsForTeacher, та fetchEventsForRoom, призначені для асинхронного завантаження подій для конкретної групи, викладача чи аудиторії. Вони приймають ідентифікатор об’єкта, а також початкову та кінцеву часові мітки у форматі Unix timestamp, які визначають період, за який потрібно отримати події. Методи повертає Future, що по завершенні міститиме іменований кортеж з трьома списками: events (список об'єктів Event), subjects (список об'єктів Subject) та types (список об'єктів EventType).

Клас EventsRequestFailure успадковується від Exception і використовується для сповіщення про помилку, що сталася під час виконання запиту на отримання даних про події або під час декодування відповіді від сервера. Використання такого спеціалізованого класу винятку дозволяє більш точно обробляти помилки, специфічні саме для операцій, пов'язаних із завантаженням подій.

CistApi є конкретним класом, який реалізує всі вищезгадані абстрактні інтерфейси: GroupsApi (як groups_api.GroupsApi), TeachersApi (як teachers_api.TeachersApi), RoomsApi, а також EventsApi. Цей клас відповідає за фактичну взаємодію із зовнішнім API cist.nure.ua. Він використовує бібліотеку Dio для виконання HTTP-запитів. У конструкторі CistApi ініціалізується екземпляр Dio з налаштуваннями, що включають тип відповіді та декодер для кодування Windows-1251.

Кожен з методів (fetchGroups, fetchTeachers, fetchRooms, fetchEventsForGroup, fetchEventsForTeacher, fetchEventsForRoom) в CistApi реалізує логіку відповідного запиту до API: формує URL, відправляє запит, перевіряє статус відповіді, декодує JSON-дані (з деякими специфічними обробками, як-от виправлення формату JSON для даних викладачів), перетворює їх у відповідні об'єкти моделей (наприклад, Faculty, Building, Event) та обробляє можливі помилки, викидаючи відповідні специфічні винятки (лістинг 3.3).

Лістинг 3.3 – Обробка отриманого JSON з розкладом для викладача

```
final Map<String, dynamic> json;
    try {
      json =
          jsonDecode(
                response.data!.replaceRange(
                  response.data!.length - 2,
                  null,
                  ']}}',
                ),
              )
              as Map<String, dynamic>;
    } catch (_) {
      throw teachers_api.TeachersRequestFailure();
    }
```

Таким чином, CistApi інкапсулює всю складність взаємодії з конкретним зовнішнім сервісом, надаючи чисті інтерфейси для використання в інших частинах програми.

3.1.2 Клієнт взаємодії з базою даних SQLite

Для зберігання, оновлення та отримання збережених даних необхідний клієнт для взаємодії з нею. Бібліотека Drift сильно спрощує створення такого клієнту. Також, оскільки над цими даними у розробника є повний контроль, інтерфейси відсутні і клієнт бази даних стане прямою залежністю для інших елементів.

Бібліотека Drift надає можливість генерації SQL-запитів, у тому числі тих, що будуть створювати таблицю та генерацію DTO-класів. Все що необхідно – це створити класи що будуть описувати таблиці у базі даних.

Клас Tasks (лістинг 3.4) є визначенням таблиці для зберігання інформації про завдання. Поряд з класом Tasks визначено перерахування (enum) TaskType. Це перерахування визначає набір можливих типів завдань. Це перерахування буде використовуватися для одного з полів таблиці, щоб класифікувати завдання.

Стовпець id визначається як IntColumn. Це цілочисельний стовпець, який автоматично збільшує своє значення (autoIncrement()) при додаванні нового запису, що робить його ідеальним кандидатом на роль первинного ключа таблиці.

Стовпець title є TextColumn. Він призначений для зберігання назви завдання. Стовпець isDone визначається як BoolColumn. Це булевий стовпець, який вказує на стан виконання завдання (виконано чи ні). Важливою особливістю є clientDefault(() => false), що означає, що якщо при створенні нового завдання значення для цього поля не надано, воно за замовчуванням буде встановлено в false (не виконано) на стороні клієнта (тобто в Dart коді перед записом до БД). Стовпець isGenerated також є BoolColumn. Цей булевий стовпець, використовується для позначення, чи було завдання створено автоматично (згенеровано програмою) або додано користувачем вручну.

Стовпець supertaskID визначається як IntColumn з назвою в базі даних

supertask_id (завдяки named('supertask_id')). Він є цілочисельним і може приймати значення null (nullable()). Це поле використовується для реалізації ієрархічної структури завдань, вказуючи на ідентифікатор "батьківського" завдання, якщо поточне завдання є підзавданням.

Стовпець deadline є DateTimeColumn. Він призначений для зберігання дати та часу, до якого завдання має бути виконане. Цей стовпець також може бути порожнім (nullable()), що означає, що не для всіх завдань може бути встановлений дедлайн. Drift конвертує DateTime об’єкти у Integer для зберігання у форматі Unix timestamp.

Стовпець type визначається як IntColumn, але з особливістю intEnum<TaskType>(). Це означає, що в базі даних цей стовпець буде зберігати ціле число, але на рівні Dart коду він буде автоматично перетворюватися на значення з перерахування TaskType і навпаки. Це дозволяє зберігати тип завдання в компактному числовому вигляді, маючи при цьому типізований доступ в коді. Цей стовпець також може бути порожнім (nullable()).

Лістинг 3.4 – Клас таблиці з задачами

```
enum TaskType { subject, project, practice, laboratory, test,
exam }

class Tasks extends Table {
  IntColumn get id => integer().autoIncrement()();
  TextColumn get title => text()();
  BoolColumn get isDone => boolean().clientDefault(() =>
false)();
  BoolColumn get isGenerated => boolean()();
  IntColumn get supertaskID =>
integer().named('supertask_id').nullable()();
  DateTimeColumn get deadline => dateTime().nullable()();
  IntColumn get type => intEnum<TaskType>().nullable()();
}
```

Таку ж таблицю можна зробити єдиним SQL-запитом і використовувати без жодних відмінностей (лістинг 3.5). Але завдяки Drift не потрібно писати запити вручну та вручну їх викликати, що особливо корисно

враховуючи відсутність можливості їх статичної перевірки.

Лістинг 3.5 – Створення таблиці tasks SQL-запитом

```
CREATE TABLE tasks (
    id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    title TEXT NOT NULL,
    is_done BOOLEAN NOT NULL DEFAULT 0,
    is_generated BOOLEAN NOT NULL,
    supertask_id INTEGER,
    deadline INTEGER,
    type INTEGER
);
```

Оскільки дані про подію мають містити many-to-many відносини (між групами та подіями для цієї групи), необхідно було обрати спосіб зберігання таких відношень. Для SQLite найпростішим та сучасним є зберігання їх у об’єкті що буде серіалізуватись у JSON та записуватись в базу даних (лістинг 3.6).

@JsonSerializable(): Ця анотація від json_serializable вказує, що для цього класу потрібно автоматично генерувати код для серіалізації (перетворення об'єкта в JSON) та десеріалізації (створення об'єкта з JSON).

final List<int> groups; та final List<int> teachers;: Ці поля зберігають списки цілочисельних ідентифікаторів. Саме ці поля будуть представлені як масиви чисел у JSON під відповідними ключами ("groups" та "teachers").

factory EventRelations.fromJson(Map<String, dynamic> json) => _$EventRelationsFromJson(json);: Цей статичний конструктор використовується для створення екземпляра EventRelations з JSON-об'єкта (представленого як Map<String, dynamic>). Згенерований код _$EventRelationsFromJson візьме значення з ключів "groups" та "teachers" у JSON і присвоїть їх відповідним полям об'єкта.

Map<String, dynamic> toJson() => _$EventRelationsToJson(this);: Цей метод перетворює екземпляр EventRelations на Map<String, dynamic>, який потім може бути легко перетворений на рядок JSON. Згенерований код _$EventRelationsToJson створить словник з ключами "groups" та "teachers" та

відповідними списками ідентифікаторів як їхніми значеннями.

converter = TypeConverter.json2(...) визначає TypeConverter для Drift. Він дозволяє зберігати об'єкт EventRelations у базі даних у вигляді JSON-рядка в одному стовпці та автоматично перетворювати його назад в об'єкт EventRelations при читанні з бази даних. fromJson та toJson всередині конвертера використовують ті ж самі методи, що й для прямої серіалізації/десеріалізації JSON.

Лістинг 3.6 – Клас для зберігання відношень подій

```
@immutable
@JsonSerializable()
class EventRelations {
  const EventRelations({required this.groups, required
this.teachers});

  final List<int> groups;
  final List<int> teachers;

  factory EventRelations.fromJson(Map<String, dynamic> json) =>
      _$EventRelationsFromJson(json);

  Map<String, dynamic> toJson() => _$EventRelationsToJson(this);

  static final converter = TypeConverter.json2(
    fromJson: (json) => EventRelations.fromJson(json as
Map<String, dynamic>),
    toJson: (column) => column.toJson(),
  );
}
```

Застосунок може мати більш ніж одну базу даних. Кожна з баз даних має мати свій клас. Він визначає її місцезнаходження, параметри створення, редагування та з’єднання та методи що будуть публічними, окрім стандартних. Тут же вказується які саме таблиці належать цій базі.

Клас надає функції що використовують watch() для спостереження за таблицею та надання всіх даних з неї у потоці. Це дозволяє підписатись на потік та отримувати завжди актуальні дані, прив’язавши певну дію до кожного оновлення таблиці.

Найцікавішою є функція запиту на отримання списку подій (лістинг

3.7) оскільки дані кожної окремої події супроводжуються даними про їх тип. Для того, щоб отримати їх одним об’єктом використовується приєднання таблиці.

Лістинг 3.7 – Функція-запит на отримання даних про події

```
Stream<List<EventData>> loadEvents() {
    final query = select(
      events,
    ).join([leftOuterJoin(eventTypes,
eventTypes.id.equalsExp(events.typeID))]);

    return query.watch().map((rows) {
      return rows
          .map(
            (row) => EventData(
              event: row.readTable(events),
              type: row.readTableOrNull(eventTypes),
            ),
          )
          .toList();
    });
  }
```

Метод loadEvents повертає Stream<List<EventData>>. Це означає, що він не просто один раз завантажує дані, а створює потік (stream). Цей потік буде автоматично видавати новий список об'єктів EventData щоразу, коли відбудуться зміни у відповідних таблицях (events або eventTypes), які впливають на результат запиту. EventData – це, спеціально створений клас або кортеж для зручного представлення об'єднаних даних про подію та її тип.

Всередині методу спочатку формується запит до бази даних. select() та join() створює цей запит.

