МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА

КУЦМАЙ ВЛАДИСЛАВ ЯРОСЛАВОВИЧ

Допускається до захисту:
в.о. завідувача кафедри
інформаційних технологій
канд. техн. наук, доцент
_____ О. В. Зелінська
«_____» _____ 20__ р.

МОБІЛЬНИЙ ДОДАТОК ДЛЯ СИНХРОНІЗАЦІЇ ОСОБИСТОГО КАЛЕНДАРЯ ТА РОЗКЛАДУ ЗАНЯТЬ

Спеціальність 122 «Комп'ютерні науки»

Кваліфікаційна (бакалаврська) робота

Керівник:
Антонов Ю.С., кан. фіз.-мат. наук, доцент,
доцент кафедри інформаційних
технологій

_____
(Підпис)

Оцінка _____/_____/_____
(бали/за шкалою ЄКТС/за національною шкалою)
Голова ЕК: _____
(Підпис)

Вінниця 2025

АНОТАЦІЯ

Куцмай В.Я. Розробка мобільного додатку для синхронізації особистого календаря та розкладу занять за допомогою Web-API. – Кваліфікаційна (бакалаврська) робота за спеціальністю 122 Комп'ютерні науки, Донецький національний університет імені Василя Стуса, Вінниця, 2025.

У роботі запропоновано архітектуру Android-застосунку, що дозволяє синхронізувати розклад занять з Google Calendar за допомогою Web-API. Реалізовано автоматичне зчитування розкладу з HTML-таблиці, його локальне збереження та інтеграцію з календарем користувача.

Ключові слова: мобільний додаток, Android, Web-API, синхронізація, розклад, календар.

Kutsmay V.Y. Development of a Mobile Application for Synchronizing a Personal Calendar and Class Schedule Using Web-API. – Bachelor's qualification thesis in 122 Computer Science, Vasyl' Stus Donetsk National University, Vinnytsia, 2025.

The thesis presents an Android application architecture for synchronizing class schedules with Google Calendar using Web-API. The system automates schedule parsing from HTML tables, stores data locally, and integrates events with the user's calendar.

Keywords: mobile application, Android, Web-API, synchronization, schedule, calendar.

ЗМІСТ

ВСТУП

Цифровізація освіти та зростання обсягів інформації, з якою щоденно працює студент, зумовлюють потребу в ефективному управлінні власним часом. Одним із ключових інструментів такого управління є розклад занять, який має бути доступним, актуальним та інтегрованим з особистими планувальниками. Проте на практиці розклад у більшості університетів все ще надається у форматі, незручному для автоматичного перенесення в персональні календарі, що створює бар'єри для організованого навчального процесу.

Сучасні мобільні платформи та хмарні сервіси, зокрема Google Calendar, Apple Calendar, Outlook тощо, пропонують широкі можливості для створення та синхронізації подій. У той же час, розклад навчальних занять часто існує в окремій цифровій системі, з якою ці календарні сервіси не інтегровані. Це змушує студентів або вручну дублювати розклад у власних календарях, або користуватися ізольованими додатками без підтримки повної синхронізації.

Актуальність теми обумовлена необхідністю розробки мобільного додатку, який дозволить автоматично синхронізувати університетський розклад занять з особистим календарем користувача через Web-API. Таке рішення мінімізує ризик помилок, сприяє своєчасному інформуванню про зміни та підвищує загальний рівень організованості. Зокрема, враховуючи, що більшість студентів вже використовують мобільні пристрої для ведення календарів, мобільний застосунок з повноцінною інтеграцією розкладу створює нову якість в управлінні навчальним часом.

Об'єктом дослідження виступає процес синхронізації календарних даних між інформаційними системами вищих навчальних закладів та персональними календарями користувачів. Предметом є методи, моделі та технології реалізації цієї синхронізації у вигляді мобільного застосунку для Android-платформи.

Метою цієї роботи є розробка мобільного додатку, який дозволяє користувачеві після автентифікації отримати розклад занять зі стороннього джерела (зокрема з HTML-таблиці ВНТУ), автоматично синхронізувати його з Google Calendar, та зберігати локально інформацію для офлайн-доступу.

Для досягнення поставленої мети було вирішено реалізувати застосунок на основі архітектури Clean Architecture з використанням патерну MVI. В якості інструментів реалізації обрано мову Kotlin, UI-фреймворк Jetpack Compose, бібліотеки Room, Retrofit, Firebase та Google Calendar API.

У першому розділі роботи проведено аналіз предметної області, сучасних підходів до інтеграції розкладу з календарними сервісами, можливостей Web-API та обґрунтовано необхідність розробки програмного рішення. У другому розділі описано архітектурні підходи, вибрані технології, логіку побудови UI та структуру програмного коду. У третьому – реалізацію, тестування та аналіз ефективності синхронізації розкладу з особистим календарем користувача.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ РІШЕНЬ

1.1 Сучасні методи управління розкладом занять та їх інтеграція з календарними сервісами

Розклад навчальних занять є ключовим елементом організації освітнього процесу для студентів і викладачів. Одним з найважливіших умінь, що визначають успішність студента, є ефективне управління часом (зокрема планування розкладу) [7, 40]. Традиційно розклад занять доводився до користувачів у вигляді друкованих бюлетенів або статичних електронних таблиць, які студенти змушені були переносити в особисті календарі вручну. З розвитком цифрових технологій все більше закладів освіти впроваджують електронні системи розкладу: інформацію про заняття можна переглядати через веб-портал або мобільний застосунок університету [50]. Наприклад, факультет наук Сіднейського університету створив електронний календар оцінювання, який автоматично збирає дані з навчальних планів та дозволяє студентам відстежувати дедлайни за допомогою улюблених календарних програм [6]. Сучасні підходи до управління розкладом забезпечують не лише зручне відображення занять на різних пристроях, а й можливість автоматичної синхронізації розкладу з персональним календарем користувача [25].

На сьогодні існують різноманітні способи цифрового керування розкладом занять. Багато закладів вищої освіти впроваджують власні електронні системи, доступні через веб-інтерфейс або офіційний мобільний додаток, що дозволяють студентам переглядати актуальний розклад своєї групи онлайн [32, 38]. Деякі університети [47, 10] надають функцію експорту розкладу у стандартизованому форматі iCalendar (*.ics) [18], що дає змогу користувачам підписатися на розклад через популярні календарні сервіси (Google Calendar, Apple Calendar, Microsoft Outlook тощо). У разі підписки на такий веб-календар розкладу всі зміни (наприклад, перенесення занять або заміна аудиторії) автоматично відображаються в особистому календарі студента без необхідності оновлювати

дані вручну [9]. Окрім університетських порталів, поширеними є універсальні мобільні застосунки для управління розкладом занять. Вони дозволяють студентам самостійно формувати свій розклад: додавати пари, налаштовувати нагадування та переглядати розклад у зручному форматі на смартфоні. Прикладом є популярний додаток MyStudyLife, що надає можливості планування занять, завдань і іспитів та синхронізується між пристроями користувача [36]. Багато таких застосунків підтримують інтеграцію з календарями пристроїв: зокрема, можуть експортувати створений розклад у системний календар OS, завдяки чому заняття відображаються поряд з іншими подіями особистого календаря. Популярні онлайн-календарі (Google Calendar, Apple Calendar тощо) також можуть бути використані для ведення навчального розкладу шляхом імпорту подій. Користувач може вручну створити окремий календар для занять і внести туди розклад на семестр або ж імпортувати готовий файл розкладу у форматі *.ics, якщо навчальний заклад його надає. Багато сервісів підтримують налаштування повторюваних подій (наприклад, щотижневих лекцій або пар по парних/непарних тижнях), що значно спрощує введення розкладу в електронний календар. Крім того, сучасні календарні платформи дозволяють спільне використання календарів: студент може отримати розклад, опублікований кафедрою або старостою групи, просто підключившись до відповідного спільного календаря. Таким чином, значна частина студентів активно користується цифровими планувальниками: за даними нещодавнього опитування, 65% студентів коледжів у США щоденно застосовують від 6 до 15 різних цифрових інструментів, а майже половина регулярно користується засобами планування розкладу чи календарями [11].

Інтеграція розкладу занять з календарними сервісами забезпечує низку переваг. По-перше, це централізація інформації: всі навчальні заходи відображаються поряд з особистими подіями (зустрічами, дедлайнами, святами) в єдиному календарному просторі, що сприяє цілісному баченню своїх планів [25]. По-друге, завдяки функціям нагадувань і сповіщень користувач своєчасно отримує інформацію про майбутні заняття на своїх пристроях (смартфон, ПК,

смарт-годинник), що підвищує пунктуальність і організованість [48]. По-третє, синхронізація усуває помилки, пов’язані з ручним переписуванням розкладу, та гарантує актуальність даних у разі змін. Дослідження показують, що використання автоматизованих засобів планування покращує ефективність навчального процесу та задоволеність студентів розкладом [41]. Водночас реалізація повноцінної інтеграції розкладу з календарними сервісами потребує відповідних технічних рішень. Для автоматичного оновлення розкладу в календарі необхідно налагодити канал передачі даних між системою розкладу та календарним сервісом. На практиці це здійснюється або через публікацію календаря розкладу на спеціальному URL-адресі (для підписки за протоколом iCalendar, визначеним у RFC 5545 [43]), або через використання програмних інтерфейсів (API) самих календарних платформ, які дозволяють зовнішнім застосункам додавати, змінювати та видаляти події у календарі користувача. У підрозділі 1.2 детальніше розглянуто можливості Web-API для такої автоматизованої синхронізації даних розкладу з особистими календарями.

1.2 Аналіз Web-API та можливості їх використання для синхронізації даних

Web-API (веб-інтерфейси програмування застосунків) відіграють ключову роль у забезпеченні взаємодії між різними програмними системами через Інтернет. Web-API визначає набір правил і протоколів, за якими одна програма може запросити дані або функції в іншої програми по мережі. Сучасні Web-API здебільшого базуються на архітектурі REST (Representational State Transfer), запропонованій Р. Філдінгом у 2000 році [13], та інших веб-стандартах. REST-інтерфейси оперують форматами даних, зручними для обміну – найчастіше це JSON, рідше XML – що спрощує інтеграцію різнорідних систем. Дослідження відзначають, що RESTful-підходи стали домінувати над важчими SOAP-службами завдяки простоті та гнучкості у клієнт-серверній взаємодії [39].

У контексті синхронізації персонального календаря з розкладом занять Web-API можуть використовуватися з двох сторін: з боку джерела даних

розкладу і з боку календарного сервісу. По-перше, джерело розкладу (наприклад, інформаційна система університету) може надавати API, через який сторонній додаток отримує актуальний розклад у машиночитному форматі. Це може бути публічний REST API, що повертає JSON-дані з розкладом для конкретної академічної групи або студента. Якщо ж такого API немає, джерело може надавати файл-експорт розкладу (наприклад, ICS або CSV) за певною веб-адресою – фактично, це теж варіант веб-інтерфейсу для отримання даних, хоча і менш гнучкий. По-друге, самі календарні сервіси (Google Calendar, Microsoft Outlook, iCloud та інші) відкривають власні Web-API для сторонніх розробників. Через такі API мобільний додаток може автоматично додавати події (заняття) до календаря користувача, оновлювати або вилучати їх при змінах у розкладі, а також читати події з календаря для реалізації двосторонньої синхронізації. Наприклад, Google надає розробникам RESTful API для Google Calendar, яке підтримує повний набір операцій CRUD (створення, читання, оновлення, видалення) для подій у календарях користувачів [15].

Для роботи з Google Calendar API застосунок повинен пройти OAuth 2.0 авторизацію (де-факто стандарт обмеженого доступу OAuth 2.0 затверджений як RFC 6749 [44]), щоб отримати від користувача дозвіл на управління його календарем. Після цього додаток може надсилати HTTPS-запити до служби: зокрема, HTTP POST із даними нового заняття (назва, час початку та закінчення, опис, локація) додає подію до календаря, HTTP DELETE вилучає скасоване заняття, а HTTP GET отримує список поточних подій. Дані при цьому передаються у форматі JSON, структура якого відповідає полям об’єкта події (тема, часові мітки, параметри повторюваності тощо). Аналогічні API існують і для інших платформ: Microsoft Graph API дозволяє програмно працювати з календарем Outlook/Office 365, а Apple Calendar підтримує інтеграцію через власний API iCloud або стандартний протокол CalDAV.

Використання Web-API для синхронізації даних розкладу має істотні переваги. Передусім, воно дає змогу автоматизувати процес оновлення інформації: коли в системі-джерелі вносяться зміни (додано нове заняття,

змінено час чи аудиторію), додаток через API отримує ці зміни й одразу відображає їх у календарі користувача. Таким чином досягається актуальність розкладу без жодного ручного втручання. Крім того, API відкриває можливості для гнучких сценаріїв: наприклад, додаток може періодично перевіряти наявність оновлень або отримувати сповіщення (webhook) про зміну розкладу, а також попереджати користувача про конфлікти між навчальними заняттями та особистими подіями, зчитуючи дані з обох календарів. Очевидно, що подібна автоматизація підвищує ефективність управління часом студента і знижує ймовірність пропусків чи накладок занять [41].

Відкриті Web-API дозволяють створити програмний доступ до даних розкладу занять та функцій календарних сервісів, що є необхідним для автоматичної синхронізації інформації між ними.У контексті синхронізації особистого календаря з розкладом занять Web-API можуть використовуватися з двох сторін: з боку джерела даних розкладу та з боку календарного сервісу. По-перше, джерело розкладу (наприклад, інформаційна система університету) може надавати API, через яке можна отримати актуальний розклад занять у машиночитному форматі. Це може бути публічний REST API, що повертає, скажімо, JSON-дані з розкладом для конкретної групи або студента. Альтернативно, якщо такого API немає, джерело може надавати файл-експорт розкладу (ICS або CSV) по URL, який теж можна розглядати як простий веб-інтерфейс для отримання даних. По-друге, календарні сервіси (такі як Google Calendar, Microsoft Outlook Calendar, iCloud Calendar тощо) надають власні Web-API для сторонніх розробників. Через ці API мобільний додаток може автоматично додавати події (заняття) до календаря користувача, оновлювати або видаляти їх при змінах, а також читати події з календаря для двосторонньої синхронізації.

Важливим аспектом є забезпечення безпеки при роботі з Web-API. Оскільки мобільний додаток отримує доступ до особистих даних користувача (його приватного календаря) і, можливо, до захищених даних університетської системи розкладу, необхідно використовувати сучасні протоколи автентифікації

та авторизації. Як вже зазначалося, OAuth 2.0 сьогодні є загальноприйнятим стандартом надання додатку обмеженого доступу до ресурсів користувача без передачі йому пароля [44]. Також слід забезпечити обмін даними через захищене з'єднання (HTTPS) з використанням шифрування, щоб унеможливити перехоплення чутливої інформації. Дослідники наголошують, що у впровадженні дистанційних освітніх сервісів на мобільних платформах питання захисту персональних даних є вкрай важливим, і порушення безпеки може звести нанівець переваги таких технологій [24].

Отже, Web-API надають необхідний інструментарій для побудови мосту між системою розкладу та особистим календарем студента. Коректне використання цих інтерфейсів дозволяє досягти високого рівня автоматизації: дані про розклад занять, доступні через API, можуть оперативно передаватися до календарного сервісу, завдяки чому кінцевий користувач завжди має під рукою актуальну інформацію про свої навчальні заняття. Практика показує, що впровадження таких інтеграційних рішень позитивно впливає на навички тайм-менеджменту студентів і їх академічні результати [31].