Частина select(events) вказує, що ми хочемо вибрати дані з таблиці events (імовірно, events – це об'єкт, що представляє таблицю подій у схемі Drift).

Далі, join() додає операцію об'єднання до запиту. Тут використовується leftOuterJoin для об'єднання таблиці events з таблицею eventTypes (імовірно,

eventTypes – це об'єкт, що представляє таблицю типів подій). Умова об'єднання eventTypes.id.equalsExp(events.typeID) вказує, що рядки з цих двох таблиць повинні бути об'єднані, якщо значення стовпця id у таблиці eventTypes співпадає зі значенням стовпця typeID у таблиці events. LEFT OUTER JOIN гарантує, що всі записи з таблиці events будуть включені до результату; якщо для події не знайдено відповідного типу в eventTypes, поля з eventTypes для цього запису будуть null.

Після формування запиту, return query.watch().map((rows) { ... }); запускає його виконання та обробку результатів. Метод query.watch() є ключовим для реактивності. Він виконує запит і повертає потік, який буде оновлюватися при змінах даних. Оператор .map((rows) { ... }) застосовується до кожного списку рядків (rows), який надходить з потоку query.watch(). Кожен елемент rows – це список сирих результатів запиту з бази даних, де кожен рядок містить дані з об'єднаних таблиць.

Всередині цього map відбувається перетворення списку сирих рядків на список об'єктів EventData. rows.map((row) => EventData(event: row.readTable(events), type: row.readTableOrNull(eventTypes))) ітерує по кожному сирому рядку (row) і для кожного створює екземпляр EventData.

Поле event об'єкта EventData заповнюється даними з таблиці events за допомогою row.readTable(events). Поле type об'єкта EventData заповнюється даними з таблиці eventTypes за допомогою row.readTableOrNull(eventTypes). Використання readTableOrNull тут важливе, оскільки через LEFT OUTER JOIN відповідного типу події може не бути, і в такому випадку поле type отримає значення null. Нарешті, .toList() перетворює результат внутрішнього map (який є Iterable об'єктом) назад у список List<EventData>.

3.1.3 Збереження налаштувань

Для збереження таких некритичних даних як налаштування застосунку немає сенсу використовувати реляційну базу даних. Кращим вибором буде

просте сховище яке надає базовий функціонал запису та отримання даних. Так як формат таких даних буде простим, немає сенсу перейматися про обмеження які накладатиме таке сховище.

Клас ScheduleData призначений для зберігання ідентифікаційної інформації про збережений розклад. Він використовує анотацію @JsonSerializable() для автоматичної генерації коду, необхідного для перетворення об'єктів цього класу в JSON-формат та навпаки, що й використовується для збереження даних.

Поряд із класом ScheduleData (лістинг 3.8) визначено перерахування ScheduleType. Це перерахування визначає можливі типи розкладів, які можуть бути збережені: group (розклад для групи), teacher (розклад для викладача) та room (розклад для аудиторії).

Поле id типу int зберігає унікальний ідентифікатор конкретного розкладу. Наприклад, це може бути ID групи, ID викладача або ID аудиторії, для якої збережено розклад.

Поле type типу ScheduleType зберігає тип розкладу, використовуючи значення з перерахування ScheduleType. Це дозволяє розрізняти, до якої сутності (групи, викладача чи аудиторії) належить збережений розклад з певним id.

Статичний конструктор factory ScheduleData.fromJson(Map<String, dynamic> json) => _$ScheduleDataFromJson(json); відповідає за створення об'єкта ScheduleData з JSON-представлення (у вигляді Map<String, dynamic>). Згенерований код _$ScheduleDataFromJson виконає необхідні перетворення.

Метод Map<String, dynamic> toJson() => _$ScheduleDataToJson(this); перетворює екземпляр ScheduleData на Map<String, dynamic>, який потім може бути легко серіалізований у JSON-рядок. Згенерований код _$ScheduleDataToJson виконає це перетворення.

Таким чином, клас ScheduleData слугує простою та ефективною структурою для представлення метаданих про збережений розклад, вказуючи на його унікальний ідентифікатор та тип, що дозволяє легко ідентифікувати

та розрізняти різні збережені розклади.

Лістинг 3.8 – Клас ScheduleData для збереження даних про розклади

```
@immutable
@JsonSerializable()
class ScheduleData extends Equatable {
  const ScheduleData({required this.id, required this.type});

  factory ScheduleData.fromJson(Map<String, dynamic> json) =>
      _$ScheduleDataFromJson(json);

  final int id;
  final ScheduleType type;

  Map<String, dynamic> toJson() => _$ScheduleDataToJson(this);

  @override
  List<Object?> get props => [id, type];
}

enum ScheduleType { group, teacher, room }
```

За зберігання даних відповідає клас SettingsStorage. Клас SettingsStorage призначений для управління налаштуваннями додатку, забезпечуючи їхнє персистентне зберігання та реактивний доступ до них. Він інкапсулює логіку читання, запису та сповіщення про зміни налаштувань.

Він залежить від SharedPreferencesWithCache (_storage), який, є обгорткою над стандартним shared_preferences з додаванням кешування для швидшого доступу. Він передається за патерном DI.

Ключовою особливістю класу є використання BehaviorSubject з бібліотеки RxDart для кожного налаштування. BehaviorSubject – це спеціальний тип потоку, який зберігає останнє отримане значення і віддає його будь-якому новому підписнику, а також сповіщає підписників про нові значення. У класі визначено три таких контролери:

- _userGroupIDStreamController для зберігання та трансляції ідентифікатора групи користувача (int?);

- _savedSchedulesStreamController для збережених розкладів (тип SavedSchedules, який, є спеціальним класом, що підтримує серіалізацію в

JSON);

- _appThemeStreamController для теми оформлення додатку (тип AppTheme, також серіалізований клас).

Для зовнішнього доступу до значень цих налаштувань у реактивному стилі надаються відповідні публічні потоки (streams): userGroupID, savedSchedules та appTheme. Вони є широкомовними (asBroadcastStream()), що дозволяє мати декілька слухачів для кожного потоку налаштувань.

Для зберігання даних у SharedPreferences використовуються константні ключі: kUserGroupID, kSavedSchedules та kAppTheme. Анотація @visibleForTesting вказує, що ці константи можуть використовуватися в тестах.

Методи setUserGroupID, setSavedSchedules та setTheme слугують для зміни відповідних налаштувань. Кожен такий метод спочатку додає нове значення у відповідний BehaviorSubject (що сповіщає всіх підписників потоку), а потім асинхронно зберігає це значення у _storage. Для складних типів, як SavedSchedules та AppTheme, перед збереженням відбувається їх серіалізація в JSON-рядок за допомогою json.encode().

Метод _init(), що викликається в конструкторі, відповідає за початкову ініціалізацію контролерів потоків. Він намагається прочитати збережені значення для кожного налаштування з _storage.

Для _userGroupIDStreamController він просто зчитує ціле число. Для _savedSchedulesStreamController та _appThemeStreamController він спочатку намагається отримати JSON-рядок. Якщо рядок існує, він десеріалізується (json.decode()) у відповідний об'єкт (SavedSchedules.fromJson або AppTheme.fromJson). Якщо ж дані для цього налаштування ще не збережені (рядок null), то контролер ініціалізується значенням за замовчуванням (SavedSchedules.empty() або const AppTheme.defaultValues()), і це значення за замовчуванням одразу ж зберігається у _storage. Це гарантує, що потоки завжди мають початкове значення.

Метод close() призначений для коректного завершення роботи зі

сховищем налаштувань. Він закриває всі контролери потоків (_userGroupIDStreamController, _savedSchedulesStreamController, _appThemeStreamController), звільняючи пов'язані з ними ресурси. Це важливо для запобігання витокам пам'яті.

Таким чином, SettingsStorage забезпечує централізоване, персистентне та реактивне управління основними налаштуваннями додатку, дозволяючи іншим частинам програми легко підписуватися на зміни налаштувань та оновлювати їх.

3.2 Domain Layer

Відокремлення Domain Layer від Data Layer важливе, тому що воно робить додаток модульним, гнучким та легким для тестування і підтримки. Бізнес-логіка (правила та розрахунки) не залежить від того, як і звідки отримуються дані, а Data Layer (доступ до баз даних, API) не знає про специфічні бізнес-правила. Це дозволяє змінювати або тестувати кожен шар незалежно, спрощує розробку та зменшує кількість помилок.

3.2.1 Репозиторій «університету»

Клас UniversityRepository відіграє роль централізованого сховища та точки доступу до всіх даних, пов'язаних з університетом, таких як розклади, групи, викладачі, аудиторії та предмети. Він інкапсулює логіку отримання цих даних як із зовнішніх API, так і з локальної бази даних (DriftDB), а також управління налаштуваннями користувача, пов'язаними з університетськими даними.

Репозиторій також активно використовує BehaviorSubject з бібліотеки RxDart для створення реактивних потоків даних. Для кожного типу сутності (події, групи, викладачі, аудиторії, предмети) існує свій BehaviorSubject (наприклад, _eventsStreamController, _groupsStreamController), який зберігає

останній актуальний список цих сутностей і сповіщає підписників про будь-які зміни. Публічні гетери, такі як events, groups, надають доступ до цих потоків у вигляді широкомовних потоків (asBroadcastStream). Існує також _updatingScheduleStreamController для відстеження стану оновлення розкладу (чи йде процес оновлення та для якого розкладу).

При створенні репозиторію викликається метод _init. У цьому методі відбувається підписка на потоки даних з локальної бази даних _driftDB (наприклад, _driftDB.loadEvents()). Коли з бази даних надходять нові дані, вони перетворюються з моделей бази даних (наприклад, db.EventData) на моделі домену (наприклад, Event за допомогою Event.fromDBModel) і передаються у відповідні BehaviorSubject контролери, оновлюючи таким чином потоки даних, доступні для решти додатку (лістинг 3.9).

Лістинг 3.9 – Процес перетворення DTO події в бізнес-модель

```
(eventsData) {
      final events = <Event>[
        ...eventsData.map(
          (e) => Event.fromDBModel(
            e.event,
            (e.type == null) ? null :
EventType.fromDBModel(e.type!),
          ),
        ),
      ];
      _eventsStreamController.add(events);
    });
```

Метод fetchEntities() слугує для оркестрації завантаження основних сутностей: груп, викладачів та аудиторій. Він встановлює прапорець оновлення через _updatingScheduleStreamController.

Методи fetchGroups(), fetchTeachers() та fetchRooms() відповідають за отримання даних з відповідних API (_groupsApi, _teachersApi, _roomsApi). Отримані дані обробляються (наприклад, для груп використовується Set для уникнення дублікатів), перетворюються на моделі, сумісні з DriftDB (toDBModel()), і зберігаються в локальну базу даних за допомогою

_driftDB.saveGroups(), _driftDB.saveTeachers(), _driftDB.saveRooms().

updateSchedule(ScheduleData schedule): основний метод для оновлення розкладу. Він встановлює стан оновлення, визначає часовий проміжок для запиту (поточний семестр), викликає відповідний метод _eventsApi (залежно від типу розкладу: для групи, викладача чи аудиторії), видаляє старі події, пов'язані з цим розкладом (_deleteEvents), зберігає нові предмети та події в локальну базу даних. Обробляє можливу помилку EventsRequestFailure.

addSchedule(ScheduleData schedule): додає інформацію про новий розклад у SettingsStorage (список збережених розкладів користувача) і потім викликає updateSchedule() для завантаження даних цього розкладу.

removeSchedule(ScheduleData schedule): видаляє інформацію про розклад з SettingsStorage і потім викликає _deleteEvents() для видалення відповідних подій з локальної бази даних.

_deleteEvents(ScheduleData schedule, [bool safe = false]): внутрішній метод для видалення подій. Він визначає, які події потрібно видалити на основі ScheduleData. Параметр safe використовується для того, щоб не видаляти події, які все ще пов'язані з іншими збереженими розкладами (перевіряється за допомогою _eventSafeToDelete).

_eventSafeToDelete(Event event, SavedSchedules savedSchedules): допоміжний метод, який перевіряє, чи можна безпечно видалити подію, тобто чи не пов'язана вона з якимось іншим активним збереженим розкладом.

Репозиторій надає прямий доступ до потоків userGroupID та savedSchedules з _settingsStorage. Метод setUserGroupID(int groupID) делегує встановлення ID групи користувача до _settingsStorage. Оскільки дане налаштування тісно пов’язане з даними про групи що належать до предметної області університету, логічно було помістити його в репозиторій університету.

Метод dispose відповідає за коректне звільнення ресурсів. Він скасовує всі активні підписки на потоки даних з _driftDB та закриває всі BehaviorSubject контролери, щоб уникнути витоків пам'яті.

3.2.2 Репозиторій задач

Клас TasksRepository слугує централізованим сховищем та менеджером для всіх операцій, пов'язаних із завданнями (як основними завданнями, так і підзавданнями) у додатку. Він абстрагує логіку взаємодії з локальною базою даних (DriftDB) та надає реактивний доступ до списку завдань.

Цей репозиторій також використовує BehaviorSubject<List<Supertask>> (_tasksStreamController) з бібліотеки RxDart. Цей контролер зберігає останній актуальний список "суперзавдань" і сповіщає всіх підписників про будь-які зміни в цьому списку. Початковим значенням є порожній список.

Публічний геттер Stream<List<Supertask>> get tasks надає зовнішнім компонентам доступ до цього потоку завдань у вигляді широкомовного потоку (asBroadcastStream), що дозволяє мати декілька підписників.

При створенні екземпляра TasksRepository викликається метод _init. У цьому методі відбувається підписка (_tasksSubscription) на потік даних, що надходить з бази даних через _driftDB.loadTasks(). Цей метод з _driftDB повертає потік усіх завдань з таблиці завдань.

Коли з бази даних надходить новий список завдань (tasks), він обробляється. Спочатку всі завдання розділяються на дві групи: supertasks (ті, у яких supertaskID == null, тобто вони не є підзавданнями) та subtasks (ті, у яких supertaskID != null). Потім для кожного "суперзавдання" з supertasks створюється екземпляр моделі домену Supertask за допомогою Supertask.fromDBModel(). При цьому йому передаються відповідні "підзавдання" з subtasks, які мають такий самий supertaskID, як id поточного суперзавдання. Сформований список об'єктів Supertask (з уже вкладеними підзавданнями) передається в _tasksStreamController, оновлюючи таким чином публічний потік tasks (лістинг 3.10).

Лістинг 3.10 – Перетворення DTO задач в бізнес-модель

```
(tasks) {
      final result = <Supertask>[];

      final supertasks = tasks.where((task) => task.supertaskID
== null);
      final subtasks = tasks.where((task) => task.supertaskID !=
null);

      result.addAll(
        supertasks.map(
          (e) => Supertask.fromDBModel(
            e,
            subtasks.where((task) => task.supertaskID == e.id),
          ),
        ),
      );

      _tasksStreamController.add(result);
    }
```

saveTask(Task task, int supertaskID): зберігає звичайне завдання в базу даних. Воно перетворюється на модель бази даних (task.toDBModel(supertaskID)) і передається в _driftDB.saveTask(). Повертає ID збереженого завдання.

saveSupertask(Supertask task): зберігає "суперзавдання" в базу даних. Аналогічно, воно перетворюється на модель БД і зберігається.

saveSupertaskWithSubtasks(Supertask task): зберігає "суперзавдання" разом з усіма його підзавданнями. Спочатку викликається _driftDB.saveTasks() для збереження всіх підзавдань (отриманих через task.subtasksToDBModels(task.id)), а потім зберігається саме суперзавдання.

saveSupertasks(Iterable<Supertask> tasks): дозволяє зберегти колекцію "суперзавдань". Для кожного суперзавдання спочатку зберігається саме воно, щоб отримати його ID, а потім збираються всі його підзавдання. В кінці всі зібрані підзавдання зберігаються одним пакетом.

deleteTask(int id): видаляє одне завдання (це може бути як підзавдання, так і суперзавдання без підзавдань) з бази даних за його ID.

deleteSupertask(int id): видаляє "суперзавдання" з бази даних за його ID. _driftDB.deleteSupertask(id) також забезпечує каскадне видалення пов'язаних

підзавдань.

deleteGeneratedTasks(): видаляє всі завдання, які були позначені як "згенеровані" (в таблиці завдань бази даних є відповідний прапорець).

Метод dispose відповідає за коректне звільнення ресурсів при знищенні репозиторію. Він скасовує підписку _tasksSubscription на потік даних з бази даних та закриває _tasksStreamController, щоб уникнути витоків пам'яті та некоректної роботи.

3.3 Application Layer

Application Layer відповідає виключно за відображення даних користувачеві та обробку його взаємодій, не втручаючись у бізнес-логіку чи механізми доступу до даних.

Такий підхід дозволяє легко змінювати користувацький інтерфейс, навіть повністю його перероблюючи, не зачіпаючи при цьому основну логіку програми. Це також дає можливість незалежно тестувати бізнес-логіку та процеси роботи з даними без необхідності запуску та перевірки самого UI. Сам Application Layer також можна тестувати ізольовано, наприклад, за допомогою спеціалізованих тестів для віджетів. Крім того, це покращує підтримуваність коду, роблячи його більш організованим і зрозумілим для розробників, які чітко бачать, де знаходиться логіка, відповідальна за вигляд, а де – за обробку даних. Нарешті, це підвищує можливість перевикористання компонентів UI, оскільки вони стають більш універсальними, не містячи специфічної бізнес-логіки.

У цьому шарі проєкт було структуровано за функціоналом. Тобто, файли проєкту були об’єднані в директорії за тим, яку роль вони виконують, що значно спрощує орієнтування у файлах проєкту.

Кожен екран додатку супроводжується відповідним ViewModel. Так як використовується бібліотека Bloc – це або Bloc, або простіший Cubit.

3.3.1 Головний екран

Головний екран застосунку складається з двох основних віджетів: AppPage та AppView. AppPage відповідає за ініціалізацію та надання ключових сервісів та сховищ даних, які будуть використовуватися в усьому додатку. AppView, у свою чергу, формує візуальну оболонку додатку, зокрема, налаштовує його тему та локалізацію.

На початку файлу імпортуються необхідні бібліотеки та компоненти. Це включає локальні пакети, що містять логіку для роботи з базою даних, інтерфейси для отримання даних про події, групи, аудиторії та викладачів з зовнішніх джерел, репозиторії для управління даними налаштувань, завдань та університету, сховище для налаштувань, а також компоненти для локалізації та управління глобальним станом додатку.

Віджет AppPage виступає кореневим елементом, який готує середовище для роботи всього додатку. Він приймає через конструктор функції для створення репозиторіїв (createUniversityRepository, createTasksRepository, createSettingsRepository), а також екземпляри клієнта бази даних (driftDB), сховища налаштувань (settingsStorage) та клієнтів для взаємодії з API (eventsApi, groupsApi, teachersApi, roomsApi). Така структура дозволяє гнучко керувати створенням та налаштуванням залежностей.

У методі build віджета AppPage використовується MultiRepositoryProvider. Цей інструмент дозволяє зробити екземпляри репозиторіїв доступними для всіх дочірніх віджетів у дереві. Створюються UniversityRepository, TasksRepository та SettingsRepository, яким передаються відповідні залежності, отримані через конструктор AppPage. Наприклад, UniversityRepository отримує клієнтів API, базу даних та сховище налаштувань. Важливо, що для кожного репозиторію вказано метод dispose, який буде викликаний при видаленні провайдера, що допомагає коректно звільняти ресурси.

Після надання репозиторіїв, AppPage використовує BlocProvider для

створення та надання екземпляра AppCubit. AppCubit відповідає за управління глобальним станом додатку, наприклад, поточною темою оформлення. При його створенні йому передається SettingsRepository, отриманий з контексту. Нарешті, AppPage відображає AppView.

Віджет AppView відповідає за візуальне представлення кореневого рівня додатку. Він використовує BlocBuilder для того, щоб реагувати на зміни стану в AppCubit. Залежно від поточного стану, AppView конфігурує та відображає основний віджет MaterialApp. У MaterialApp налаштовуються підтримувані локалі та делегати локалізації для інтернаціоналізації додатку. Режим теми (світла, темна або системна) встановлюється на основі значення state.theme.appThemeMode з AppCubit. Також визначаються світла та темна теми за замовчуванням. Як головний екран додатку (home) вказується HomePage.

AppCubit (лістинг 3.11) – компонент, що відповідає за управління глобальним станом додатку, зокрема, за тему оформлення. Цей AppCubit є реалізацією BLoC у його спрощеному варіанті Cubit і слугує своєрідним ViewModel для глобальних налаштувань програми.