1.3 Визначення основних проблем та обґрунтування необхідності розробки мобільного додатку

Незважаючи на наявність описаних підходів і інструментів для роботи з розкладом занять, аналіз предметної області виявляє низку невирішених проблем, що створюють передумови для розробки нового програмного рішення. По-перше, багато навчальних закладів досі надають розклад у статичному вигляді (PDF-документ, дошка оголошень, таблиця на сайті), який не забезпечує автоматичної синхронізації з особистими календарями студентів. Студентам доводиться вручну переносити інформацію про пари в свої електронні календарі, витрачаючи час та ризикуючи припуститися помилок. Така ситуація знижує ефективність планування навчального часу і може призводити до пропусків занять через людський фактор [25, 29]. Дослідження підтверджують, що нестача контролю над своїм розкладом та перевантаження адміністративними

завданнями негативно позначаються на рівні стресу студентів та їх успішності [33]. Таким чином, відсутність автоматизації ускладнює студентам раціональне управління своїм часом.

По-друге, навіть якщо заклад пропонує цифровий розклад чи файл для експорту, користувач не завжди має зручний інструмент для його використання. Наприклад, підписка на календар розкладу через URL потребує певних технічних навичок: необхідно знайти відповідне покликання на сайті, додати його до свого календарного застосунку. Не всі студенти обізнані з такою можливістю чи вміють нею скористатися. В результаті значна частина аудиторії продовжує застосовувати ручні методи або окремі додатки, що не інтегруються з їх основним календарем. Дослідники відзначають, що для успішного впровадження цифрових інструментів планування необхідно враховувати фактор зручності та інтерфейсних рішень: інтуїтивно зрозумілий інтерфейс підвищує залученість користувачів і частоту використання застосунку [17]. Відсутність же легкої і зрозумілої для пересічного студента інтеграції розкладу з наявними засобами планування призводить до того, що потенціал технологій використовується не повністю.

По-третє, універсальні календарні сервіси (Google Calendar, Outlook тощо), хоч і дозволяють вручну додавати навчальні події, проте не надають спеціалізованих функцій, врахованих під потреби навчального процесу. Зокрема, вони не підтримують без додаткових налаштувань індивідуальні особливості розкладу (поділ тижнів на «червоний/синій», скорочені заняття, позапланові заміни) і не пов’язані напряму з навчальними ресурсами (курсами, викладачами). Спеціалізовані студентські планувальники частково вирішують цю проблему, однак часто працюють як ізольовані системи. Якщо додаток не синхронізується із зовнішнім календарем, студент має дві розрізнені картини: окремо навчальний розклад у додатку і окремо особисті справи у календарі. Така фрагментація незручна, оскільки змушує перемикатися між програмами і тримати у пам’яті два списки подій. За результатами згаданого опитування, 65% студентів щодня користуються багатьма цифровими сервісами, при цьому близько половини

застосовують засоби планування – отже, існує ризик дублювання функціоналу та розпорошення уваги між кількома додатками [11]. В ідеалі потрібне єдине рішення, що об’єднає академічний розклад із особистим плануванням.

Ще одна проблема – відсутність універсального рішення, адаптованого під специфічні потреби конкретного закладу чи користувача. Загальнодоступні комерційні додатки можуть не підтримувати імпорт розкладу з тієї системи, яку використовує даний університет, або не враховувати локальний контекст (мову інтерфейсу, формат часу, структуру навчального тижня тощо). Тому виникає необхідність у розробці кастомізованого мобільного додатку, який враховуватиме особливості розкладу саме цього навчального закладу і забезпечуватиме гнучку синхронізацію з персональним календарем студента.

Отже, основні виявлені проблеми можна звести до наступних: відсутність повної автоматизації перенесення розкладу в особистий календар, складність використання наявних часткових рішень для пересічного студента та фрагментованість інформаційних платформ. Усе це обґрунтовує доцільність створення нового мобільного додатку для синхронізації особистого календаря з розкладом занять. Такий додаток має заповнити зазначені прогалини: надати користувачу простий механізм отримання актуального розкладу, автоматично відображати навчальні заняття у його власному календарі, враховувати специфіку навчального процесу і загалом підвищити зручність планування. Розробка спеціалізованого рішення дозволить покращити тайм-менеджмент студентів і викладачів, мінімізувати ймовірність помилок та пропусків, а також сприятиме подальшій цифровій трансформації освітнього процесу в університеті. Доведено, що вдосконалення навичок управління часом позитивно впливає на академічну успішність студентів [7, 26], а впровадження технологій для підтримки планування навчання допомагає студентам краще адаптуватися до вимог самостійної роботи у закладі вищої освіти [6]. Таким чином, розробка інтеграційного мобільного застосунку є своєчасною і необхідною для підвищення ефективності організації навчального процесу.

РОЗДІЛ 2. АРХІТЕКТУРА ТА ІНСТРУМЕНТИ РОЗРОБКИ МОБІЛЬНОГО ЗАСТОСУНКУ

2.1 Вимоги до програмного забезпечення

Мобільний додаток для синхронізації розкладу занять із персональним календарем має відповідати низці вимог, що забезпечують його функціональність, стабільність, зручність використання та безпечну інтеграцію з зовнішніми сервісами. Вимоги сформульовано на основі цільового сценарію використання додатку студентами та особливостей платформи Android.

З функціональної точки зору, застосунок повинен реалізовувати повний цикл роботи з даними розкладу: від отримання розкладу з веб-сайту навчального закладу до інтеграції подій у календар користувача. Основними функціями є автентифікація через обліковий запис Google, автоматичне формування розкладу на основі парсингу HTML-таблиць, збереження інформації в локальній базі даних, а також синхронізація з Google Calendar через REST API. Окрім цього, має бути передбачена можливість керування подіями, підтримка офлайн-доступу та адаптація інтерфейсу до вибору теми (світлої або темної).

До нефункціональних вимог належать критерії, що впливають на якість роботи додатку. Інтерфейс має бути інтуїтивно зрозумілим, адаптивним до різних типів екранів, побудованим відповідно до принципів Material Design. Важливою є стабільність і швидкодія: основні операції (зчитування розкладу, синхронізація, перегляд подій) повинні виконуватися без затримок. Система кешування повинна гарантувати доступ до розкладу навіть за відсутності інтернету. Безпека взаємодії з обліковим записом користувача забезпечується через використання протоколу OAuth 2.0, який виключає необхідність передачі паролів до сторонніх сервісів.

З точки зору технічної реалізації, програмне забезпечення має відповідати таким обмеженням:

- Підтримка операційної системи Android версії не нижче 8.0 (API level 26);
- Доступ до Google Calendar API можливий лише при наявності облікового запису Google;

- Для роботи з календарем необхідне стабільне інтернет-з’єднання на момент синхронізації;
- Можливість роботи в офлайн-режимі повинна реалізовуватися через локальну базу даних (Room);
- Взаємодія з зовнішніми джерелами розкладу передбачається у форматі HTML.

Таким чином, сформульовані вимоги визначають як функціональні можливості, що повинні бути реалізовані в додатку, так і технічні аспекти, пов’язані з продуктивністю, надійністю, безпекою та зручністю використання. Їх дотримання є ключовою умовою успішного впровадження мобільного рішення для студентів, яке має інтегрувати академічний розклад із персональним календарем у єдиний цифровий інтерфейс.

2.2 Вибір архітектурного підходу

Для забезпечення масштабованості та підтримуваності застосунку обрано архітектуру Clean Architecture [30] Це багаторівневий підхід, що розділяє систему на ізольовані шари з чіткими відповідальностями та односпрямованою залежністю – зовнішні шари можуть залежати від внутрішніх, але не навпаки [12]. Класично виділяються такі рівні: сутності (Entities) – ядро бізнес-логіки, use cases – правила застосунку, інтерфейсні адаптери (Controllers, Gateways, Presenters) – перетворювачі даних між бізнес-логікою та зовнішніми системами, та зовнішній шар фреймворків і драйверів (UI, база даних, сервіси) [8]. Така багатошарова архітектура сприяє розділенню відповідальностей і слабкому зв’язуванню компонентів, що полегшує модифікацію і тестування коду [37]. Внутрішні бізнес-правила не залежать від деталей реалізації зовнішніх систем – це дозволяє змінювати UI або базу даних без переписування ядра логіки [22]. Як наслідок, система стає більш гнучкою до змін та легшою в супроводі.

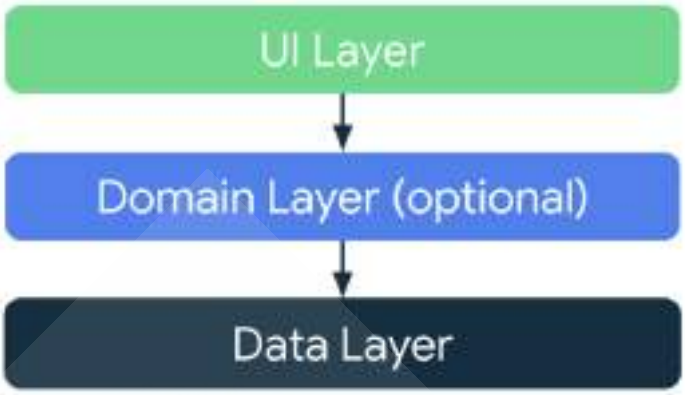


Рисунок 2.1 – Приклад багатошарової архітектури застосунку (UI, Domain, Data), де стрілки відображають односпрямовані залежності від зовнішніх шарів до внутрішніх

У даному проєкті принципи Clean Architecture реалізовано через поділ коду на шари: Presentation (UI), Domain (доменна логіка) та Data (робота з даними). Такий підхід дозволяє досягти високої модульності, незалежності бізнес-логіки від зовнішніх фреймворків та полегшити тестування і підтримку системи. Згідно з теоретичними основами, викладеними в роботах Роберта Мартіна [30], кожен шар має чітко визначену відповідальність та взаємодіє з іншими шарами виключно через абстракції. На UI-шарі зосереджено відображення даних та взаємодію з користувачем, він містить лише код інтерфейсу і мінімум логіки. Логіка UI реагує на зміни стану і події користувача, делегуючи обробку у внутрішні шари. Domain-шар містить бізнес-логіку застосунку – алгоритми синхронізації розкладу, правила обробки даних розкладу занять, перевірки тощо. Цей шар ізольований від деталей реалізації зберігання чи мережі і оперує абстракціями (інтерфейсами репозиторіїв, use-case класами). Data-шар відповідає за доступ до джерел даних: локальної бази даних та зовнішніх API. Він містить реалізацію репозиторіїв, моделі даних (DTO, Entity) і перетворювачі даних. Таким чином, застосовано принцип єдиного джерела правди (Single Source of Truth) – кожен тип даних має єдине місце зберігання і управління, що гарантує узгодженість станів. Взаємодія між шарами відбувається через інтерфейси: наприклад, UI шар викликає методи Domain-шару (use case) для виконання дій, а Domain звертається до Data-шару через інтерфейс репозиторію. Це гарантує, що внутрішня логіка не залежить від зовнішніх фреймворків.

Додатково, використано архітектурний шаблон MVI (Model-View-Intent), який забезпечує унідирекційний потік даних і чітке розділення стану інтерфейсу та подій . На відміну від класичного MVVM, де стан може зберігатися і в UI, і у ViewModel, в MVI усі зміни стану інтерфейсу представлені єдиною моделлю стану (ViewState) – одним джерелом правди для UI [5, 4]. Користувацькі дії оформлені як інтенти (Intent) – події, що надходять від View до ViewModel, але не плутати з Android Intents – тут ідеться про наміри виконати дію в логіці застосунку. ViewModel виступає посередником: отримує Intents, обробляє їх (викликає відповідні use-case в Domain-шарі або репозиторії), та формує новий стан Model (ViewState), який потім унідирекційно передається назад у View для відображення. Такий цикл Intent → Model(State) → View забезпечує реактивне оновлення UI при зміні даних і спрощує відстеження станів. MVI сприяє тому, що кожен стан інтерфейсу є детермінованим результатом певної послідовності подій, а логіка оновлення зосереджена у ViewModel (ред'юсер стану). Це підвищує передбачуваність роботи інтерфейсу та полегшує тестування – можна перевіряти, що на певну послідовність інтеракцій формується очікуваний стан UI. Шаблон MVI добре узгоджується з Clean Architecture, оскільки гарантує єдиний потік даних і чітке розділення відповідальностей між шарами, зменшуючи побічні ефекти та можливі помилки синхронізації стану.

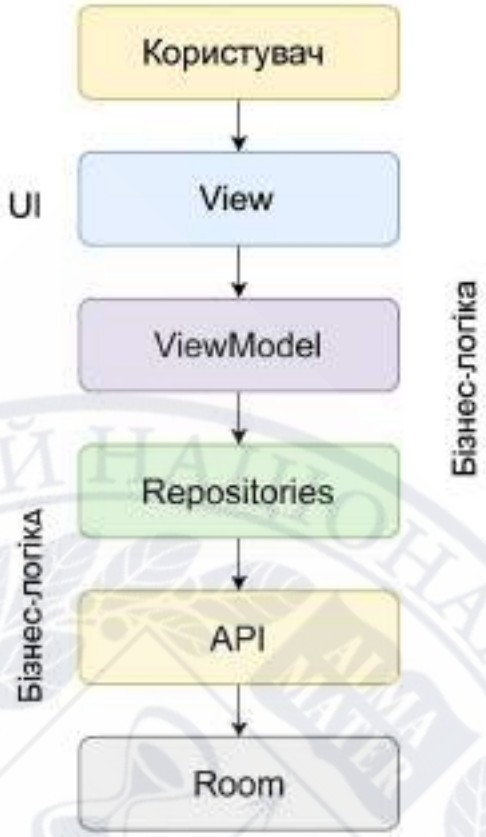


Рисунок 2.2 – Архітектура мобільного додатку на основі Clean Architecture з патерном MVI

Отже, комбінація Clean Architecture і патерну MVI забезпечила чистий код із чіткою структурою, де бізнес-логіка ізольована від платформенних деталей, а інтерфейс реагує на зміни єдиного джерела стану. Це значно спрощує модифікацію (наприклад, додавання нового джерела даних або зміна UI) і поліпшує тестованість – бізнес-правила можна перевіряти окремо від UI та бази даних. Вибір такої архітектури обґрунтований потребою у довготривалій підтримці застосунку, який буде легко розширювати під нові вимоги (наприклад, зміну формату розкладу чи інтеграцію з іншими календарями) без повної переробки.

2.3 Архітектурна структура мобільного додатку та UML-діаграми

Мобільний застосунок спроєктовано за принципами чистої архітектури (Clean Architecture), у якій код поділено на взаємозалежні шари із чіткими відповідальностями. Зокрема, виокремлено Presentation-шар (UI), Domain-шар

(доменна логіка) та Data-шар (доступ до даних). Згідно з правилом односпрямованої залежності, зовнішні шари залежні від внутрішніх, але не навпаки. Патерн MVI (Model-View-Intent) гарантує унідирекційний потік даних у Presentation-шарі: View надсилає наміри (Intent) у ViewModel, який викликає відповідний use case із Domain-шару та формує новий стан екрану (ViewState) для оновлення UI. У такій моделі дані та логіка оновлення ізольовані у ViewModel, а UI-стан формується детерміновано за послідовністю подій.

- Presentation (UI): містить компоненти взаємодії з користувачем (екрани, viewmodel'і, моделі стану). Вхідними даними є події (наміреня) користувача, вихідними – оновлений ViewState.
- Domain (бізнес-логіка): включає сутності предметної області (наприклад, Подія – об'єкт календаря з полями «заголовок», «час» тощо) і use case – класи, що реалізують сценарії синхронізації. Domain-шар не залежить від деталей платформи і працює з абстрактними інтерфейсами.
- Data (дані): реалізує інтерфейси репозиторіїв, забезпечує доступ до зовнішніх джерел (Google Calendar API, локальна БД). Тут знаходяться реалізації CalendarRepository (що інкапсулює виклики Google API та локальної бази) та сервіси доступу до даних. Наприклад, GoogleCalendarService формує HTTP-запити до API, а LocalDatabase керує локальною СУБД. Інжекція залежностей і принцип інверсії (Dependency Inversion) гарантують, що Presentation та Domain шари працюють з абстракціями, не знаючи про конкретні реалізації

У межах шару даних (Data), окрім взаємодії із зовнішніми API, реалізовано локальне сховище даних за допомогою бібліотеки Jetpack Room. Це сучасна ORM (Object-Relational Mapping) бібліотека, яка надає абстрактний шар над SQLite для зручної та надійної роботи з локальною базою даних на пристрої Android. Використання Room у шарі даних дозволяє досягти кількох ключових цілей:

- Кешування даних розкладу: Заняття, отримані шляхом парсингу або через API, зберігаються локально. Це забезпечує швидкий доступ до розкладу

при наступних запусках програми та можливість роботи в офлайн-режимі, коли відсутнє інтернет-з'єднання

- Зберігання стану синхронізації: Для кожного заняття, експортованого до Google Календаря, в локальній базі даних зберігається унікальний ідентифікатор події (eventId) з Google Календаря. Це дозволяє відстежувати, які заняття вже синхронізовані, уникати дублювання подій при повторних синхронізаціях та коректно оновлювати або видаляти події з календаря користувача.
- Забезпечення єдиного джерела правди (Single Source of Truth) для кешованих даних: Локальна база даних виступає як основне джерело даних розкладу для відображення в інтерфейсі користувача, що гарантує узгодженість інформації та спрощує управління станом.

Структура локальної бази даних включає таблиці для зберігання інформації про заняття (LessonEntity) з усіма необхідними атрибутами (назва, час, викладач, аудиторія, тип тижня, підгрупа тощо) та таблицю для зв'язку цих занять з подіями Google Календаря (LessonEventEntity), де зберігаються їхні ідентифікатори. Доступ до цих таблиць здійснюється через об'єкти доступу до даних (DAO – LessonDao, LessonEventDao), які інкапсулюють логіку запитів до бази даних та надають відповідні методи для шару доменної логіки (Domain) через інтерфейси репозиторіїв. Це повністю відповідає принципам Clean Architecture, де деталі реалізації джерел даних приховані від бізнес-логіки.

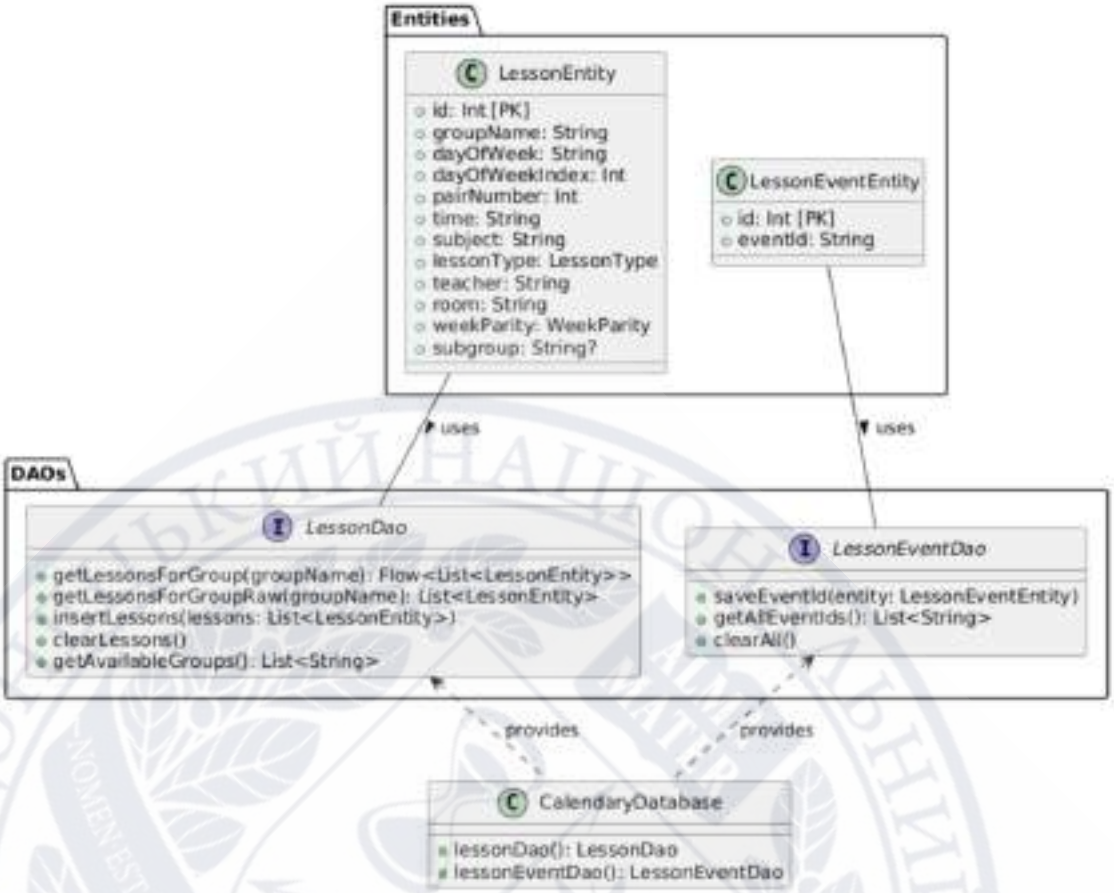


Рисунок 2.3 – Структура локальної бази даних Room

Діаграма розгортання (Рисунок 2.4) візуалізує фізичну конфігурацію системи, показуючи, де саме розгорнуті її компоненти. Вона чітко розмежовує Мобільний пристрій (Client Device), який є вузлом для клієнтської частини застосунку. На цьому пристрої розміщується компонент користувацького інтерфейсу, що відповідає за взаємодію з користувачем, а також локальна база даних, яка слугує для кешування розкладу та швидкого доступу до даних.

Окрім мобільного пристрою, діаграма включає Сервер Google Calendar API (Cloud). Це зовнішній хмарний сервіс, що обробляє HTTP-запити до календаря користувача, виконуючи функції віддаленого сховища та синхронізації. Між мобільним додатком на пристрої та Google Calendar API встановлено мережеве з’єднання, зазвичай через протоколи HTTP/REST.

Такий розподіл компонентів ілюструє, що на боці пристрою виконуються операції, пов'язані з UI та локальними даними, тоді як основні запити на

синхронізацію та взаємодію з глобальними даними надсилаються в хмару. Наприклад, коли користувач хоче створити або отримати подію, мобільний додаток на пристрої посилає HTTP-запит до Google Calendar API. Отримані дані потім відображаються у застосунку, забезпечуючи безперебійний користувацький досвід. Це наочно демонструє конфігурацію, де клієнтські та сервісні компоненти функціонально розподілені між окремими фізичними вузлами – смартфоном та хмарним сервером.

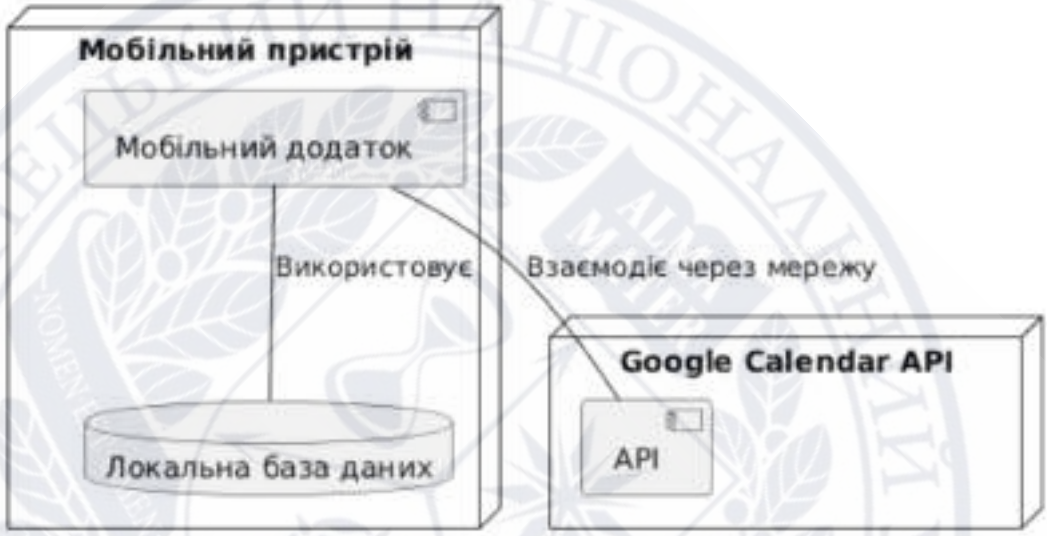


Рисунок 2.4 – Розгортання мобільного додатку з компонентами (застосунок на пристрої, локальна база даних, зовнішній Google Calendar API)

Діаграма класів (Рисунок 2.5) відображає статичну структуру основних класів системи. Серед основних елементів виділяються: клас-сутність Event (подія календаря) з атрибутами, такими як заголовок, час та учасники. Інтерфейс CalendarRepository визначає методи для отримання та оновлення подій. Класи GoogleCalendarService та LocalDatabase реалізують цей інтерфейс, забезпечуючи взаємодію з віддаленим сервісом Google Calendar та локальним сховищем відповідно. Класи шару Presentation, такі як ViewModel, ViewState та Intent, відповідають за представлення даних та обробку взаємодії з користувачем.

Відношення між класами показують залежності: класи Presentation залежать від інтерфейсів Domain (use case), які, своєю чергою, залежать від репозиторію. Репозиторій, у свою чергу, залежить від конкретних Data-класів. На діаграмі класи представлені прямокутниками з розділеними секціями (назва,

атрибути, методи), а зв’язки (залежності, реалізація інтерфейсів) показані відповідними стрілками. Ця модель демонструє, що бізнес-логіка та моделі не прив’язані до деталей реалізації, а класи UI оперують абстрактними сервісами.

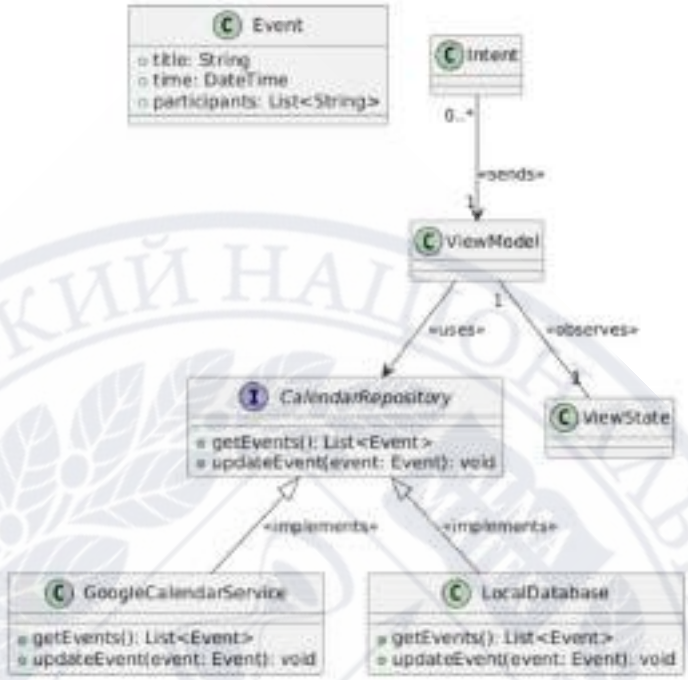


Рисунок 2.5 – UML-діаграма класів: ключові сутності (Event), інтерфейси та класи репозиторію, ViewModel та моделі стану UI

Компонентна діаграма (Рисунок 2.6) є потужним інструментом для візуалізації архітектури програмної системи. Вона допомагає розбити складну систему на високорівневі, функціональні модулі, де кожен компонент відповідає за свою чітко визначену частину функціональності. Ці модулі взаємодіють між собою через строго визначені інтерфейси, що є ключовим для побудови гнучких та масштабованих систем.

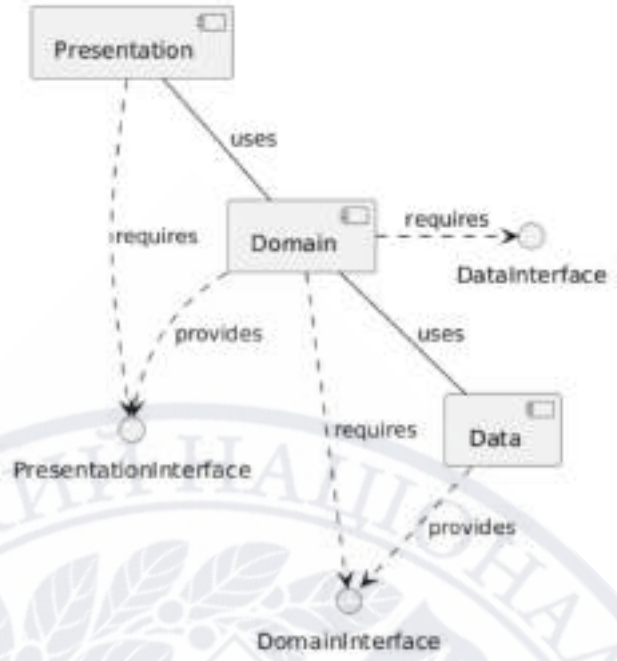


Рисунок 2.6 – UML-діаграма компонентів: високорівневі модулі системи (Presentation, Domain, Data), їх основні інтерфейси і залежності

У типовій системі можна виділити кілька ключових компонентів. Наприклад, UI-компонент відповідає за взаємодію з користувачем, приймаючи його дії та відображаючи поточний стан системи. Компонент бізнес-логіки (Domain) містить основні правила та алгоритми роботи системи, такі як синхронізація розкладу. Компонент взаємодії з Google API керує підключенням та обміном даними із зовнішнім сервісом календаря Google, тоді як компонент локального сховища надає доступ до даних, збережених у локальній базі даних. Крім того, компонент авторизації забезпечує отримання та зберігання OAuth-токенів для безпечного доступу до ресурсів.

Такий підхід значно спрощує масштабування та підтримку системи. Якщо потрібно змінити або навіть повністю замінити один компонент (наприклад, перейти на інший спосіб зберігання даних), це можна зробити, не зачіпаючи решту системи. Це можливо завдяки чітко визначеним контрактам взаємодії між модулями.

На діаграмі компоненти пов'язані стрілками інтерфейсів:

- "Вимагає" (requires): Показує, який інтерфейс компонент потребує для своєї роботи.
- "Надає" (provides): Вказує, який інтерфейс компонент надає іншим модулям.

Ці стрілки та інтерфейси символізують чіткий контракт взаємодії між різними частинами системи, роблячи її більш модульною, гнучкою та легкою для розвитку.

2.4 Архітектура інтеграції з Google Calendar API

Взаємодія з Google Calendar API реалізована через шар даних системи. З точки зору архітектури, робота з цим API проходить за схемою: UI → Domain (use case) → Data (репозиторій) → Google API. Кроки спрощено виглядають так:

1. Користувач ініціює дію (наприклад, синхронізацію календаря або додавання події) у UI, що породжує Intent.
2. ViewModel приймає цей Intent і викликає відповідний use case з доменного шару (наприклад, СинхронізуватиРозклад).
3. Use case звертається до інтерфейсу репозиторію (CalendarRepository) для виконання операції (отримання чи створення подій). Domain шар оперує тільки доменними моделями і не знає деталей доступу.
4. У шарі даних репозиторій реалізує методи, які готують дані до відправки й викликають Google Calendar API. Зокрема, об’єкт «Event» перетворюється у формат HTTP-запиту. GoogleCalendarService (виконувач репозиторію) формує REST-запит (наприклад, POST-запит на метод events.insert або GET на events.list) і відправляє його через мережу.
5. Google Calendar API отримує запит і повертає відповідь (наприклад, створену подію з новим ідентифікатором або список подій). Репозиторій приймає відповідь, перетворює її на доменні об’єкти і повертає у Domain.
6. Domain use case отримує дані і передає їх у ViewModel, який оновлює ViewState. UI, у свою чергу, відображає актуальні дані календаря.