Лістинг 3.11 – Код AppCubit

```
class AppCubit extends Cubit<AppState> {
  AppCubit({required SettingsRepository settingsRepository})
    : _settingsRepository = settingsRepository,
      super(AppState(theme: AppTheme.defaultValues())) {
    init();
  }

  final SettingsRepository _settingsRepository;

  late final StreamSubscription<AppTheme> _subscription;

  void init() {
    _subscription = _settingsRepository.appTheme.listen(
      (event) => emit(state.copyWith(event)),
    );
  }

  @override
  Future<void> close() {
```

```
    _subscription.cancel();
    return super.close();
  }
}
```

При створенні екземпляра AppCubit йому обов'язково передається SettingsRepository. Цей репозиторій відповідає за доступ до збережених налаштувань додатку. Початковий стан AppCubit встановлюється як AppState з темою за замовчуванням (AppTheme.defaultValues()). Одразу після створення, у конструкторі викликається метод init().

Метод init() виконує ключову роль: він підписується на потік змін appTheme з _settingsRepository. Це означає, що кожного разу, коли тема додатку змінюється у SettingsRepository (наприклад, користувач обрав іншу тему в налаштуваннях), AppCubit отримає сповіщення про цю зміну. У відповідь на отриману нову тему (event), AppCubit випускає (emit) новий стан AppState, копіюючи попередній стан та оновлюючи в ньому лише значення теми (state.copyWith(event)). Таким чином, будь-які компоненти інтерфейсу, що слухають цей AppCubit, будуть автоматично перебудовані з новою темою.

3.3.2 Екран огляду задач

TasksOverviewBloc – ViewModel для екрану огляду задач. Він відповідає за логіку відображення списку суперзадач, їх створення, зміну стану виконання, а також за генерацію задач на основі університетського розкладу.

TasksOverviewBloc взаємодіє з двома репозиторіями: TasksRepository для операцій безпосередньо з задачами (збереження, видалення, отримання списку) та UniversityRepository для отримання даних про розклад (події, предмети, ідентифікатор групи користувача), які необхідні для генерації задач.

При ініціалізації BLoC підписується на потік задач з TasksRepository.

Коли надходить список задач, він випускає стан TasksOverviewSuccess з цими задачами, що дозволяє UI оновитися.

Ключовою функцією є обробка запиту на генерацію задач (_onGenerationRequested). Спочатку видаляються всі раніше згенеровані задачі. Потім BLoC отримує розклад для групи поточного користувача, виділяє унікальні предмети та для кожного предмета створює суперзадачу. Підзадачі для цих суперзадач генеруються на основі типів занять (практики, лабораторні, екзамени) за допомогою допоміжного методу _createTasksForType. Цей метод групує події певного типу та створює відповідні підзадачі зі строками виконання, взятими з розкладу на основі алгоритму (рис. 3.2). Даний алгоритм створює задачу для однієї, двох або всіх занять певного типу для кожного предмету. Після формування списку згенерованих суперзадач вони зберігаються через TasksRepository.

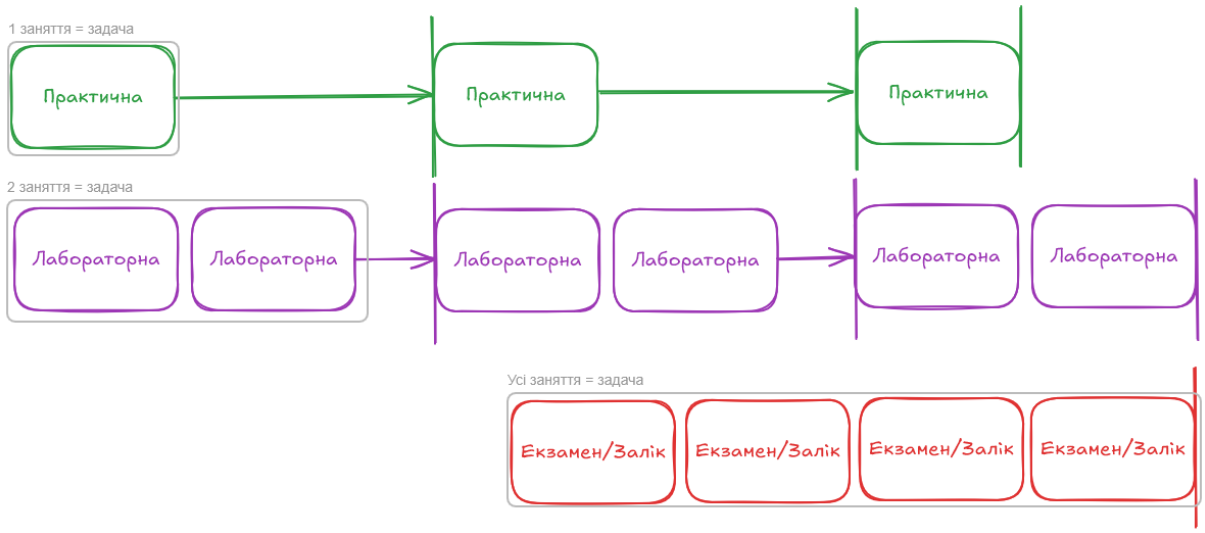


Рисунок 3.2 – Принцип генерації задач

Також BLoC обробляє зміну стану виконання суперзадачі (коли користувач відмічає її як виконану/невиконану) та запит на створення нової суперзадачі, зберігаючи відповідні зміни через TasksRepository та випускаючи стан TasksOverviewSupertaskCreated для навігації на екран новоствореної задачі.

Додатково, є розширення EventBaseTypeToTaskType, яке перетворює тип події з розкладу на відповідний тип задачі для коректної класифікації згенерованих задач.

3.3.3 Екран задачі

Даний екран відповідає за відображення задачі та її підзадач та складається з двох основних віджетів: SupertaskViewPage та SupertaskViewView.

SupertaskViewPage є віджетом-обгорткою, який відповідає за початкове налаштування Bloc для цього екрану та обробку деяких глобальних змін стану, що впливають на навігацію або відображення модальних вікон. У ньому є статичний метод route, який використовується для переходу на цей екран. При переході цей метод створює BlocProvider для SupertaskViewBloc. SupertaskViewBloc ініціалізується з TasksRepository (отриманим з контексту, що означає, що він був наданий десь вище у дереві віджетів) і одразу ж отримує початкову подію SupertaskViewSubscriptionRequested, яка завантажує дані для суперзадачі з вказаним initialTaskID, якщо такий є.

Усередині SupertaskViewPage використовується BlocListener. Цей компонент слухає зміни стану SupertaskViewBloc і виконує певні дії залежно від стану, не перебудовуючи при цьому весь інтерфейс. Наприклад, якщо стан змінюється на SupertaskViewSupertaskDeleted, то екран автоматично закривається (Navigator.pop(context)). Якщо ж стан стає SupertaskViewSubtaskCreated, це означає, що було створено нову підзадачу. У такому випадку, після невеликої затримки, відкривається модальне вікно (showModalBottomSheet), в якому відображається сторінка для редагування цієї нової підзадачі (TaskViewPage).

Основний інтерфейс екрану будується у віджеті SupertaskViewView. Цей віджет використовує BlocBuilder для того, щоб динамічно перебудовувати свій вміст залежно від поточного стану SupertaskViewBloc.

Інтерфейс SupertaskViewView представлений Scaffold. У ньому є плаваюча кнопка дії (FloatingActionButton), яка з'являється тільки тоді, коли дані суперзадачі успішно завантажені (стан SupertaskViewSuccess). Натискання на цю кнопку відправляє подію SupertaskViewSubtaskCreationRequested до BLoC, ініціюючи процес створення нової підзадачі.

Основний вміст екрану реалізовано за допомогою CustomScrollView, що дозволяє створювати гнучкі прокручувані інтерфейси. Верхня панель (SliverAppBar.large) відображає заголовок екрану "Редагування суперзадач" (локалізований рядок) та кнопку видалення суперзадачі. Кнопка видалення також з'являється тільки при успішному завантаженні даних і при натисканні відправляє подію SupertaskViewSupertaskDeletionRequested з ідентифікатором поточної суперзадачі.

Нижче верхньої панелі, якщо дані успішно завантажені, відображаються поля для редагування властивостей самої суперзадачі. Це включає поле для назви (SupertaskTitleField), поле для встановлення дедлайну (SupertaskDeadlineField) та віджет для вибору типу суперзадачі (SupertaskTypeSelectionWidget). Кожна зміна у цих полях відправляє подію SupertaskViewDataChanged до BLoC з оновленими даними. Також відображається текстовий напис "Підзадачі".

Далі, залежно від поточного стану SupertaskViewBloc (рис. 3.3), відображається або індикатор завантаження (для станів SupertaskViewInitial, SupertaskViewLoading, SupertaskViewSupertaskDeleted), або повідомлення про помилку (ErrorSubtasksWidget для стану SupertaskViewFailure), або список підзадач. Якщо підзадач немає (стан SupertaskViewSuccess і список підзадач порожній), то відображається відповідне повідомлення (NoSubtasksWidget). Якщо ж підзадачі є, вони відображаються списком (SliverList.builder), де кожен елемент списку є SubtaskListItem. Кожен елемент підзадачі дозволяє перейти до її редагування (відкривається TaskViewPage у модальному вікні), позначити її як виконану/невиконану (відправляє подію

SupertaskViewSubtaskCheckboxToggled) або видалити її (відправляє подію SupertaskViewSubtaskDeletionRequested).