У цій схемі реалізовано кілька архітектурних принципів. Авторизація (OAuth 2.0) організована окремим компонентом (менеджером авторизації) у шарі

даних: під час входу користувача з нього отримується токен доступу, який потім автоматично додається до HTTP-запитів до Google API. Domain шар і UI жодним чином не працюють із токенами напряму – вони інкапсульовані у Data-компоненті. Таким чином, процедура авторизації відділена від бізнес-логіки (принцип єдиної відповідальності), а дані автентифікації захищені. Інкапсуляція доступу: інтерфейс CalendarRepository визначає абстракцію операцій (отримати/створити події) без згадки про те, що використовується HTTP. Шари Presentation та Domain взаємодіють із календарем через цей інтерфейс, не знаючи деталей мережевих викликів або формату запитів. Реалізація в Data-слої (наприклад, GoogleCalendarService) ін’єкцією залежностей (або іншим механізмом IoC) отримує необхідні сервіси і виконує HTTP-запити, реалізуючи принцип інверсії залежностей.

Передача даних між системою та API відбувається у вигляді JSON-об’єктів: наприклад, на стороні застосунку створюється об’єкт Event з полями «summary», «start», «end» тощо (заголовок події, час початку/кінця, список учасників. Потім він відправляється у REST-запиті (наприклад, POST /calendars/{id}/events) і зберігається в Google-календарі. У відповідь приходить інформація про створену подію (з її ID), яка повертається через шар даних і оновлює локальну модель. Усі такі операції охоплені діаграмами: відповідний клас або компонент змальовує інтерфейс (CalendarRepository з методами createEvent, getEvents), а взаємозв’язки між ними показані на діаграмі.

Таким чином, архітектура взаємодії із Google Calendar API реалізована з урахуванням таких принципів: автентифікація і авторизація відокремлені, доступ до даних інкапсульований у Data-шарі, а залежності спрямовані від зовнішніх модулів до внутрішніх (domain). Це забезпечує гнучкість та безпеку: наприклад, за необхідності можна змінити спосіб автентифікації або протестувати логіку, підмінюючи реалізацію CalendarRepository без змін у domain-логіці.

2.5 Інтеграція Google Calendar API

Google Calendar API v3 – це RESTful API для роботи з календарями та подіями користувачів. Його ключові ресурси – Event і Calendar. Event – це одиниця календаря з полями (заголовок, час початку/кінця, учасники тощо), а Calendar – колекція таких подій з власною метаданою. API використовує HTTP-запити до базової URL https://www.googleapis.com/calendar/v3. Наприклад, метод events.insert (HTTP POST /calendars/{calendarId}/events) створює нову подію, а events.list (GET /calendars/{calendarId}/events) повертає список подій у календарі.

Для доступу до API додаток використовує OAuth 2.0 авторизацію. Спочатку розробник отримує облікові дані (client ID/secret) у Google API Console. Потім при запуску програми відбувається авторизація користувача через OAuth: додаток запитує код авторизації, після чого отримує access token від Google Authorization Server. Токен зберігається (і при необхідності поновлюється за допомогою refresh token) і додається до HTTP-запитів (зазвичай у заголовку Authorization). Як зазначено в офіційній документації, «клієнтський додаток запитує у Google Authorization Server токен доступу, отримує його та передає до API». У Android зазвичай використовується GoogleSign-In або окремі бібліотеки OAuth для спрощення цього процесу.

Прикладами ключових запитів є створення нової події та перевірка дублювань. Для додавання події застосовується events.insert. Щоб уникнути випадкового дублювання події (наприклад, при повторному відправленні запиту через помилку мережі), рекомендується генерувати власний унікальний eventId при створенні. Згідно з документацією, вказаний ідентифікатор зберігається, і якщо сервер бачить такий самий eventId при повторному запиті, створення дублікату не відбувається. Якщо власний eventId не вказано, сервер генерує новий, і повторний запит призведе до створення іншої (дубльованої) події. Перевірка наявних подій за певними параметрами (наприклад, по часу чи ідентифікатору) може виконуватися через events.list з необхідними фільтрами.

Google Calendar API накладає квотні обмеження на кількість запитів. Для кожного проєкту та користувача існують ліміти на добу та на хвилину. Якщо

ліміт перевищено, API поверне код помилки 403 або 429. Щоб уникнути цього, рекомендується дотримуватися кращих практик: використання експоненціального backoff при помилках і зменшення частоти запитів (наприклад, використання push-сповіщень про зміни замість постійного опитування). Google також радить розподіляти навантаження рівномірно в часі, аби не створювати піки трафіку.

Інтеграція Calendar API в архітектуру додатку відбувається через шар репозиторію. У Data Layer створено CalendarRepository (наприклад, CalendarRepositoryImpl), який містить методи для виклику API (наприклад, createEvent, getEvents). Цей репозиторій використовує збережений OAuth-токен для авторизації запитів. ViewModel викликає відповідні use cases (наприклад, SyncCalendarUseCase), які у свою чергу звертаються до CalendarRepository. Після виконання API-запиту відповідь може записуватися в локальну базу або відразу транслюватися у стан UI. Детальну взаємодію між компонентами показано на умовній діаграмі компонентів (Рисунок 2.7) і послідовностей (Рисунок 2.8): користувач через інтерфейс ініціює подію (Intent), ViewModel обробляє цей намір, викликає UseCase, який через Repository надсилає запит до Google Calendar API з токеном.

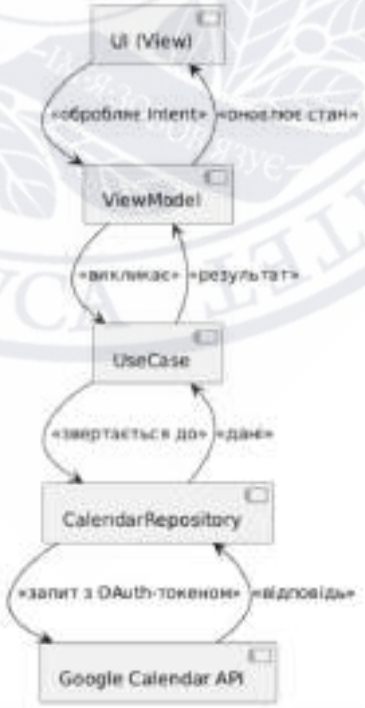


Рисунок 2.7 – Умовна діаграма компонентів: Архітектура інтеграції Google Calendar API

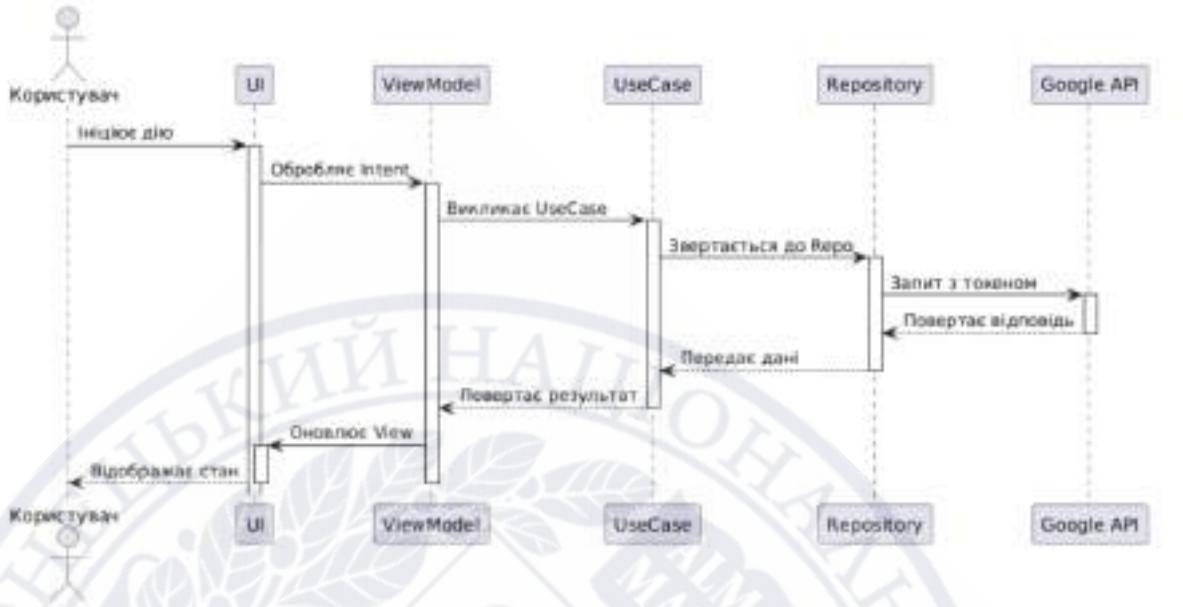


Рисунок 2.8 – Діаграма послідовності: Потік взаємодії з Google Calendar API

Після отримання відповіді дані (наприклад, підтвердження створення події) повертаються по ланцюжку: Repository → UseCase → ViewModel → View, а новий стан відображається користувачу.

Таблиця 2.1 – Основні ресурси Google Calendar API

Ресурс	**Призначення**
Event	Подія календаря (заголовок, час, учасники тощо), репрезентує один об’єкт Event resource.
Calendar	Календар користувача – колекція подій зі своїми властивостями (назва, часовий пояс тощо), репрезентує Calendar resource.
Event List	Збірка подій календаря, що використовується у відповідних методах (наприклад, при запиті events.list).
ACL, Settings	Додаткові ресурси для правил доступу та налаштувань календаря (при потребі).

Висновок до другого розділу

У другому розділі бакалаврської роботи було розглянуто архітектурну організацію мобільного додатку для синхронізації особистого календаря з розкладом занять. Основна увага приділялася формуванню логічної структури застосунку, яка відповідає сучасним принципам розробки програмного забезпечення та забезпечує масштабованість, підтримуваність і тестованість системи.

Було обґрунтовано вибір архітектурного підходу, що базується на Clean Architecture з використанням патерну MVI. Така структура передбачає чітке розмежування відповідальностей між трьома основними шарами: presentation, domain та data. Це дозволяє мінімізувати зв’язність компонентів, забезпечити гнучкість у розширенні функціональності, а також спростити впровадження змін.

У межах розділу було побудовано та проаналізовано ключові UML-діаграми, що відображають логіку побудови системи. Діаграма розгортання ілюструє фізичну архітектуру застосунку: вона демонструє, як мобільний пристрій із локальними компонентами (інтерфейс, база даних) взаємодіє через мережу з хмарним сервісом Google Calendar API. Такий розподіл показує, що обробка подій відбувається на стороні клієнта, тоді як зовнішній API виконує роль віддаленого сховища календарних даних.

Діаграма класів відображає логічну модель основних сутностей системи. У центрі цієї моделі знаходиться доменна сутність «Подія» (Event), інтерфейс репозиторію CalendarRepository, а також класи-реалізації для доступу до локальних і віддалених джерел даних — GoogleCalendarService та LocalDatabase. Окремо представлено клас ViewModel, який реалізує бізнес-логіку керування станом інтерфейсу користувача. Діаграма демонструє залежності між цими класами відповідно до принципів інверсії залежностей, де доменна логіка не залежить від реалізацій нижчого рівня.

Компонентна діаграма подає високорівневу структуру системи як набір окремих модулів, кожен з яких має чітко окреслену відповідальність. У структурі

системи виокремлено модуль користувацького інтерфейсу, доменний модуль (use cases), модуль доступу до даних (репозиторій), а також компонент авторизації, що забезпечує керування доступом до Google API за допомогою OAuth 2.0. Взаємодія між компонентами здійснюється через визначені інтерфейси, що дозволяє легко підмінювати реалізації або розширювати систему без втручання в інші частини коду.

Особливу увагу приділено архітектурній моделі взаємодії з Google Calendar API. Цей процес повністю інкапсульовано в шарі даних. Вся логіка автентифікації користувача, формування HTTP-запитів, обробки відповідей API та обробки помилок реалізується окремим сервісом, що не впливає на логіку UI або доменну частину. Взаємодія з API здійснюється через інтерфейс CalendarRepository, який забезпечує абстракцію доступу до календаря — це дозволяє застосунку працювати з календарними подіями незалежно від того, чи вони зберігаються локально, чи надходять із хмарного джерела. Така архітектура відповідає принципам інверсії залежностей і сприяє безпечній, масштабованій та гнучкій роботі з зовнішнім API.

Таким чином, у другому розділі було сформовано концептуальну архітектуру мобільного застосунку, яка закладає надійну основу для подальшої реалізації функціональних підсистем. Застосування сучасних підходів до моделювання структури програмного забезпечення дозволяє забезпечити якість, підтримуваність і адаптивність системи. Це створює всі передумови для ефективної реалізації бізнес-логіки та інтеграції із зовнішніми сервісами, що буде детально розглянуто у наступному розділі.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ФУНКЦІОНАЛЬНОСТІ ЗАСТОСУНКУ

3.1 Мова програмування і платформа

У якості основної мови розробки для проєкту було обрано Kotlin — офіційно підтримувану Google мову для Android-застосунків. Kotlin є сучасною статично типізованою мовою, орієнтованою на зменшення шаблонного коду, підвищення безпеки та покращення читабельності. Однією з ключових особливостей Kotlin є вбудована система Nullable-типів, яка дозволяє уникнути типових помилок Java, зокрема NullPointerException, на етапі компіляції. Це позитивно впливає на якість програмного коду, особливо у великих проєктах, де контроль стабільності є критично важливим [46].

Окрім цього, Kotlin підтримує функціональне програмування, включаючи lambda-функції, операції з колекціями (наприклад, map, filter) та іммутабельність за замовчуванням, що дозволяє створювати більш передбачуваний і тестований код. У дослідженні зазначено, що застосування Kotlin у порівнянні з Java дозволяє на 40% скоротити кількість рядків коду при реалізації стандартних Android-компонентів, що безпосередньо впливає на продуктивність розробника та читабельність проекту [1].

Також Kotlin має повну сумісність з Java, що дозволяє використовувати існуючі бібліотеки без необхідності їх переписування. Це надало змогу безперешкодно інтегрувати такі сторонні компоненти як Retrofit, Firebase, Google Calendar API тощо. Зважаючи на активну підтримку спільноти та офіційні рекомендації Google, Kotlin став обґрунтованим вибором для розробки цього проєкту [16].

Для побудови інтерфейсу обрано Jetpack Compose [34] — декларативний UI-фреймворк, який поступово витісняє XML-підхід у сучасній Android-екосистемі. На відміну від традиційної верстки, де інтерфейс описується окремо від логіки (XML + Activity/Fragment), Compose дозволяє описувати UI-компоненти безпосередньо у вигляді функцій на Kotlin [16]. Це спрощує

підтримку, підвищує реактивність інтерфейсу, дозволяє легко реалізовувати динамічні сценарії без шаблонного коду (findViewById, View Binding тощо).

У таблиці 3.1 наведено порівняння підходів до побудови інтерфейсу користувача:

Таблиця 3.1 – Порівняння підходів до побудови UI в Android-застосунках

Критерій	XML	Jetpack Compose
Складність інтеграції зі станом UI	Висока	Низька (reactive model)
Кількість коду	Більше	Менше
Підтримка темізації	Обмежена	Підтримка Material Design 3
Навчальна крива	Невисока	Помірна
Гнучкість	Низька	Висока

Compose використовує парадигму унідирекційного потоку даних, що добре узгоджується з MVI-підходом, реалізованим у даному проєкті. UI автоматично оновлюється при зміні стану у ViewModel, що дозволило уникнути проблеми «розсинхронізації стану» між логікою та інтерфейсом.