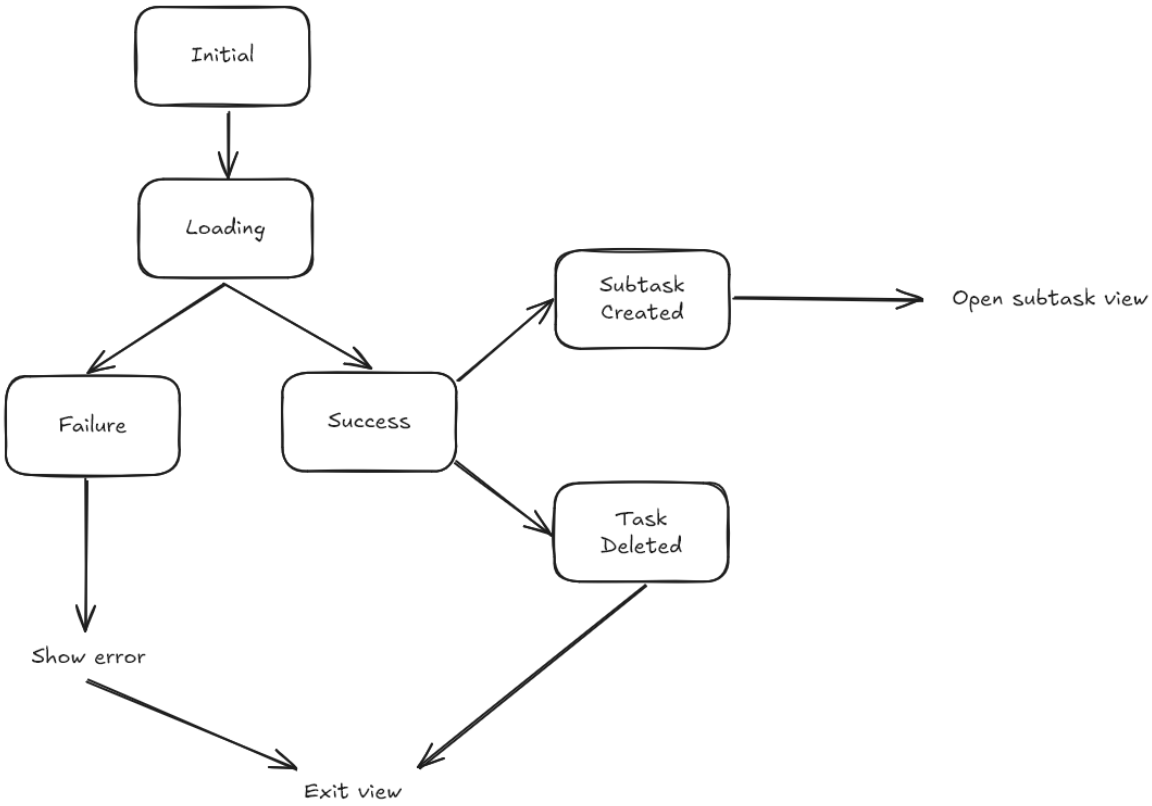


Рисунок 3.3 – Граф станів екрану задачі

SupertaskViewBloc є ViewModel для екрану перегляду та редагування задачі. Цей Bloc відповідає за управління станом цього екрану, обробку користувацьких дій та взаємодію з TasksRepository для отримання та збереження даних про задачі.

Клас SupertaskViewBloc успадковується від Bloc<SupertaskViewEvent, SupertaskViewState>, що вказує на те, що він буде обробляти події типу SupertaskViewEvent та випускати стани типу SupertaskViewState. Файли supertask_view_event.dart та supertask_view_state.dart (на що вказують директиви part) містять визначення цих подій та станів.

У конструкторі SupertaskViewBloc приймає обов'язковий параметр tasksRepository. Цей репозиторій надає доступ до логіки роботи з задачами. Початковий стан Bloc встановлюється як SupertaskViewInitial. Далі

реєструються обробники для різних типів подій.

При отриманні події SupertaskViewSubscriptionRequested (лістинг 3.12), Bloc спочатку випускає стан SupertaskViewLoading, сигналізуючи про початок завантаження даних. Потім він підписується на потік tasks з _tasksRepository. Кожного разу, коли дані в цьому потоці оновлюються (тобто змінюється список суперзадач), Bloc намагається знайти суперзадачу з ідентифікатором, вказаним у події (event.taskID). Якщо така задача не знайдена (можливо, її було видалено), випускається стан SupertaskViewSupertaskDeleted. Якщо ж задача знайдена, випускається стан SupertaskViewSuccess, який містить дані знайденої суперзадачі та перевіряє її назву на коректність за допомогою внутрішнього методу _checkTitle.

Лістинг 3.12 – Функція обробки подій потоку

```
emit.forEach(
      _tasksRepository.tasks,
      onData: (supertasks) {
        final supertask = supertasks.firstWhereOrNull(
          (task) => task.id == event.taskID,
        );

        if (supertask == null) {
          return const SupertaskViewSupertaskDeleted();
        } else {
          return SupertaskViewSuccess(
            task: supertask,
            titleError: _checkTitle(supertask.title),
          );
        }
      },
    );
```

Коли відбувається зміна даних суперзадачі на екрані (подія SupertaskViewDataChanged), Bloc спочатку перевіряє нову назву. Якщо назва некоректна (наприклад, порожня), випускається стан SupertaskViewSuccess з тією ж суперзадачею, але з вказанням помилки назви. Якщо ж назва коректна, Bloc викликає метод saveSupertask у _tasksRepository, передаючи йому оновлену суперзадачу. Суперзадача оновлюється шляхом копіювання

поточного стану суперзадачі та заміни її полів (назва, тип, дедлайн) на нові значення з події.

Коли надходить запит на створення нової підзадачі (подія SupertaskViewSubtaskCreationRequested), Bloc створює нову підзадачу з назвою за замовчуванням та іншими початковими значеннями. Потім він викликає метод saveTask у _tasksRepository для збереження цієї нової підзадачі, пов'язуючи її з поточною суперзадачею. Після успішного збереження Bloc випускає стан SupertaskViewSubtaskCreated, який містить дані поточної суперзадачі та ідентифікатор новоствореної підзадачі. Цей стан потім використовується на UI для відкриття екрану редагування нової підзадачі.

Внутрішній метод _checkTitle перевіряє, чи не є надана назва порожньою або такою, що складається лише з пробілів. Якщо так, він повертає помилку TitleError.emptyOrWhitespace, інакше – null.

3.3.4 Керування збереженими розкладами

Основний інтерфейс екрану будується у віджеті SchedulesViewView. Цей віджет використовує BlocBuilder для того, щоб динамічно перебудовувати свій вміст залежно від поточного стану SchedulesViewCubit.

Перед побудовою основного інтерфейсу, SchedulesViewView отримує зі стану (state.schedules) списки збережених розкладів та фільтрує їх за типом: окремо для груп (groups), викладачів (teachers) та аудиторій (rooms). Це робиться для того, щоб потім відобразити кожен тип розкладу у своєму розділі.

Інтерфейс SchedulesViewView представлений Scaffold, а його основний вміст реалізовано за допомогою CustomScrollView. Це дозволяє мати гнучкий прокручуваний інтерфейс з різними типами елементів. Верхня панель (SliverAppBar.large) відображає заголовок екрану "Керування розкладами" (локалізований рядок).

Далі йдуть секції для кожного типу розкладів (групи, викладачі, аудиторії). Кожна секція починається з віджета SchedulesTypesDivider. Цей віджет відображає назву типу розкладу (наприклад, "Групи") та кнопку "додати". Натискання на кнопку "додати" ініціює перехід на екран EntitySelectionPage, де користувач може обрати відповідну сутність (групу, викладача чи аудиторію) для додавання її розкладу. Тип сутності передається на екран вибору через параметр tab.

Після роздільника для кожного типу розкладу відображається список збережених розкладів цього типу за допомогою SliverList.builder. Кожен елемент списку є віджетом SchedulesListItem. Цей віджет відображає назву розкладу. Також SchedulesListItem містить логіку для відображення стану оновлення (чи йде зараз оновлення цього конкретного розкладу, чи заборонено оновлення взагалі). Важливою є перевірка deleteProhibited для розкладів груп: якщо ідентифікатор групи збігається з ідентифікатором групи поточного користувача (state.userGroupID), то видалення такого розкладу заборонено. Кожен елемент списку надає кнопки для оновлення (onRefresh) та видалення (onDelete) розкладу. Натискання на ці кнопки відправляє відповідні події (updateSchedule або removeSchedule) до SchedulesViewCubit з даними конкретного розкладу.

Важливим є поле стану updateStatus, що зберігає стан оновлення. Якщо відбувається оновлення розкладу чи списку груп, викладачів або аудиторій, то bool буде true, що заблокує інші дії. Це викликано особливостями API, яке не варто навантажувати.

3.3.5 Локалізація застосунку

Локалізація застосунків, тобто їх адаптація до різних мов, є важливим кроком для охоплення ширшої аудиторії. У Flutter для локалізації треба створити файли з локалізованими рядками за допомогою яких буде згенеровано код класів що відповідають за локалізацію.

У кореневому віджеті MaterialApp (лістинг 3.13) було вказано делегати локалізації та підтримувані локалі, що додає AppLocalizations певної мови до дерева віджетів застосунку для подальшого використання.

Лістинг 3.13 – Віджет Material App

```
MaterialApp(
  supportedLocales: AppLocalizations.supportedLocales,
  localizationsDelegates:
AppLocalizations.localizationsDelegates,
  theme: ThemeData(
    colorScheme: ColorScheme.fromSeed(
      seedColor: state.theme.color.toColorSeed(),
      brightness: state.theme.themeMode.toBrightness(),
    ),
  ),
  home: const HomePage(),
);
```

Для отримання локалізованих рядків необхідно отримати згенерований клас AppLocalizations з дерева віджетів (лістинг 3.14). На самому початку методу build створюється змінна l10n що отримує референс на існуючий об’єкт AppLocalizations.

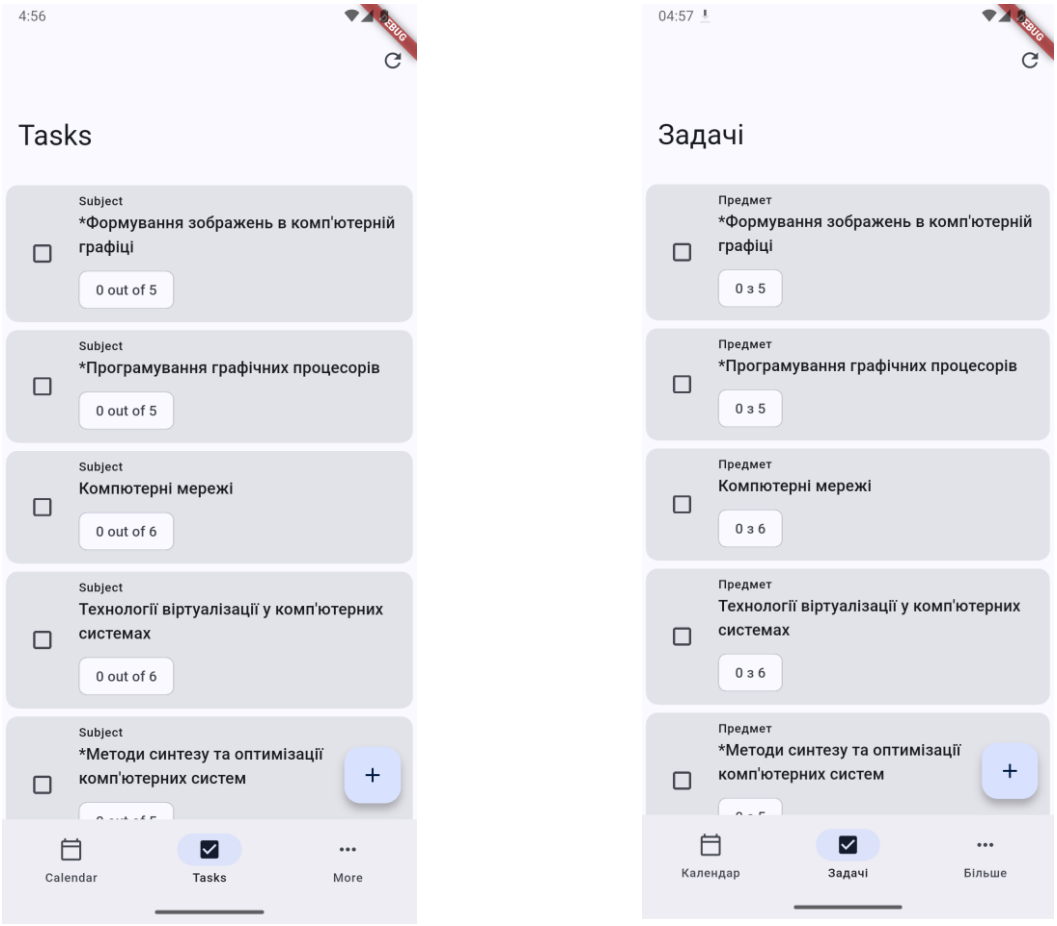
Даний клас надає гетери для отримання локалізованих рядків. Все, що необхідно розробнику для отримання локалізованих рядків – це викликати необхідні гетери.