Окремо варто згадати, що Jetpack Compose значно спрощує реалізацію адаптивного дизайну – тобто таких інтерфейсів, які коректно виглядають і функціонують на пристроях із різною діагоналлю екрана, щільністю пікселів або орієнтацією. Одним із ключових інструментів для цього є система Modifier, яка дозволяє налаштовувати відступи, розміри, вирівнювання, прокрутку, кліки та інші поведінкові й візуальні параметри елементів в єдиній декларативній манері. Наприклад, за допомогою `Modifier.fillMaxSize().padding(16.dp)` можна легко описати універсальний макет без залежності від розмітки XML.

Також варто відзначити використання BoxWithConstraints – спеціального контейнера, який дозволяє отримати доступ до доступних розмірів батьківського компонента та динамічно змінювати вигляд або логіку рендерингу залежно від ширини чи висоти. Це особливо корисно для підтримки планшетів або пристроїв із нестандартними екранами.

Для відображення списків використано LazyColumn, що є ефективним способом відображення великих обсягів повторюваних елементів із відкладеним рендерингом (on-demand rendering). У поєднанні з AnimatedVisibility це дало

змогу реалізувати плавне появлення або зникнення елементів, створюючи візуально приємну взаємодію користувача з контентом.

Крім того, інтерфейс підтримує компоненти Material Design 3, зокрема TabRow, Button, TextField, Surface, Card, що мають вбудовану адаптивність і враховують системні теми, кутові радіуси, анімації та кольорові схеми. Це дозволило досягти візуальної цілісності інтерфейсу та його легкої адаптації до світлої й темної теми. Приклад декларативного компонента Jetpack Compose: SelectorField (див. Додаток А.1)

Таким чином, поєднання Kotlin та Jetpack Compose дозволило створити швидкий, безпечний і сучасний Android-додаток, що повністю відповідає вимогам до масштабованості, стабільності та зручності використання.

3.2 Бібліотеки та фреймворки

Для реалізації проєкту були ретельно відібрані сучасні бібліотеки з екосистеми Android та Java, які відповідають вимогам функціональності застосунку. Обґрунтуємо вибір кожної з них та наведемо їх переваги:

- Jetpack Room – ORM-бібліотека для роботи з локальною SQL-базою даних. Room є частиною Android Jetpack і надає зручний абстрактний рівень над SQLite, забезпечуючи компіляторну перевірку SQL-запитів та зменшення boilerplate-коду при доступі до бази . На відміну від використання SQLiteOpenHelper, Room автоматично генерує необхідний код для створення/оновлення бази та впроваджує DAO (Data Access Objects) для типізованих запитів. Основні переваги Room: верифікація запитів під час компіляції (помилки в SQL будуть виявлені ще до запуску програми), зменшення шаблонного коду (розробник описує лише інтерфейси DAO та ентіті, без ручного написання коду керування курсорами), проста інтеграція з RxJava, LiveData та Kotlin Coroutines для реактивного отримання даних . У застосунку Room використовується для зберігання розкладу занять та пов’язаних даних (наприклад, ідентифікаторів подій календаря). Це дозволяє працювати з розкладом офлайн і забезпечує кешування – навіть без інтернет-з’єднання користувач може переглянути

свій розклад . Завдяки Room, реалізація локального сховища значно спростилась: визначено сутності (Entity) для занять, DAO для операцій (отримання розкладу, вставка/оновлення занять, пошук за датою тощо) та база даних з відповідною конфігурацією. Запити, сформульовані як методи DAO, гарантують типобезпеку та коректність, що знижує ризик runtime-помилок доступу до БД [49, 35].

- Hilt (Dagger Hilt) – фреймворк для dependency injection (впровадження залежностей) в Android-проєктах. Hilt побудований поверх популярної бібліотеки Dagger і автоматизує значну частину її налаштування, надаючи стандартизований та простіший API для впровадження залежностей у компоненти Android . Впровадження залежностей – ключовий механізм Clean Architecture, що дозволяє інвертувати залежності між модулями (UI, Domain, Data) та дотримуватися принципу односпрямованої залежності. Використання Hilt дає змогу легко надавати об’єкти (репозиторії, джерела даних, API-сервіси) туди, де вони потрібні, без жорсткого зв’язування класів. Основні переваги Hilt: зменшення boilerplate-коду в порівнянні з ручним DI або навіть Dagger (Hilt сам генерує необхідні компоненти для Android-класів – Application, Activity, ViewModel тощо), використання перевірок під час компіляції (Dagger генерує код DI, виявляючи помилки невідповідностей залежностей на ранньому етапі), висока продуктивність виконання (оскільки впровадження відбувається через згенерований код, що не створює накладних витрат в runtime). У цьому проєкті Hilt забезпечує впровадження таких залежностей, як репозиторій розкладу, сервіс для роботи з Google Calendar API, DAO бази даних, тощо у ViewModel та інші компоненти. Наприклад, @HiltViewModel анотація дозволяє Hilt автоматично створити необхідні об’єкти (репозиторії, DAO) і передати їх у конструктор ViewModel. Це полегшило дотримання Clean Architecture – верхні шари не створюють залежностей напряму, а отримують їх, не знаючи про конкретну реалізацію (наприклад, ViewModel оперує інтерфейсом репозиторію, реалізація якого надається Hilt-ом).

Таким чином вдалося досягти слабкого зв’язування модулів і високої тестованості (можна підміняти залежності на імплементації для тестів) [42, 21].

- Kotlin Coroutines – бібліотека для асинхронного програмування, яка є частиною мови Kotlin. Корутини дозволяють писати асинхронний код у послідовному стилі, спрощуючи роботу з потоками та зворотними викликами (callbacks). На Android, корутини є рекомендованим підходом для виконання довготривалих операцій без блокування основного потоку UI . В даному застосунку корутини широко використовуються для мережевих запитів (отримання HTML розкладу), роботи з диском (запис/читання з бази Room) та інтеграції з Google Calendar API. Основні переваги Kotlin Coroutines: легковагість (тисячі корутин можуть виконуватися на одному потоці, перемикаючись без суттєвих накладних витрат, на відміну від важких потоків) , підтримка механізму structured concurrency (структурованої конкурентності) – корутини мають контекст і життєвий цикл, прив’язаний до певного scope, що дозволяє автоматично скасовувати всі дочірні задачі при скасуванні батьківської, запобігаючи витокам ресурсів . Крім того, корутини мають вбудовану підтримку відкладення (suspending), що дозволяє писати послідовний код з асинхронними викликами (словом suspend), і механізми обробки виключень між потоками. У поєднанні з Flow (реактивний стрім від Kotlin), корутини забезпечують зручну реалізацію потоків даних – наприклад, потік змін розкладу з бази даних до UI. Завдяки корутинам, код, що раніше вимагав вкладених callback-ів або використання RxJava, став значно читабельнішим та простішим у підтримці. Наприклад, завантаження розкладу з сервера виконується в корутині viewModelScope у ViewModel, після чого результат відправляється в стан ViewState – UI автоматично оновлюється. У випадку довгих операцій (синхронізація з календарем), корутини дозволяють показати прогрес (через оновлення

стану loading=true) і не блокувати основний потік, що зберігає відгук інтерфейсу [27, 20].

- Firebase – для реалізації автентифікації користувача було вирішено використати Firebase Authentication. Firebase – це платформа від Google, що надає набір сервісів для мобільних застосунків, і зокрема модуль автентифікації дозволяє легко інтегрувати входження користувачів через популярні провайдери (Google, Facebook, email тощо) без необхідності розробляти власний бекенд для цієї мети . У нашому випадку головною вимогою було забезпечити авторизацію через обліковий запис Google користувача для подальшої синхронізації з його Google Calendar. Firebase спрощує цей процес: він забезпечує безпечний OAuth-вхід через Google та повертає інформацію про користувача (ID, ім’я, email) і токен, необхідний для подальших викликів Google API. Переваги використання Firebase Auth: висока безпека та надійність (усі складнощі OAuth2-процесу і зберігання токенів реалізовані на боці Firebase, розробник отримує вже аутентифікованого користувача), мінімум коду для інтеграції (через SDK Firebase потрібні кілька методів для ініціації входу та обробки результату), підтримка множинних провайдерів (в разі потреби можна легко додати інші методи входу) . У застосунку після вибору опції “Увійти через Google” запускається потік Firebase Auth: відкривається стандартний діалог обрання Google-акаунту, далі Firebase виконує аутентифікацію і повертає об’єкт FirebaseUser. При першому вході також виконується налаштування зв’язку з Google API – отримання OAuth2-токена з необхідними правами (доступ до Google Calendar). Таким чином, використання Firebase значно прискорило розробку модуля автентифікації, не потребувало власної реалізації протоколів безпеки і зменшило ризик помилок при роботі з обліковими записами користувачів [3].
- Retrofit – бібліотека для виконання HTTP-запитів. Retrofit від Square є високорівневим REST-клієнтом для Java і Android, який спрощує взаємодію з веб-сервісами та API. Ми використали Retrofit для отримання

HTML-сторінки розкладу занять з веб-сайту університету. Переваги Retrofit: він дозволяє описати REST API у вигляді інтерфейсу з анотаціями (GET, POST, тощо), та автоматично генерує реалізацію клієнта, що виконує HTTP-запити і перетворює відповіді у задані об’єкти (через конвертери JSON/XML) . Retrofit побудований поверх низькорівневої бібліотеки OkHttp і успадковує її продуктивність та можливості (кешування, повторні з’єднання, пул потоків) . У проєкті Retrofit спростив реалізацію клієнта до веб-розкладу: достатньо було вказати базовий URL сайту та створити інтерфейс на кшталт @GET("schedule") fun getScheduleHtml(): Response<ResponseBody>. Далі бібліотека сама здійснює мережевий виклик у фоні (в інтеграції з корутинами – повертає suspend функцію) і надає результат. Це суттєво скоротило обсяг коду порівняно з вручну налаштованим HttpUrlConnection або навіть OkHttp. Retrofit також полегшив обробку помилок мережі – у разі відсутності підключення чи помилки завантаження, ми отримуємо виняток/код відповіді, який можна обробити і видати користувачу повідомлення. Важливо, що Retrofit типобезпечний та підтримує додавання конвертерів: наприклад, якби розклад був в JSON, можна було б одразу парсити його в Kotlin-дані класи. У нашому випадку розклад був HTML-сторінкою, тому ми отримували ResponseBody і далі парсили його вручну (детальніше – в розділі 3). Проте, навіть для такого сценарію Retrofit виявився корисним, адже взяв на себе усі аспекти HTTP-з’єднання (URI, запити, редіректи, кеш) [23, 45].

- Google Calendar API – зовнішній веб-API, який надає доступ до Google Календаря користувача. Через цей API наш застосунок синхронізує події (заняття) з особистим календарем. API Google Calendar (версія v3) дозволяє програмно створювати, читати, змінювати та видаляти події у календарях користувача за його згодою. Використання цього API є ключовим для реалізації функціональності “синхронізації розкладу занять з персональним календарем”. Основна перевага – тісна інтеграція з

екосистемою Google: події, створені через API, відразу відображаються в календарі Google користувача на всіх пристроях. З технологічної точки зору, Google Calendar API – це RESTful API, що працює через протокол HTTPS. Для доступу застосунок повинен мати OAuth2-токен з відповідним scope доступу (https://www.googleapis.com/auth/calendar.events – право керувати подіями). Після отримання токена (через Firebase/Auth, як згадано вище), застосунок може виконувати запити до ендпойнтів API. Наприклад, для створення події використовується метод HTTP POST до ресурсу events: у запиті передається JSON-об'єкт події, що містить дату/час початку і закінчення, назву, опис тощо. У разі успіху API повертає створену подію з унікальним ідентифікатором. У нашому проєкті інтеграція з Calendar API реалізована через HTTP-виклики (задіяно той самий Retrofit/OkHttp клієнт, налаштований на базовий URL https://www.googleapis.com/calendar/v3/). Такий підхід дозволив безпосередньо викликати REST-методи API. Альтернативою було б використання бібліотеки-клієнта від Google для Java, але вона є досить важкою і не повністю оптимізованою для Android, тому вибір на користь прямого REST-виклику з Retrofit виправданий. В результаті, синхронізація подій відбувається так: застосунок відправляє запит events.insert до первинного календаря користувача (calendarId = 'primary') з параметрами події. Якщо запит успішний, отриманий eventId зберігається локально (щоб уникнути дублювання подій при повторній синхронізації). Таким чином, Google Calendar API надав можливість автоматизації додавання подій у календар користувача – ключову функцію даного застосунку [28, 14].

- Lottie – бібліотека від Airbnb для відтворення анімацій у форматі JSON, експортованих з Adobe After Effects. Хоч Lottie і не є критичною для архітектури, її було використано для покращення UX – зокрема, показу анімованих заставок і індикаторів завантаження (наприклад, під час синхронізації чи очікування). Lottie дозволяє розробникам легко вбудувати

складні векторні анімації без великого впливу на продуктивність, оскільки анімації рендеряться нативно на Android на основі заздалегідь підготовленого JSON-файлу. Переваги: дизайнери можуть підготувати анімацію в After Effects, використовуючи плагін Bodymovin експортувати її як JSON, і цей файл без змін підключається до Lottie-анімації в застосунку . Рендеринг виконується апаратно прискорено і виглядає плавно навіть на середніх пристроях. У нашому застосунку використані Lottie-анімації для заставки (логотип університету, що з'являється при запуску) та як прогрес-індикатори (наприклад, обертання іконки синхронізації). Це покращує сприйняття додатку користувачем, робить його більш сучасним. Наявність Lottie не вплинула на архітектуру кодово, проте її застосування виявило певні нюанси (затримка запуску анімації, вплив на потік UI) [19, 2].

Структура проєкту. Відповідно до обраної архітектури, проєкт структуровано за пакетами, що відображають шари Clean Architecture та компоненти MVI. Зокрема, виділено пакети: ui (містить composable-функції, Activity та ViewModel – тобто View і частково Presenter у термінах архітектури), domain (use-case, моделі предметної області), data (репозиторії, джерела даних, моделі даних для зберігання, DAO). Для інтеграції з зовнішніми API створено окремий пакет network або api, де розміщено сервіси Retrofit для університетського розкладу та Google Calendar. Таке розмежування спрощує навігацію по проєкту та підтримку: під час внесення змін зрозуміло, до якого шару належить функціонал. Компоненти DI Hilt (модулі, які позначено @Module та надають залежності) винесено у пакет di. Ресурси інтерфейсу (теми, стилі) – у відповідні каталоги ресурсів (хоча Compose здебільшого визначає теми в коді Kotlin, у нас підключено тему Material3).

На рисунку 3.1 показано процес налаштування доступу до Google Calendar API через консоль Google Cloud. Необхідно було створити OAuth 2.0 клієнт, прив'язаний до пакета нашого застосунку і SHA-1 сертифікату, щоб отримати Client ID, який використовувався для автентифікації запитів.