Лістинг 3.14 – Код отримання та використання локалізованих рядків

```
final l10n = AppLocalizations.of(context)!;

TextField(
  onChanged:
    (value) => context
      .read<EntitySelectionCubit>()
      .setSearchValue(value),
  decoration: InputDecoration(
    hintText: l10n.search,
    prefixIcon: Icon(Icons.search),
  ),
),
```

Додання нової мови до застосунку є дуже простим. Спочатку потрібно створити новий файл з перекладами у форматі ARB, назвавши його відповідно до коду нової мови, наприклад, app_uk.arb для української. У цьому файлі необхідно вказати код мови у полі "@@locale" та перекласти всі наявні текстові рядки на цю нову мову. Нарешті, потрібно запустити команду для перегенерації коду локалізацій, щоб згенерований клас для роботи з перекладами, такий як AppLocalizations, оновився та включив підтримку нової мови. Додана мова буде використовуватись за необхідністю (рис. 3.4).



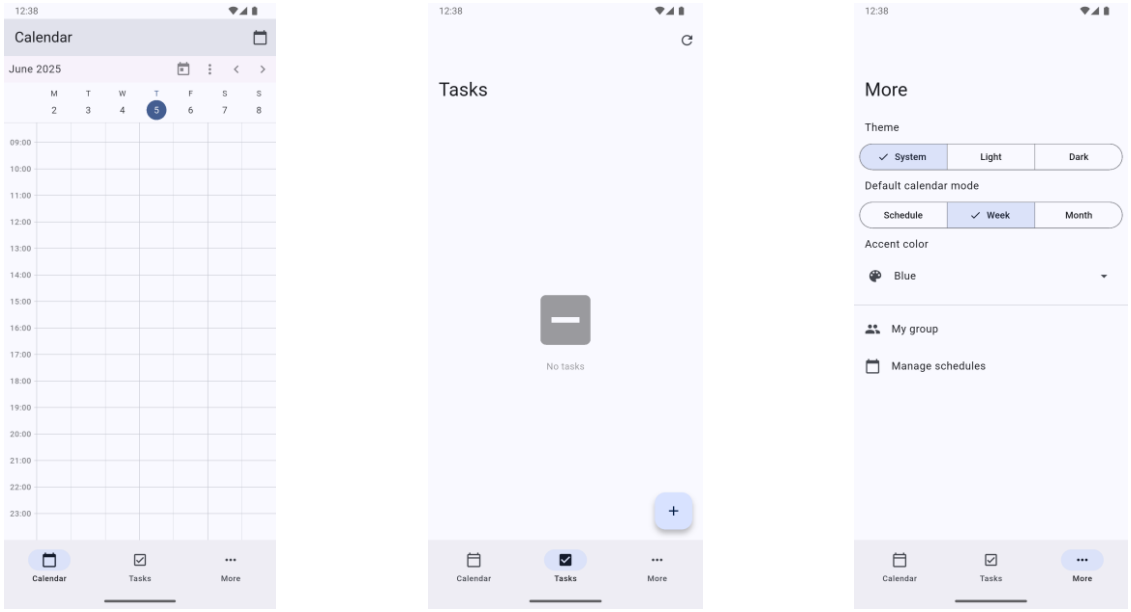
а) б)

Рисунок 3.4 – Локалізації застосунку: а) англійська; б) українська

4 АНАЛІЗ РЕЗУЛЬТАТІВ

4.1 Огляд застосунку

При першому запуску користувач бачить головний екран застосунку (рис 4.1). Він складається з навігаційної панелі та обраного режиму. Режим календаря показує календар з подіями, режим задач – задачі, а третій режим є своєрідним меню налаштувань.



а) б) в)

Рисунок 4.1 – Головний екран додатку: а) екран календаря; б) екран задач; в) екран налаштувань

При першому запуску застосунку він не буде містити жодних даних. Якщо користувач хоче побачити розклад то йому необхідно його додати та обрати. Для додавання розкладів використовується меню керування розкладами (рис 4.2). Тут користувач може додати, оновити, видалити чи обрати для відображення розклад.

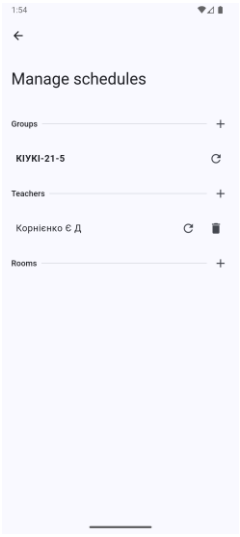
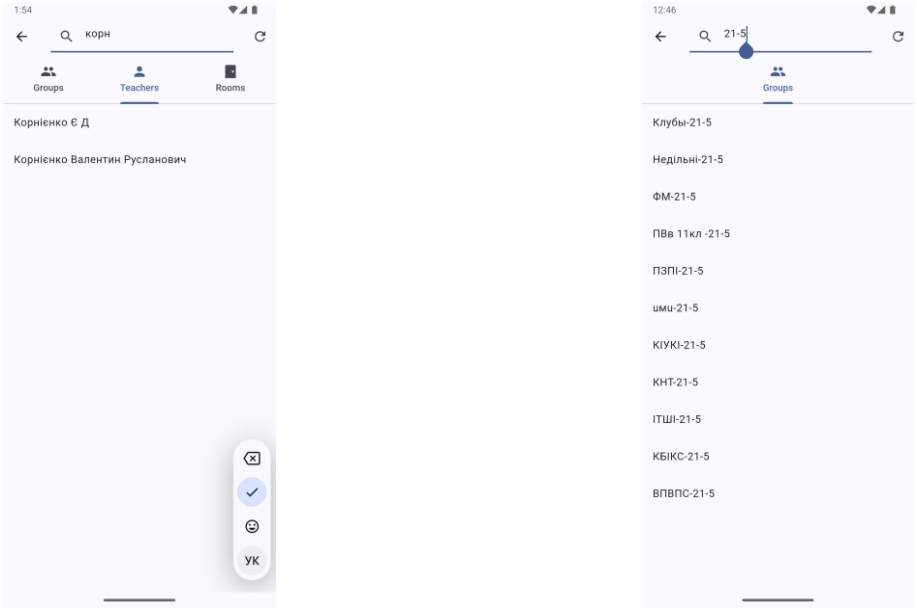


Рисунок 4.2 – Екран керування розкладами

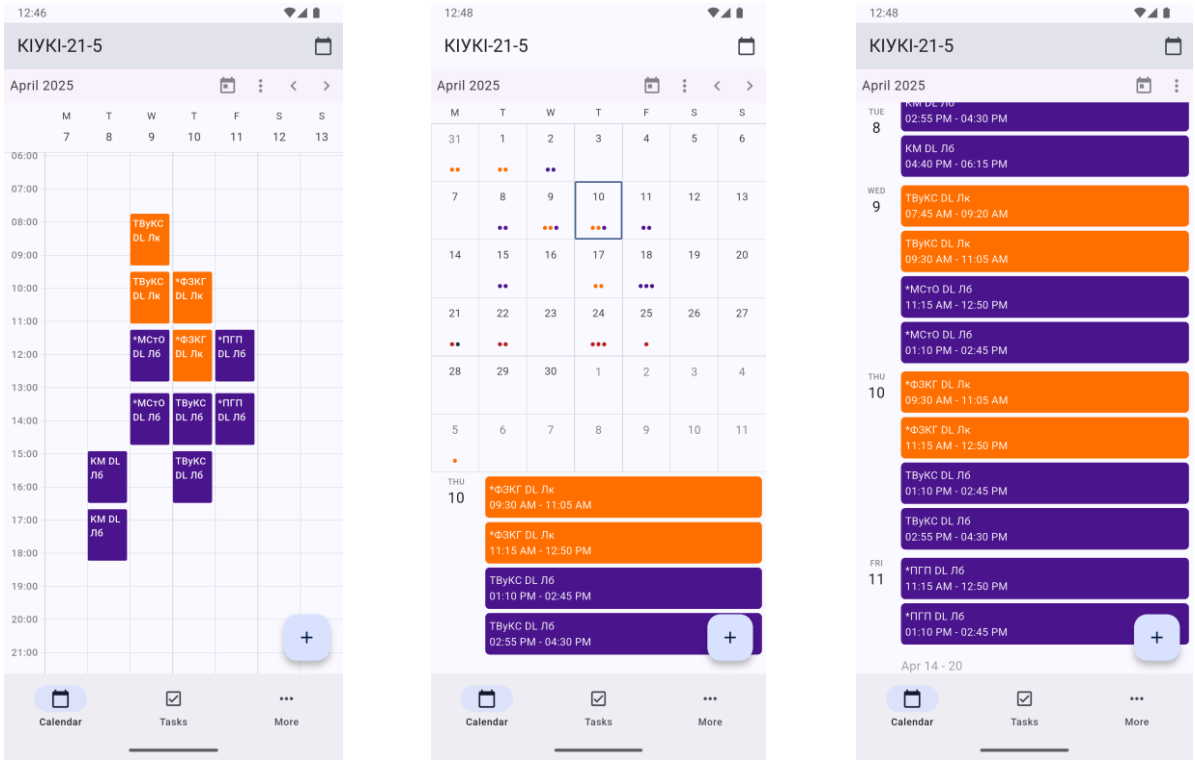
Натискання на кнопку додавання розкладу відкриє відповідний екран (рис 4.3). Тут користувач може знайти групу, викладача чи кімнату та додати відповідний розклад. Спроба його оновлення станеться автоматично. Також в налаштуваннях користувач може обрати пункт вибору своєї групи. Це відкриє цей ж екран, але з можливістю вибору тільки групи, що й буде збережена як група користувача.