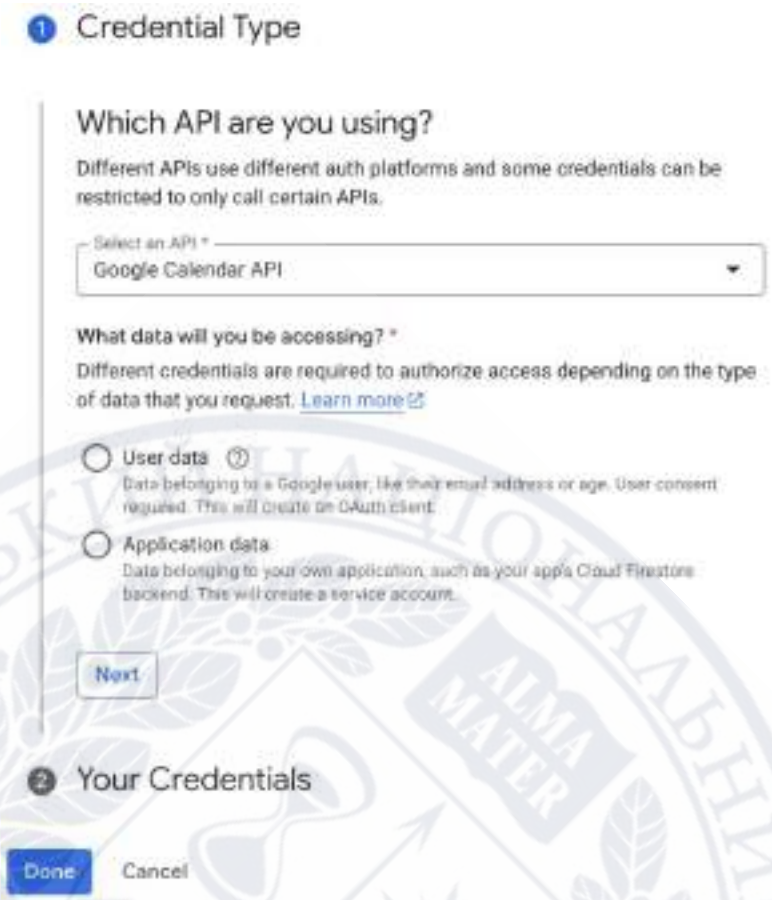


Рисунок 3.1 – Налаштування OAuth2-клієнта для доступу до Google Calendar API у консолі Google (вибір API Google Calendar та типу даних користувача)

На боці застосунку, після автентифікації через Firebase, отримуються облікові дані, які використовуються бібліотекою Google Sign-In для отримання OAuth-токена. В результаті, архітектура системи включає взаємодію таких компонентів: мобільний застосунок (Android) ↔ Firebase Auth (для входу) ↔ Google Calendar REST API (для операцій з подіями) ↔ база даних Room (для кешування та відслідковування синхронізованих подій). Кожна з цих складових була ретельно інтегрована, дотримуючись принципів слабкого зв’язування та інверсії залежностей: наприклад, виклики до Calendar API інкапсульовано в окремому класі-репозиторії CalendarRepository, що імплементує інтерфейс (це спрощує, зокрема, модульне тестування, де можна замінити справжній виклик API на мок).

На завершення, вибрані технології та архітектурні рішення відповідають сучасним рекомендаціям розробки Android-застосунків. В офіційних матеріалах

Google наголошується на використанні реактивної архітектури, односпрямованого потоку даних, а також комбінації ViewModel + Coroutines + DI для побудови масштабованих та тестованих застосунків — усе це було реалізовано у нашому проєкті.

Такий стек технологій:

- Kotlin
- Compose
- Clean Architecture
- MVI (Model-View-Intent)
- Room
- Hilt (Dependency Injection)
- Coroutines
- Firebase
- Retrofit
- Google API

Забезпечив баланс між надійністю, швидкодією та швидкістю розробки, дозволивши сконцентруватися на бізнес-логіці синхронізації календаря, а не на вирішенні другорядних технічних проблем.

3.3 Процес автентифікації та вибору навчальної групи

При першому запуску застосунку користувачеві пропонується пройти автентифікацію через обліковий запис Google (див. Додаток Б.2). Це необхідно, оскільки синхронізація розкладу занять здійснюється з персональним календарем Google користувача. Реалізація авторизації виконується через Firebase Authentication з провайдером Google Sign-In. Після натискання кнопки “Увійти через Google” відкривається стандартний інтерфейс вибору облікового запису Google. За лаштунками, Firebase SDK ініціює OAuth2-процес. В результаті успішного входу ми отримуємо FirebaseUser – об’єкт, що містить інформацію про користувача. Також Firebase забезпечує отримання OAuth-токена, дійсного для викликів Google API від імені цього користувача. Цей токен

зберігається безпечно Firebase-ом і може бути отриманий через метод FirebaseAuth та використаний у подальших запитах до Calendar API.

Однією з перших функцій, реалізованих у застосунку, є вхід користувача через Google-акаунт. Для цього було використано бібліотеку FirebaseUI, яка значно спрощує інтеграцію OAuth-авторизації.

Процес складається з двох етапів: ініціації входу та обробки результату. Нижче наведено приклад реалізації запуску інтенту авторизації:

```
val providers = arrayListOf(AuthUI.IdpConfig.GoogleBuilder().build())
AuthUI.getInstance()
    .createSignInIntentBuilder()
    .setAvailableProviders(providers)
    .build()
```

Приклад 3.1 – Ініціація входу через FirebaseUI

Отриманий інтент передається в startActivityForResult(), а результат обробляється у onActivityResult відповідно до стандартів Android:

```
override fun onActivityResult(requestCode: Int, resultCode: Int, data:
Intent?) {
    super.onActivityResult(requestCode, resultCode, data)
    if (requestCode == SIGN_IN_CODE) {
        val result = IdpResponse.fromResultIntent(data)
        if (resultCode == Activity.RESULT_OK) {
            val user = FirebaseAuth.getInstance().currentUser
            Log.d("Auth", "Logged in as ${user?.displayName}")
        } else {
            Log.e("Auth", "Sign-in failed: ${result?.error?.message}")
        }
    }
}
```

Приклад 3.2 – Обробка результату входу через Firebase

Після успішного входу користувач перенаправляється на екран вибору навчальної групи, реалізований з використанням Jetpack Compose (див. Додаток Б.2). Інтерфейс складається з трьох послідовних селекторів: спочатку обирається семестр (осінній або весняний), далі — факультет, після чого користувачу пропонується перелік курсів, доступних у вибраному факультеті, і лише після цього — список груп, які відповідають вибраним параметрам. Така послідовна

ієрархія дозволяє ефективно структурувати великий масив даних та забезпечити інтуїтивно зрозумілу навігацію.

У якості джерела даних про факультети та курси використовується HTML-розмітка з офіційного веб-сайту ВНТУ (https://jetiq.vntu.edu.ua). Парсинг виконується на стороні додатка за допомогою бібліотеки Jsoup, що дозволяє обробляти HTML як документ-дерево та легко отримувати доступ до потрібних елементів.

Факультети витягуються зі спеціального HTML-документа, який містить таблицю розкладу, де перший рядок таблиці містить комірки з назвами факультетів. Комірки мають клас dow_h, з яких за допомогою селекторів витягується текстовий вміст тегу <b> — це і є назва факультету.

Курси групуються за допомогою підструктури, де кожен факультет може мати кілька типів курсів: бакалаврат, магістратура, денна або заочна форма. Для кожного курсу ми створюємо окрему сутність типу Course, що містить назву, тип і форму навчання. Ці курси потім виводяться у вигляді сегментованого списку (BottomSheetSectionedList) з групуванням за типом та формою, що дозволяє зручно орієнтуватися в доступних варіантах.

З технічної точки зору обрані факультет, курс і група зберігаються у DataStore у вигляді ключ-значення (selected_faculty, selected_course, selected_group), що дозволяє при наступному запуску додатку миттєво перейти до основного функціоналу без потреби повторного вибору.

Таке рішення поєднує динамічність, гнучкість і зручність, дозволяючи не лише зберігати обрані параметри, але й у майбутньому реалізувати автоматичне оновлення списків із зовнішнього джерела. Фрагмент коду парсингу факультетів та курсів наведено у Додатку А.2 та Додатку А.3.

3.4 Парсинг розкладу занять з веб-сайту

Після того, як відомий факультет та курс, застосунок завантажує розклад з офіційного сайту університету. Розклад доступний на сайті у вигляді HTML-сторінки, що містить таблицю занять по днях тижня. На рисунку 3.2 наведено фрагмент такої HTML-таблиці розкладу: видно, що для кожної пари (номер по

вертикалі) і кожної групи (стовпці з кодами груп) вказано предмет, тип заняття, викладача, аудиторію, поділ на підгрупи та поділ за тижнями (позначені римськими I та II)

Рисунок 3.2 – Фрагмент сторінки розкладу занять (2 курс, 4 семестр), що обробляється застосунком

Завантаження HTML здійснюється за допомогою клієнта Retrofit (як описано в розділі 2.3). Після отримання HTML-коду виникає завдання парсингу – видобути з неструктурованого HTML впорядковані дані про заняття конкретної групи. Для цього застосовано бібліотеку JSoup (Java HTML Parser), яка дозволяє зручно обходити HTML-документ і вибирати елементи за CSS-селекторами або тегами. Спочатку HTML розклад завантажується у Document через Jsoup.parse(htmlString). Далі знаходиться таблиця розкладу – наприклад, через пошук за тегом <table> з певним класом або ідентифікатором. Усередині таблиці кожен рядок (<tr>) відповідає певній парі (номер заняття), а стовпці (<td>) – різним тижням. Необхідно визначити індекс стовпця, що відповідає обраній групі (наприклад, шляхом знаходження у заголовку таблиці <th> з назвою групи). Після цього для кожного рядка (кожної пари) ми беремо <td> відповідного стовпця – це вміст, що містить інформацію про заняття цієї групи у цей день і на цю пару (див. Додаток Б.3).

HTML-клітинка розкладу зазвичай містить текст з назвою предмету, типом заняття (лекція, практ. і т.д.), прізвищем викладача, аудиторією та підгрупою. У

випадку поділу на підгрупи, в комірці можуть бути два блоки тексту, розділені символом переводу рядка або
, відповідно для підгрупи 1 та 2. Наприклад:

Приклад 3.3 – Структура HTML-клітинки без підгрупи

```
<td>Фізика, Лекція<br>доц. Іваненко 101а</td>
```

У складніших випадках, коли розклад стосується лише однієї з підгруп або містить декілька блоків для підгруп I та II, HTML виглядає наступним чином:

Приклад 3.4 – Структура HTML-клітинки з підгрупами

```
<td>
     <b>Алгоритми, ЛР</b><br> <i><br>проф. Петренко</i><b>2022</b>
1п.<br>
     <b>Іноземна мова за професійним спрямуванням (англ), ПЗ</b>
<i><br>ст.в. А.М.Коваль</i><b>4220, 4224 4221</b> 2п.<br>
</td>
```

Тут видно, що для підгрупи 1 вказано заняття, а для 2 – інший тип заняття. Парсер повинен враховувати такі випадки. Використовуючи JSoup, можна розбити вміст <td> по тегам
 або отримати текст цілком і розділити за переносами рядка. У нашій реалізації було вирішено розділяти вміст на підгрупи за шаблоном "1п." і "2п." – якщо такі мітки є, розклад розпізнається як містить розподіл на підгрупи. Якщо міток немає, вважається, що обидві підгрупи мають однакове заняття (або група не ділиться).

Після видобування тексту заняття проводиться його розбір на структуровані поля: назва предмету, тип (лекція, практична, лабораторна), викладач, аудиторія. Це досягається пошуком відомих ключових слів і шаблонів. Наприклад, тип заняття можна розпізнати за певними скороченнями: “лек.”, “ЛК” – лекція, “прак.”/“ПЗ” – практична, “лаб.”/“ЛР” – лабораторна. У розкладі ВНТУ тип позначається абревіатурою (ЛК, ПЗ, ЛР, тощо). Застосунок містить відповідний словник для декодування цих скорочень у повні назви типів занять. Викладача, як правило, можна розпізнати по наявності ініціалів (наприклад, “доц. О.О. Сніговий”), аудиторію – як останнє число у рядку (напр. 101, 202-1 тощо). Цю логіку було акуратно закодовано через регулярні вирази.

Результатом парсингу кожної комірки є один або два об’єкти класу Lesson (в залежності від підгруп). Клас Lesson у доменній моделі містить поля: день тижня, номер пари, назва, тип, викладач, аудиторія, номер підгрупи (або null, якщо заняття спільне для всієї групи), а також тиждень (якщо розклад побудований по тижнях). У розкладі ВНТУ використовується двотижневий цикл (верхній/нижній тиждень, позначені римськими I/II зверху таблиці). Це також враховано: під час парсингу визначається, на який тиждень стосується запис. Якщо, наприклад, заняття стоїть тільки у стовпці “I” (верхній тиждень) – воно отримує мітку тижня 1, якщо тільки “II” – тиждень 2, якщо об’єднує обидва – тиждень = 0 (кожен тиждень). Внутрішньо ці позначки потрібні для коректної постановки подій у календарі (Google Calendar підтримує повторювані події з періодичністю раз на два тижні, тому заняття, що йдуть лише по парних тижнях, будуть відповідно марковані).

Побачити весь код реалізації парсера для розкладу можна побачити в Додатку А.4. Після повного обходу таблиці розкладу формується список об’єктів Lesson для всієї навчальної групи на весь тиждень. Цей список передається далі для збереження в локальну базу та для відображення у UI. Парсинг HTML – досить складна частина реалізації, оскільки формат розкладу може змінюватися, містити нестандартні записи (відміна занять, злиття груп тощо). В процесі розробки виникли виклики, зокрема:

- Обробка підгруп. Як згадано, потрібно було врахувати різні ситуації: обидві підгрупи мають одне заняття; у підгрупи 1 є заняття, у 2 – нема; навпаки; обидві мають різні заняття. Це вимагало гнучкого парсингу і рішення, як зберігати ці дані. В моделі було вирішено зберігати окремі об’єкти для кожної підгрупи (так легше потім синхронізувати з календарем – кожне заняття як окрема подія). Для випадку спільного заняття для всієї групи створюється один об’єкт Lesson без вказання підгрупи.
- Різні формати викладачів. Деякі записи містили звання (доцент, асистент), інші ні; також прізвища могли бути скорочені. Парсер був налаштований досить універсально – все, що не схоже на тип або аудиторію, вважалось

ім’ям викладача. Це працювало у більшості випадків, але вимагало ручного тестування на вибірці розкладів різних факультетів, щоб переконатися у відсутності збоїв.

- Неоднорідність розкладу. В розкладі могли траплятися порожні клітинки (якщо у групи взагалі немає пари в цей час), злиті клітинки (наприклад, одна пара на дві групи – рідко, але можливо у потокових лекціях). JSoup спрощує обробку порожніх комірок (вони просто дають порожній текст), але злиті комірки (rowspan/colspan) потребували додаткової логіки – якщо комірка охоплює два рядки, то це означає, що пара триває дві академічні години поспіль (такого не було у даному випадку), або якщо одна комірка на кілька груп (colspan), то це спільне заняття для цих груп (наприклад, потокова лекція). На щастя, у розкладі бакалаврату такі ситуації нечасті, і парсер їх розпізнавав лише як одне заняття для однієї групи, просто продубльоване у розкладі інших – тобто цей нюанс не став критичним.

Після парсингу застосунок показує користувачу розклад на екрані – в зручному форматі. У Compose це реалізовано як список елементів по днях тижня: наприклад, заголовок “Понеділок” і під ним картки занять по порядку часу. Кожна картка відображає назву предмета, час (номер пари перетворюється на реальний час початку і кінця, згідно університетського розкладу – наприклад, 1 пара 8:30–10:05), аудиторію, викладача. Якщо заняття тільки для однієї підгрупи – це позначається (текстом “(п. 1)”). При прокрутці таких списків Compose ефективно перерахує тільки видимі на екрані компоненти, що добре позначається на продуктивності навіть для довгого списку занять, що можна побачити в Додатку Б.4.

Реалізовано також можливість оновлення розкладу вручну – наприклад, через кнопку “Оновити” користувач може змусити додаток перезапитати HTML з сайту і перепарсити його. Це корисно, якщо університет вніс зміни в розклад. При натисканні “Оновити”, виконуються ті ж дії завантаження і парсингу, замінюючи старі дані в базі новими. На час завантаження UI показує індикатор прогресу (Lottie-анімацію обертання) замість списку занять. Важливо, що

операція виконується у фоновому потоці (корутині) – UI не блокується. Після завершення оновлення індикатор зникає, і оновлені дані відображаються. Один з викликів був пов’язаний з Lottie: виявилося, що перше відтворення анімації може починатися з затримкою ~0.5 секунди (ймовірно через завантаження JSON-анімації). Щоб UX не страждав, було прийнято рішення запускати анімацію завчасно, щоб вона закешувалася (див. Додаток А.5).