а) б)

Рисунок 4.3 – Екран додання розкладу: а) вибір розкладу; б) вибір групи користувача

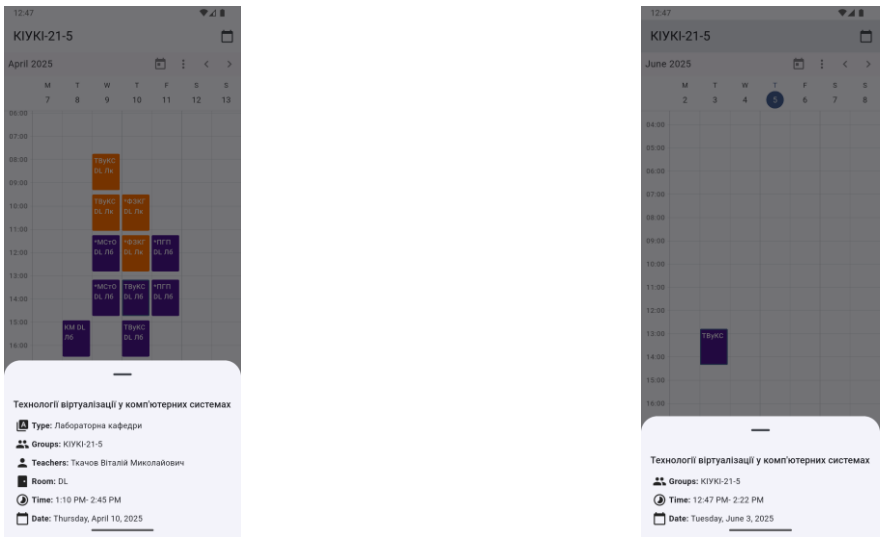
Після того як хоч один розклад було додано, користувач може обрати його для відображення і побачити його на екрані календаря (рис 4.4). Тут можна побачити різноманітні варіанти відображення розкладу. А саме три: список подій, тиждень та місяць. За замовчуванням відображається тиждень, але користувач може це змінити у налаштуваннях.



а) б) в)

Рисунок 4.4 – Екран календаря з розкладом: а) тижневий календар; б) календар на місяць; в) список подій

Натискання на подію відкриє повну інформацію про неї: повну назву предмету, тип, групи та викладачів, кімнати, точний час та дати (рис 4.5).



а) б)

Рисунок 4.5 – Відображення інформації про подію: а) інформація з розкладу; б) додана користувачем подія

Натискання на кнопку додавання події чи затискання користувацької події викличе вікно створення та редагування події (рис 4.6) де користувач зможе додати свою подію чи змінити вже додану.

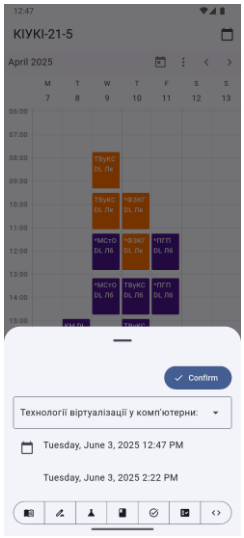


Рисунок 4.6 – Редагування події

Екран задач (рис 4.7) буде пустим доки користувач не додасть свою задачу чи не згенерує їх на основі розкладу. Генерація потребує вибору групи, додавати ж задачі можна одразу.

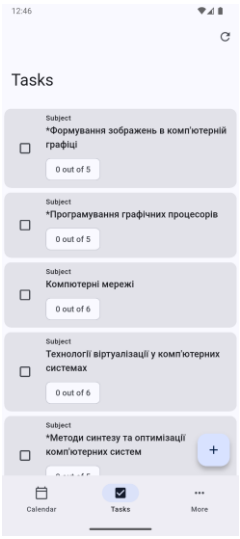


Рисунок 4.7 – Список задач після генерації на основі розкладу

Натискання на задачу перенесе користувача на її екран (рис. 4.8). Тут він зможе її редагувати, наприклад, додати строк виконання чи додати підзадачу.

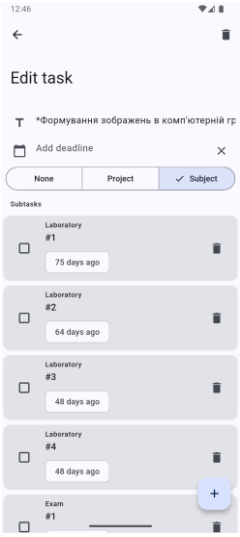


Рисунок 4.8 – Екран редагування задачі

Натискання на підзадачу чи створення нової відкриє екран редагування підзадачі (рис 4.9). Тут можна змінити, наприклад, назву, чи строк виконання.

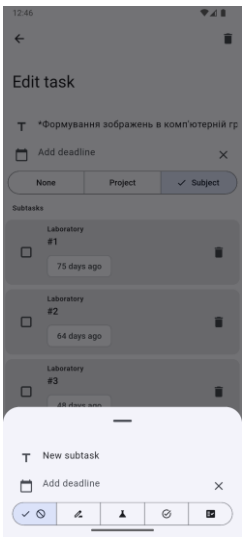
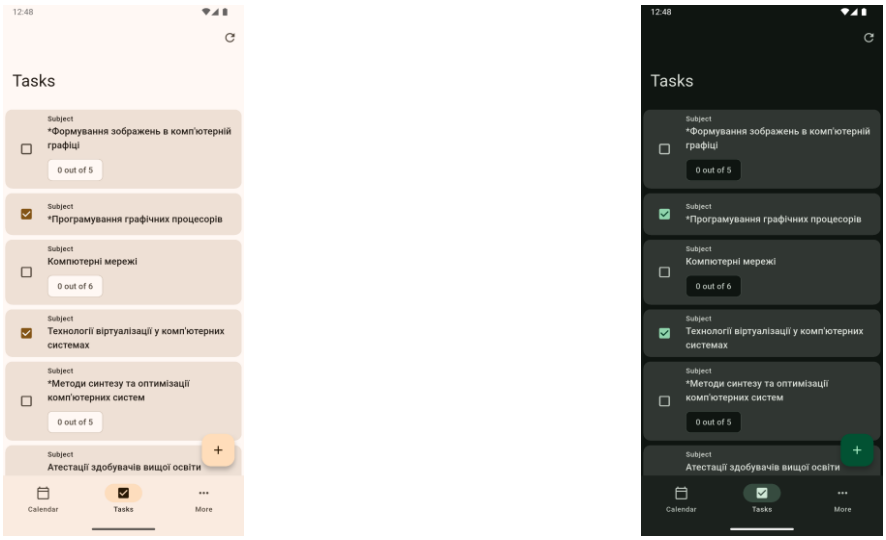


Рисунок 4.9 – Екран редагування підзадачі

На екрані налаштувань користувач може змінити тему застосунку. Зміни будуть прийняті одразу. Користувач може обрати між світлою, темною та темою пристрою. Також можна обрати колір акценту застосунку (рис 4.10).



а) б)

Рисунок 4.10 – Зовнішній вигляд застосунку за різних тем: а) світла тема з червоним акцентом; б) темна тема з зеленим акцентом

4.2 Тестування застосунку

Так як практично весь код застосунку має перевірку даних, що вводить користувач, і блокує дії що можуть призвести до проблем, тестування застосунку зводиться до тестування тієї його частини, що взамодіє з кодом, над яким у розробника немає контролю. У даному випадку, це API-клієнт CistApi. Для його тестування було розроблено два види тестів: тести сутностей та тести подій.

Тести сутностей (рис 4.13) мають переконатися що методи отримання сутностей повертають помилку навіть тоді, коли відповідь технічно коректна, але сутностей немає. Це свідчить про те, що, скоріше за все, сталась помилка на сервері. У такому випадку варто відкинути отриману відповідь.

Інший вид тестів це тести подій. API повертає некоректний JSON у разі якщо подій немає (рис 4.11). Відсутність подій не є помилкою, тому в такому разі має бути повернений пустий список подій. У випадку помилки, JSON міститиме помилку ORA-XXXXX (рис 4.12), яка вказує на те, що в результаті запиту виникла помилка.

Рисунок 4.11 – Відповідь серверу коли подій не знайдено

Рисунок 4.12 – Відповідь серверу у разі помилки

Знаючи це було розроблено відповідні тести (рис 4.13) щоб переконатися, що кожен з випадків обробляється вірно з логічної точки зору.

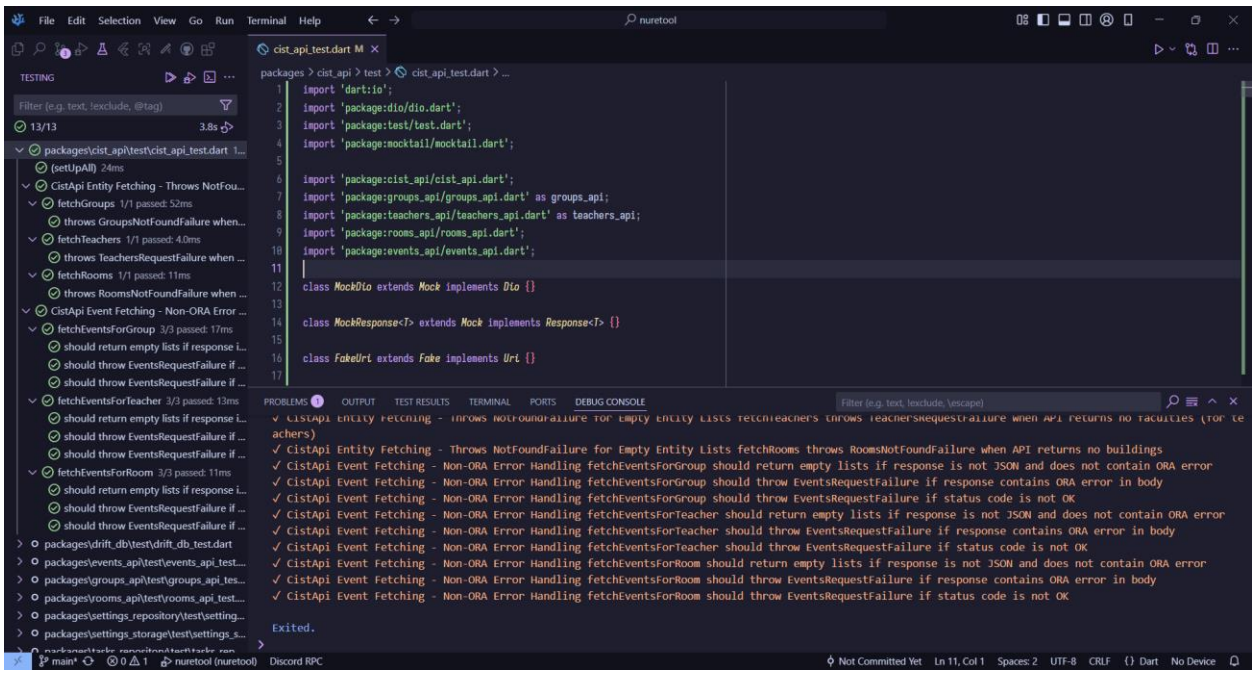


Рисунок 4.13 – Результат тестування API-клієнту

Ручне тестування було проведено декількома майбутніми користувачами і показало велику надійність застосунку. Серед знайдених проблем були виключно візуальні проблеми, що не впливали на функціонал застосунку, та які було легко виправити.

4.3 Можливості для покращення

Наразі застосунок ефективно працює з локальними даними, отримуючи розклад з API cist.nure.ua та зберігаючи користувацькі дані локально. Це забезпечує швидкий доступ та офлайн-роботу. Однак, відсутність хмарної синхронізації обмежує користувацький досвід.

Ключовим напрямком для покращення є впровадження хмарної синхронізації. Це дозволило б користувачам отримувати доступ до своїх задач, доданих вручну подій у календарі, налаштувань збережених розкладів та налаштувань самого застосунку з будь-якого свого пристрою. Також це забезпечило б резервне копіювання та відновлення даних у випадку проблем з пристроєм. Для реалізації цього потрібно буде інтегрувати хмарний сервіс, реалізувати автентифікацію користувачів та розробити логіку синхронізації, враховуючи можливі конфлікти даних при офлайн-змінах.

Також корисною була б інтеграція з зовнішніми календарями через двосторонню синхронізацію. Це дозволило б користувачам не просто експортувати свої події, а й бачити зміни, зроблені, наприклад, в Google Calendar, безпосередньо в застосунку, і навпаки, що забезпечило б централізоване керування всіма подіями.

Корисно було б додати розширені налаштування задач, такі як пріоритети та теги. Також можна надати користувачеві більш гнучке налаштування генерації задач з розкладу, дозволяючи обирати конкретні предмети чи типи занять. Розробка віджету для головного екрану пристрою дозволила б швидко переглядати найближчі події або задачі. Нарешті, аналітика та статистика щодо успішності виконання завдань або завантаженості могли б бути цікавими тих, хто хоче аналізувати свою продуктивність.

ВИСНОВКИ

У ході виконання даної кваліфікаційної роботи було успішно досягнуто поставлену мету – розроблено кросплатформенний застосунок для організації та керування навчальним процесом, орієнтований на потреби студентів, зокрема ХНУРЕ.

Було проведено детальний аналіз предметної області, що включав огляд існуючих програмних рішень для управління розкладом та задачами. Цей аналіз виявив ключові недоліки наявних інструментів, такі як обмежена функціональність, відсутність інтеграції з університетськими сервісами або незручний інтерфейс, що обґрунтувало актуальність розробки нового, більш комплексного продукту. Було також проаналізовано сучасні засоби кросплатформенної розробки, що призвело до вибору фреймворку Flutter та мови програмування Dart як оптимальних інструментів для реалізації поставленої задачі, завдяки їхній продуктивності, гнучкості та можливості створення єдиної кодової бази для різних платформ. Особливу увагу було приділено архітектурним підходам, зокрема принципам «чистої архітектури» та патерну MVVM, що забезпечило створення добре структурованого, підтримуваного та масштабованого застосунку.

Процес розробки охоплював створення всіх ключових шарів застосунку: шару даних, відповідального за взаємодію з API cist.nure.ua та локальною базою даних SQLite (за допомогою бібліотеки Drift) для зберігання розкладів, задач та користувацьких налаштувань; шару бізнес-логіки, що інкапсулює основні правила та операції над даними; та шару презентування, який реалізує користувацький інтерфейс та взаємодію з користувачем. Було реалізовано функціонал відображення актуального розкладу, отриманого з університетського сервісу, можливість додавання та керування власними подіями, а також комплексну систему управління задачами, включаючи створення, редагування, видалення та автоматичну

генерацію задач на основі розкладу. Застосунок також підтримує локалізацію для забезпечення зручності використання різними користувачами.

Тестування застосунку було зосереджено на перевірці коректності взаємодії з зовнішнім API та обробці різних сценаріїв відповіді сервера, що підтвердило надійність ключових функцій отримання даних.

Результатом роботи є функціональний кросплатформний застосунок, який інтегрує можливості календаря та списку задач, адаптований до потреб студентів ХНУРЕ, та вирішує виявлені проблеми існуючих рішень. Застосунок надає зручний інструмент для планування навчальної діяльності, ефективно поєднуючи офіційний розклад з особистими планами та завданнями. Хоча поточна версія задовольняє основні потреби, існують перспективи для подальшого розвитку, зокрема впровадження хмарної синхронізації даних для забезпечення доступу з різних пристроїв та резервного копіювання.

Таким чином, поставлені завдання було виконано в повному обсязі, а розроблений застосунок демонструє потенціал для практичного застосування та подальшого вдосконалення.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Zarichuk O. Comparative analysis of frameworks for mobile application development: native, hybrid, or cross-platform solutions. Вісник Черкаського державного технологічного університету. 2023. Vol. 28, no. 4. P. 19–27. URL: https://doi.org/10.62660/2306-4412.4.2023.19-27 (date of access: 08.06.2025).

2. Gerges M., Elgalb A. Comprehensive comparative analysis of mobile apps development approaches. Journal of artificial intelligence general science (JAIGS) ISSN:3006-4023. 2024. Vol. 6, no. 1. P. 430–437. URL: https://doi.org/10.60087/jaigs.v6i1.269 (date of access: 08.06.2025).

3. Lano K., Yassipour Tehrani S. Introduction to clean architecture concepts. Introduction to software architecture. Cham, 2023. P. 35–49. URL: https://doi.org/10.1007/978-3-031-44143-1_2 (date of access: 08.06.2025).

4. JavaScript in mobile applications: React native vs ionic vs NativeScript vs native development / H. Brito et al. 2018 13th iberian conference on information systems and technologies (CISTI), Caceres, 13–16 June 2018. 2018. URL: https://doi.org/10.23919/cisti.2018.8399283 (date of access: 22.05.2025).

5. Mukhamedin A. A., Abitova G. A. Research of a react native vs flutter, cross-platform mobile application frameworks. Bulletin of shakarim university. technical sciences. 2024. Vol. 1, no. 2(14). P. 26–33. URL: https://doi.org/10.53360/2788-7995-2024-2(14)-4 (date of access: 22.05.2025).

6. Architecture Overview · React Native. React Native. URL: https://reactnative.dev/architecture/overview (date of access: 09.06.2025).

7. Classon I. Migrating from Xamarin.Forms to .NET MAUI. Berkeley, CA: Apress, 2025. URL: https://doi.org/10.1007/979-8-8688-1215-6 (date of access: 22.05.2025).

8. Kozub Y., Kozub H. Features of multiplatform application development on Kotlin. Herald of Khmelnytskyi National University. Technical sciences. 2023. Vol. 317, no. 1. P. 224–229. URL: https://doi.org/10.31891/2307-5732-2023-317-

1-224-229 (date of access: 22.05.2025).

9. Flutter architectural overview. Docs | Flutter. URL: https://docs.flutter.dev/resources/architectural-overview (date of access: 09.06.2025).

10. Clean architecture: a craftsman's guide to software structure and design. Pearson, 2017. 432 p.

11. Patterns of enterprise application architecture. Addison-Wesley, 2011. 560 p.

12. An exploratory study of MVC-based architectural patterns in Android apps / A. Daoudi et al. SAC '19: the 34th ACM/SIGAPP symposium on applied computing, Limassol Cyprus. New York, NY, USA, 2019. URL: https://doi.org/10.1145/3297280.3297447 (date of access: 22.05.2025).

13. Design patterns: elements of reusable object-oriented software (addison-wesley professional computing series). Addison-Wesley Professional, 1995. 395 p.

14. Implementing domain-driven design. Addison-Wesley Professional, 2012.

15. Dart overview. Dart programming language | Dart. URL: https://dart.dev/overview (date of access: 08.06.2025).

16. Sinha S. Quick start guide to dart programming. Berkeley, CA : Apress, 2020. URL: https://doi.org/10.1007/978-1-4842-5562-9 (date of access: 22.05.2025).

17. Feiler J. Introducing sqlite for mobile developers. Berkeley, CA : Apress, 2015. URL: https://doi.org/10.1007/978-1-4842-1766-5 (date of access: 22.05.2025).

18. Аналіз та використання способів керування станом застосунку у Flutter / Ю. Ю. Коба та ін. Збірник наукових праць Харківського національного університету Повітряних Сил. 2023. № 2 (76). С. 75–81. URL: https://doi.org/10.30748/zhups.2023.76.09 (дата звернення: 22.05.2025).

19. Aditya H. Accelerating cross-platform mobile development through modular architecture: insights from google chat’s architecture overhaul.

International journal of computer engineering and technology. 2024. Vol. 15, no. 6. P. 1576–1588. URL: https://doi.org/10.34218/ijcet_15_06_132 (date of access: 08.06.2025).

20. Bloc architecture. Bloc. URL: https://bloclibrary.dev/architecture/ (date of access: 08.06.2025).

21. Drift documentation. URL: https://drift.simonbinder.eu/ (date of access: 22.05.2025).

22. Zimmerle C., Gama K. On the usability of reactive programming apis: a mixed evaluation. Software: practice and experience. 2025. URL: https://doi.org/10.1002/spe.3435 (date of access: 08.06.2025).

23. Shared_preferences - Dart API docs. The official repository for Dart and Flutter packages. URL: https://pub.dev/documentation/shared_preferences/ (date of access: 08.06.2025).

24. Rxdart - Dart API docs. The official repository for Dart and Flutter packages. URL: https://pub.dev/documentation/rxdart/ (date of access: 08.06.2025).

25. Equatable - Dart API docs. The official repository for Dart and Flutter packages. URL: https://pub.dev/documentation/equatable/ (date of access: 08.06.2025).

26. Json_serializable - Dart API docs. The official repository for Dart and Flutter packages. URL: https://pub.dev/documentation/json_serializable/ (date of access: 08.06.2025).

27. Dart Code - Dart & Flutter support for Visual Studio Code. URL: https://dartcode.org/ (date of access: 22.05.2025).