3.5 Збереження розкладу в локальній базі даних

Після успішного парсингу список Lesson зберігається у локальну базу даних Room. Це забезпечує кешування даних, дозволяючи відображати розклад миттєво при наступних запусках застосунку, навіть без інтернет-з’єднання. Крім того, локальне сховище дозволяє відстежувати синхронізацію з календарем, додаючи до кожного запису поле з ідентифікатором події в Google Calendar, якщо вона була експортована.

Основну роль у збереженні розкладу відіграє таблиця lessons, яка відображає сутність LessonEntity. Ця таблиця містить усі необхідні поля для зберігання інформації про заняття: унікальний ідентифікатор id, назву групи groupName, день тижня у вигляді тексту dayOfWeek та числовий індекс дня dayOfWeekIndex, номер пари pairNumber, часовий проміжок time, назву дисципліни subject, тип заняття lessonType (наприклад, лекція, лабораторна або практична), прізвище викладача teacher, номер аудиторії room, а також інформацію про тиждень (верхній або нижній) через поле weekParity і можливу підгрупу subgroup.

Ця структура описується у класі LessonEntity, який анотований як @Entity(tableName = "lessons"), що вказує Room створити відповідну таблицю в базі даних. У класі-поданні Lesson відображається та сама структура, але без анотацій — він використовується на рівні domain, тоді як LessonEntity — на рівні data.

У процесі синхронізації розкладу з Google Calendar кожному запису пари може бути створено відповідний Event, ідентифікатор якого зберігається в окремій таблиці lesson_events. Це дозволяє уникнути дублювання подій при

повторній синхронізації або видаленні занять з календаря. eventId – це унікальний рядок, який повертає Google Calendar API при створенні події (наприклад, "oq31abcde7rpgrsu7op0hn"). Ми зберігаємо його, щоб знати, які заняття вже синхронізовані з календарем: якщо в базі у запису є eventId, це означає, що подія вже створена в календарі. Це використовується при повторних синхронізаціях: щоб не створювати дублі, програма спочатку перевіряє, чи є eventId. Якщо ні – створює нову подію, якщо так – видаляє минулу подію та створює нову.

Для взаємодії з Room використано Kotlin Coroutines: DAO методи позначені як suspend або повертають Flow<List<...>>. Це дозволяє легко виконувати операції вводу/виводу асинхронно. Наприклад, отримання списку занять для відображення у UI виконується через lessonDao.getAllLessonsFlow().collect { ... }. При цьому Room сам відслідковує зміни в базі і надсилає оновлені дані у стрім (потік). Таким чином, коли користувач натискає “Оновити розклад”, після вставки нових даних у базу, UI автоматично отримує сигнал про зміну через Flow і перевідмальовує список занять. Це є реалізацією патерну Observable Data: база даних виступає джерелом даних, на яке підписаний UI, і дотримується принципу Single Source of Truth – єдиного джерела правди для стану інтерфейсу.

Збереження eventId в базі є важливою деталлю. eventId – це унікальний рядок, який повертає Google Calendar API при створенні події (наприклад, "oq31abcde7rpgrsu7op0hn"). Ми зберігаємо його, щоб знати, які заняття вже синхронізовані з календарем: якщо в базі у запису є eventId, це означає, що подія вже створена в календарі. Це використовується при повторних синхронізаціях: щоб не створювати дублі, програма спочатку перевіряє, чи є eventId. Якщо ні – створює нову подію, якщо так – видаляє минулу подію та створює нову.

3.6 Експорт розкладу в Google Calendar

Головна функція застосунку – синхронізація з Google Календарем – виконується на вимогу користувача (при натисканні кнопки “Синхронізувати

календар”). Ця дія ініціює процес створення календарних подій для всіх (або вибраних) занять у Google Calendar користувача.

Алгоритм синхронізації виглядає так:

1. Переконатися, що користувач автентифікований і має необхідні права. На цьому етапі, якщо з якоїсь причини токен доступу до Google Calendar втрачено або протерміновано, може знадобитися повторна авторизація. Як правило, FirebaseAuth зберігає сесію, і токен діє, тож цей крок прозорий.
2. Отримати зі сховища всі записи розкладу (LessonEntity) для поточного семестру.
3. Для кожного запису, що не має eventId (ще не експортований), сформувати запит до Google Calendar API для створення події.

Формування даних події враховує: дату і час. Оскільки розклад вказує дні тижня і номери пар, треба прив’язати їх до конкретних дат. Наприклад, якщо семестр розпочався 1 вересня 2024 року (понеділок верхнього тижня), то “Понеділок, пара 1, верхній тиждень” відповідає 01.09.2024 8:30. В застосунку дату першого понеділка (початку семестру) задано константою. На основі цього, для кожного заняття обчислюється дата першого його проведення. Якщо тиждень = I (тільки верхні тижні) – перша дата це тиждень №1 семестру, і подія буде повторюватися раз на 2 тижні; якщо тиждень = II (нижні) – перша дата це тиждень №2 семестру, повтор раз на 2 тижні. Для створення повторюваних подій Google Calendar API підтримує правила рекурентності (RRULE), але ми пішли шляхом створення індивідуальних подій на кожну дату повторення – для спрощення (потенційно створюється більше записів, але так простіше управляти ними індивідуально).

Виклик до Google Calendar API здійснюється методом events.insert до ресурсу календаря користувача. Оскільки автентифікація пройдена, ми маємо OAuth2 токен, який додається в Authorization-header кожного запиту. Запит містить JSON з полями події: summary (назва предмету), description (наприклад, тип заняття і викладач), location (аудиторія), start і end (дата-час початку і завершення пари, в ISO-форматі, з таймзоною). У відповіді, якщо успішно,

сервер повертає JSON нового event, зокрема поле id – це унікальний ідентифікатор події в календарі. Ми зберігаємо його в поле eventId відповідного LessonEntity та виконуємо lessonDao.updateEventId(id).

Однією з ключових функцій мобільного застосунку є автоматична синхронізація занять з Google Calendar користувача. Це дозволяє створити зручну, централізовану систему нагадувань про пари та забезпечити безперервну доступність розкладу на всіх пристроях, пов’язаних з обліковим записом Google.

Для реалізації інтеграції з Google Calendar було використано Google Calendar API. Доступ до нього забезпечується через OAuth 2.0 авторизацію на базі Firebase Authentication. Після входу в обліковий запис користувача, система отримує email, який використовується для ініціалізації клієнта Google Calendar:

```
val credential = GoogleAccountCredential.usingOAuth2(
    context,
    listOf(CalendarScopes.CALENDAR)
).apply {
    val account = Account(accountName, "com.google")
    selectedAccount = account
}

val calendar = Calendar.Builder(
    NetHttpTransport(),
    GsonFactory.getDefaultInstance(),
    credential
).setApplicationName("Calendary App").build()
```

Фрагмент 3.1 – Ініціалізація сервісу Google Calendar з обліковим записом користувача.

Після ініціалізації сервісу застосунок виконує парсинг збереженого розкладу з локальної бази даних (Room), де зберігається список занять для вибраної групи. Система автоматично визначає, які тижні є “верхніми”, а які “нижніми”, та додає лише ті заняття, які відповідають конкретному типу тижня. Для кожного заняття формується подія Google Calendar з точним часом початку та завершення, описом заняття, викладачем, підгрупою, аудиторією та іншими деталями.

```
val credential = GoogleAccountCredential.usingOAuth2(
    context,
    listOf(CalendarScopes.CALENDAR)
```

```
).apply {
    val account = Account(accountName, "com.google")
    selectedAccount = account
}

val calendar = Calendar.Builder(
    NetHttpTransport(),
    GsonFactory.getDefaultInstance(),
    credential
).setApplicationName("Calendary App").build()
```

Фрагмент 3.2 – Формування події Google Calendar з описом заняття.

Перед кожною новою синхронізацією календар очищується: усі події, створені раніше застосунком, видаляються за допомогою збережених eventId, що зберігаються локально в таблиці lesson_events. Це дозволяє уникнути дублювання подій та гарантує, що оновлення розкладу відображається коректно.

```
val events = lessonEventDao.getAllEventIds()
events.forEach { eventId ->
    try {
        calendar.events().delete(calendarId, eventId).execute()
    } catch (e: Exception) {
        // Подія, можливо, вже видалена
    }
}
lessonEventDao.clearAll()
```

Фрагмент 3.3 – Очищення Google Calendar перед новим експортом.

Варто зазначити, що Google Calendar API досить потужний: з його допомогою можна не лише створювати події, а й, наприклад, налаштовувати нагадування, прикріплювати файли, створювати власні календарі тощо. В нашій задачі це зайве – достатньо базових подій. API дозволяє також одним запитом додавати кілька подій (batch), але було вирішено не ускладнювати і надсилати послідовні запити в циклі для кожного заняття. При швидкості інтернету сучасних мобільних це займає не більше кількох секунд для ~30-40 подій.

Після завершення процесу користувач отримує повідомлення про успішну синхронізацію. Тепер, відкривши додаток Google Calendar, він побачить свій

розклад занять як звичайні події календаря у відповідні дні і час. Це значно зручніше, ніж дивитися статичний розклад – можна отримувати нагадування, бачити конфлікти з іншими подіями, спільно користуватися календарем. Побачити результат експорту розкладу можна в Додатку Б.6 та Додатку Б.7.

Одним із нюансів була різниця між часовими поясами та переходом на літній/зимовий час. Ми явно вказували timezone “Europe/Kyiv” при створенні подій, щоб час однозначно інтерпретувався. Також, оскільки дані розкладу фіксовані на семестр, події створюються лише в межах семестру (наприклад, з 1 вересня до 31 грудня) – щоб не продовжувати їх на невизначений термін. Це реалізовано шляхом перевірки дати: ми генеруємо події до останнього навчального тижня. Якщо додаток використовується довше (наступний семестр) – користувач може обрати нову групу/семестр і знову виконати експорт (див. Додаток Б.5).

3.7 Налаштування інтерфейсу (темна/світла тема)

Застосунок підтримує перемикання теми (світла або темна) для інтерфейсу. Це реалізовано засобами Jetpack Compose Theming. На екрані налаштувань (SettingsScreen) є перемикач (Switch) “Темна тема”. Його стан зберігається у SharedPreferences (наприклад, ключ "dark_theme_enabled"). При зміні стану перемикача UI викликає метод SettingsViewModel.toggleTheme(dark: Boolean), який зберігає значення і постачає його назад до вікна. Коренева композиція (AppTheme) читає це значення через observeAsState() і вибирає відповідну кольорову палітру. У Compose це виглядає приблизно так:

```
MaterialTheme(
      colorScheme = if(isDark) darkColorScheme() else lightColorScheme(),
      content = { ... }
)
```

Фрагмент 3.4 – Логіка зміни кольорової схеми застосунку.

Де darkColorScheme() і lightColorScheme() – це визначені нами набори кольорів для тем. Material3 надає стандартні палітри, які можна кастомізувати. Ми використовували базові кольори Material3 з невеликими змінами під

корпоративні кольори університету (див. Додаток Б.8). Перемикання теми відбувається динамічно – Compose автоматично перемалює всі елементи з новими кольорами. Також в налаштуваннях є опція “Системна тема” – коли ввімкнено, додаток підлаштовується під системні налаштування (наприклад, нічний режим Android). Це зроблено з використанням DynamicTheme API: якщо увімкнено, то замість фіксованих палітр застосунок бере кольори через DynamicColors (ті, що побудовані на основі шпалер користувача) – ця можливість Material You (Android 12+) також була випробувана. Проте, основний фокус – ручне перемикання теми користувачем.

Крім зміни теми, SettingsScreen дозволяє виконати вихід із акаунту Google (кнопка “Вийти”), яка викликає FirebaseAuth.getInstance().signOut() – після чого стан програми скидається, і при наступному відкритті знову покажеться екран входу.

3.8 Тестування застосунку

В ході розробки було написано кілька тестів – як модульних, так і інструментальних – для перевірки коректності роботи ключових компонентів.

Модульні тести ViewModel (див. Додаток А.7). Оскільки в архітектурі значна логіка зосереджена у ViewModel (особливо парсинг та підготовка даних), було вирішено написати для них unit-тести. Наприклад, для ScheduleViewModel (яка відповідає за завантаження/парсинг розкладу і управління станом UI) створено набір тестових випадків: перевірка, що з HTML-заготовки коректно парсяться Lesson-об’єкти; перевірка, що при помилці мережі ViewModel встановлює стан помилки (наприклад, поле errorMessage не пусте); перевірка роботи перемикання підгруп. Для ізоляції від реальних мережевих викликів, застосовано підміни (mock) RetrofitService і LessonDao. Використовуючи бібліотеки Mockito або MockK, ми створили фіктивні реалізації, що повертають зафіксований HTML або працюють із тимчасовою в пам’яті колекцією замість БД. Таким чином, тести зосередилися саме на логіці ViewModel. Один з тестів, наприклад, давав на вхід фрагмент HTML з відомим розкладом для однієї пари і очікував, що в стані scheduleState.lessons з’явиться відповідний Lesson з

правильними полями. Ці модульні тести допомогли виявити кілька помилок на ранніх етапах – зокрема, невірне розпізнавання підгруп у специфічних випадках. Після виправлення всі тести пройшли успішно.

Інструментальні UI-тести. Для Compose-інтерфейсу використовувалася бібліотека тестування Compose Testing. Було написано тест для екрану налаштувань (див. Додаток А.6**)** з наступним сценарієм: відобразити екран у тестовому контейнері, переконатися, що перемикач теми спочатку у правильному положенні (відповідає збереженим налаштуванням), натиснути на нього (симулювати кліком через Compose rule), перевірити, що стан ViewModel змінився**,** наприклад isDarkTheme стало true, і що текстове поле “Темна тема” відображається як увімкнене. Такі тести використовують createComposeRule() та методи onNodeWithText, onNodeWithTag для пошуку елементів, та performClick для симуляції дій. Перевагою Compose-тестів є те, що вони дуже чітко імітують взаємодію користувача з UI, дозволяючи перевірити, що інтерфейс реагує правильно. Інший тестовий сценарій перевіряв компонент відображення заняття (LessonItem composable): ми передавали йому об’єкт Lesson з певними полями і перевіряли через assertTextContains на дочірніх нодах, що всі дані (назва, аудиторія, викладач) показано.

Окрім автоматизованих тестів, проводилося також ручне тестування на різних пристроях і екранах. Переконалися, що адаптивність інтерфейсу працює: на маленьких екранах список розкладу прокручується, на таблете компоненти розтягуються, темна тема оформлюється коректно (тексти читабельні на темному фоні). Перевірено сценарії зміни налаштувань телефону (мова, часовий пояс) – застосунок адекватно реагує (часи пар коригуються під часовий пояс). При повній відсутності інтернету – повідомлення про неможливість оновити розклад, але останній кешований варіант все ще доступний.

Таким чином, у ході розробки було забезпечено багаторівневе тестування, що охопило як бізнес-логіку (парсинг, синхронізація), так і користувацький інтерфейс. Це відповідає принципам побудови якісного програмного забезпечення, де Clean Architecture сприяє тестуванню – оскільки більшість

логіки ізольовано від платформозалежних деталей, її легко перевіряти автономно.

Висновок до третього розділу

У цьому розділі було безпосередньо реалізовано функціональність мобільного застосунку, спрямованого на синхронізацію розкладу занять з персональним календарем користувача. Результати роботи підтвердили доцільність обраної архітектури, технологічного стеку та продемонстрували життєздатність запропонованої концепції.

Першим важливим етапом стала реалізація процесу автентифікації користувача через Google-акаунт за допомогою Firebase Authentication. Це рішення дозволило забезпечити безпечний вхід та автоматизовано отримувати доступ до Google Calendar API без збереження чутливих даних на боці клієнта. Саме через отриманий OAuth-токен додаток надалі здійснює авторизовану взаємодію з календарем користувача. Реалізація проходить у два етапи: ініціація входу за допомогою FirebaseUI та обробка результату авторизації у відповідній activity. Такий підхід значно пришвидшив інтеграцію й дозволив уникнути складної ручної реалізації OAuth2.

Після входу користувач проходить початкове налаштування: вибір семестру, факультету, курсу та групи. Цей етап реалізовано через зручний інтерфейс Jetpack Compose із кастомними селекторами та використанням модальних вікон (BottomSheet), що забезпечує інтуїтивну навігацію навіть при великій кількості доступних значень. Дані факультетів, курсів і груп парсяться з офіційного сайту університету з використанням бібліотеки Jsoup. Особливістю реалізації стало оброблення HTML-структур зі складними комбінаціями підгруп, кількох аудиторій і викладачів. Було реалізовано гнучкий парсер, який витягує всі необхідні дані з таблиць, враховуючи розмітку, реляційну структуру та контекст кожного елемента.

Особливо складними виявилися випадки, коли HTML-клітинка містила по декілька блоків для підгруп, з різними викладачами, типами занять та аудиторіями. У цих ситуаціях парсер виконує аналіз регулярних виразів і формує уніфіковану структуру, яка зберігається у локальну базу даних через Room. Ці дані включають день тижня, номер пари, назву предмета, тип заняття, викладача,

аудиторію, підгрупу та парність тижня. Такий рівень деталізації дозволяє точно відтворити структуру реального розкладу та відобразити її в UI.

Окрему увагу приділено збереженню вибору користувача та кешуванню даних. Використання DataStore для зберігання параметрів налаштування дозволяє запускати додаток без необхідності повторного вибору факультету чи групи, що значно покращує UX. Для зберігання самих розкладів ідентифікованих подій використовується Room, що забезпечує швидкий доступ до даних та можливість роботи в офлайн-режимі.

Синхронізація з Google Calendar реалізована через прямі REST-запити до Google Calendar API v3. Під час імпорту дані конвертуються у формат подій, який включає назву, опис, дату, час початку/закінчення та унікальний eventId. Для уникнення дублювання кожна подія зберігається з ідентифікатором у локальну базу. Додатково реалізовано механізм повторного імпорту (оновлення) та видалення синхронізованих подій. Усі ці операції працюють на базі Kotlin Coroutines з відображенням стану завантаження через прогрес-індикатори.

Інтерфейс програми був вдосконалений для забезпечення сучасного вигляду відповідно до Material Design 3. Було створено кастомні селектори, перероблено вигляд списків та форм, додано підтримку темної/світлої теми, інтегровано Lottie-анімації для покращення сприйняття застосунку. Особлива увага приділялась адаптивності: усі компоненти побудовані з використанням Modifier API для коректної роботи на різних діагоналях.

Також реалізовано блок налаштувань, який дозволяє змінити тему, групу, повторно імпортувати розклад, синхронізувати або видалити події з Google Calendar, а також вийти з акаунту. Усі ці дії тестуються через unit та UI тести, зокрема написані тести ViewModel та інтеграційні тести з Compose Testing API.

Підсумовуючи, реалізація всіх основних функцій мобільного застосунку, описаних у цьому розділі, повністю відповідає початковим цілям проєкту. Було успішно реалізовано повний цикл: від авторизації користувача до завантаження, збереження, парсингу та синхронізації розкладу з Google Calendar, а також забезпечено зручний та естетичний інтерфейс для взаємодії з додатком.

Отриманий результат не лише демонструє працездатність технічного рішення, а й має потенціал до масштабування та подальшого розвитку, зокрема інтеграції з іншими календарними сервісами, підтримки декількох акаунтів чи джерел розкладу.

ВИСНОВКИ

У межах виконання бакалаврської кваліфікаційної роботи на тему «Розробка мобільного додатку для синхронізації особистого календаря та розкладу занять за допомогою Web-API» було всебічно досліджено предметну область, проаналізовано сучасні технічні рішення в галузі цифрового управління розкладом, визначено науково-технічну проблему та успішно реалізовано її вирішення у вигляді повнофункціонального Android-застосунку.

У процесі дослідження було розглянуто існуючі підходи до організації розкладу занять у закладах вищої освіти, охарактеризовано методи інтеграції з календарними сервісами, зокрема Google Calendar, а також оцінено наявні програмні засоби, що реалізують часткову синхронізацію чи ручне управління подіями. Результати аналізу підтвердили актуальність проблеми: відсутність централізованих рішень, складність користування існуючими інструментами для більшості студентів, дублювання дій, ризики помилок при перенесенні розкладу вручну.

З огляду на поставлену мету було розроблено архітектуру застосунку на базі Clean Architecture, яка забезпечує чітке розділення відповідальностей між рівнями: дані, домен та презентація. Як патерн управління інтерфейсом обрано MVI, що дозволяє реалізувати реактивне управління станом та ефективно обробляти події користувача. У якості основних технологій використано Kotlin, Jetpack Compose для побудови інтерфейсу, Room для локального зберігання, Retrofit для роботи з мережею, Firebase Authentication для автентифікації користувача, а також Google Calendar API для синхронізації подій.

У межах реалізації було створено застосунок, який дозволяє студенту авторизуватись через Google, обрати групу, отримати актуальний розклад занять шляхом парсингу HTML-таблиці з сайту університету, зберегти його у локальну базу даних, і автоматично перенести пари до особистого календаря з урахуванням парності тижнів, підгруп, викладачів, часу та місця проведення. Також реалізовано логіку збереження ідентифікаторів подій для уникнення дублювання під час повторної синхронізації. Інтерфейс адаптований під темну і

світлу тему, забезпечено плавну взаємодію з користувачем, підтримку офлайн-доступу та мінімальну кількість кроків для повноцінного використання.

Тестування показало високу стабільність роботи, коректну обробку типових сценаріїв та виняткових ситуацій. Отримані результати демонструють практичну цінність створеного рішення, що дозволяє студентам уникнути ручної роботи з розкладом, бути завжди в курсі змін у навчальному графіку, а також ефективніше організовувати власний час. Запропонований підхід може бути адаптований для інших вищих навчальних закладів України, а також масштабований на інші платформи, зокрема iOS.

Таким чином, поставлена мета бакалаврської роботи досягнута в повному обсязі. Розв’язано всі поставлені завдання: проведено аналіз предметної області, обґрунтовано доцільність створення додатку, розроблено архітектуру, реалізовано функціонал і перевірено його працездатність. Отримані результати свідчать про високий рівень теоретичної та практичної підготовки автора до професійної діяльності в галузі розробки програмного забезпечення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Acar M., Fogelstrom D. A Practical Study of Kotlin and Java Performance in Android Apps // Journal of Software Engineering and Applications. – 2022.
2. Airbnb Engineering. Lottie: Render After Effects animations natively on Android and iOS. URL: https://airbnb.io/lottie (дата звернення: 14.05.2025).
3. Almalki H., Ghubaish R. Evaluation of Firebase Authentication in Android Application Development // Int. J. of Computer Science and Network Security. – 2021. – Vol. 21(10). – C. 50–55.
4. Atwood C. Unidirectional Data Flow in Mobile Architecture. – IEEE Software, 2021.
5. Bellamy J. Architectural Patterns in Modern Android Applications. – ACM, 2020.
6. Bridgeman A., Arndell M., Goldsworthy R., Taylor C., Tzioumis V. An Electronic Calendar for Organizing Assessments in a Large Faculty // EDUCAUSE Review. – 2013. – 48(2). – C. 64–65.
7. Britton B.K., Tesser A. Effects of time-management practices on college grades // Journal of Educational Psychology. – 1991. – 83(3). – C. 405–410.
8. Budnikov R. Using Service Objects in Go. URL: https://itnext.io/using-service-objects-in-go-d899dc599335 (дата звернення: 14.05.2025).
9. BYU OIT. Assignment Updates on Calendar (BYU Learning Suite). URL: https://softwaresupport.byu.edu/learning-suite/student/schedule/assignment-updates-on-calendar (дата звернення: 10.05.2025).
10. California State University San Marcos. Exporting Cougar Courses Calendar. URL: https://www.csusm.edu/iits/services/ats/idesign/moodle/guides/students-basics-exporting-cougar-courses-calendar/index.html (дата звернення: 20.05.2025).
11. Doodle. Time Management in Education – Research Study. URL: https://doodle.com/en/resources/research-and-reports/time-management-in-education/ (дата звернення: 11.05.2025).

12. Dutkiewicz S. The Importance of Clean Architecture. URL: https://blog.nimblepros.com/blogs/the-importance-of-clean-architecture/ (дата звернення: 14.05.2025).
13. Fielding R.T. Architectural Styles and the Design of Network-based Software Architectures. – Doctoral dissertation, University of California, Irvine, 2000. – 162 с.
14. Google Developers. Google Calendar API v3 Documentation. URL: https://developers.google.com/calendar (дата звернення: 14.05.2025).
15. Google Developers. Google Calendar API Overview. URL: https://developers.google.com/calendar/api/guides/overview (дата звернення: 11.05.2025).
16. Hazzard D., Ignatov A. Kotlin in Action. – Manning Publications, 2017.
17. Herumurti D., Bimantara I.M.S., Supriana I.W. User-Centered Design-Based Approach in Scheduling Management Application Design and Development // IPTEK Journal of Technology and Science. – 2023. – 34(1). – C. 26–43.
18. iCalendar.org. Internet Calendaring and Scheduling Core Object Specification (RFC 5545). URL: https://icalendar.org (дата звернення: 20.05.2025).
19. Jain P., Sinha R. Enhancing Mobile UI/UX using Lottie Animations in Android Apps // Journal of Human-Computer Interaction & Design. – 2022. – Vol. 3(2). – C. 90–96.
20. JetBrains. Kotlin Coroutines: Design and Implementation. – KotlinConf, 2020.
21. Jivani P., Patel D. Implementation of Dependency Injection Using Hilt in Android Applications // Journal of Software Engineering & Intelligent Systems. – 2022. – Vol. 4(3). – C. 75–80.
22. Kat Z. How I Write HTTP Services in Go after 13 Years // Grafana Labs. URL: https://grafana.com/blog/2024/02/09/how-i-write-http-services-in-go-after-13-years/ (дата звернення: 22.05.2025).
23. Khan S. Effective RESTful API Consumption in Android using Retrofit and OkHttp // Journal of Mobile Computing and Development. – 2020. – Vol. 6(1). – C. 12–19.

24. Khvostov A., Skrypnikov V., Valentinovich L. et al. Security threats to personal data in the implementation of distance educational services using mobile technologies // Journal of Theoretical and Applied Information Technology. – 2021. – 99(15). – С. 3543–3555.
25. Kizin G.O., Lishchynska L.B. Інтеграція календаря подій у веб-платформу для організації конференцій // Матеріали LIV НТКП ВНТУ. – 2025. – С. 52–54.
26. Krumrei-Mancuso E.J., Newton F.B., Kim E., Wilcox D. Psychological factors predicting university students' academic success: Do self-efficacy and self-regulated learning matter? // College Student Journal. – 2013. – 47(3). – С. 524–535.
27. Kusmin A., Grechuk M. Asynchronous Programming in Kotlin Using Coroutines // Int. J. of Computer Applications. – 2021. – Vol. 183(23). – С. 18–23.
28. Liang H., Wang Y. Integration of Google Calendar API in Smart Scheduling Systems // Int. Conf. on Information Science and Systems. – IEEE, 2021.
29. Macan T.H., Shahani C., Dipboye R.L., Phillips A.P. College students' time management: Correlates with academic performance and stress // Journal of Educational Psychology. – 1990. – 82(4). – С. 760–768.
30. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. – Pearson Education, 2017.
31. Mei J. Learning Management System Calendar Reminders and Effects on Time Management and Academic Performance // International Research and Review. – 2016. – 6(1). – С. 30–48.
32. Minyazev R.R. Разработка инструмента для интеграции Timetable в приложения-календари: Отчёт по учебной практике. – СПбГУ, 2023. – 18 с.
33. Misra R., McKean M. College students' academic stress and its relation to their anxiety, time management, and leisure satisfaction // American Journal of Health Studies. – 2000. – 16(1). – С. 41–51.

34. Muller J., Rabe M. Declarative UI in Practice: Adoption of Jetpack Compose in Android Development // Software Engineering 2021. – Lecture Notes in Informatics, 2021.
35. Murali P. Android Jetpack: Room Database – Complete Practical Guide // Int. J. of Advanced Trends in Computer Science and Engineering. – 2022. – Vol. 11(2). – C. 290–295.
36. MyStudyLife. Free Student Planner for Web, Android & iOS. URL: https://www.mystudylife.com (дата звернення: 10.05.2025).
37. Oliveira M. Android Clean Architecture — A “Unicorn” Approach. URL: https://engineering.talkdesk.com/android-clean-architecture-an-unicorn-approach-a5076d1b409 (дата звернення: 14.05.2025).
38. Pambudi A.P., Waluyo A., Fatich E.V.L.N. Perancangan Sistem Penjadwalan Perkuliahan Berbasis Website Menggunakan Algoritma Genetika // Jurnal Teknik Informatika dan Sistem Informasi. – 2021. – 8(3). – C. 1133–1146.
39. Pautasso C., Zimmermann O., Leymann F. Restful web services vs. “big” web services: Making the right architectural decision // Proceedings of the 17th International Conference on World Wide Web (WWW2008). – 2008. – C. 805–814.
40. Pintrich P.R., Smith D.A., García T., McKeachie W.J. Reliability and predictive validity of the Motivated Strategies for Learning Questionnaire (MSLQ) // Educational and Psychological Measurement. – 1993. – 53(3). – C. 801–813.
41. Purnomo F.A., Isha N.F., Sahara S. Development of MPLMSC Schedule App to Facilitate Students in Accessing Course Schedules // International Journal of Software Engineering and Computer Science. – 2024. – 4(2). – C. 586–594.
42. Rahman R. Hilt: A New Way to Do Dependency Injection in Android. – Android Dev Summit, Google Developers, 2021.
43. RFC 5545: Internet Calendaring and Scheduling Core Object Specification (iCalendar). – IETF, 2011. – 128 с.
44. RFC 6749: The OAuth 2.0 Authorization Framework. – IETF, 2012. – 76 с.

45. Tan J., Lim S. Network Efficiency and Error Handling Using Retrofit in Android Apps // Mobile Applications and Services Journal. – 2021. – Vol. 7(4). – C. 44–53.
46. Tastan B., Top E. A Comparison of Java and Kotlin for Android Mobile Application Development // International Journal of Computer Trends and Technology. – 2020. – Vol. 68(4). – C. 26–30.
47. University of Warwick. Import and export iCalendar files. URL: https://warwick.ac.uk/services/its/servicessupport/web/sitebuilder2/manual/pages/calendar/icalendar/ (дата звернення: 20.05.2025).
48. Yang J., Huang Y., Morwitz V. How using a paper versus mobile calendar influences everyday planning and plan fulfillment // Journal of Consumer Psychology. – 2023. – 33(1). – C. 153–170.
49. Yao Y., Wang X. An Empirical Study of Android Room Persistence Library // Proc. 18th Int. Conf. on Mobile Systems, Applications, and Services. – ACM, 2020.
50. Zimmerman B.J., Greenberg D., Weinstein C.E. Self-regulating academic learning: the role of individual differences // Learning and Individual Differences. – 1994. – 6(2). – C. 205–225.

ДОДАТОК А

Компоненти використані у додатку

Лістинг А.1 SelectorField компонент

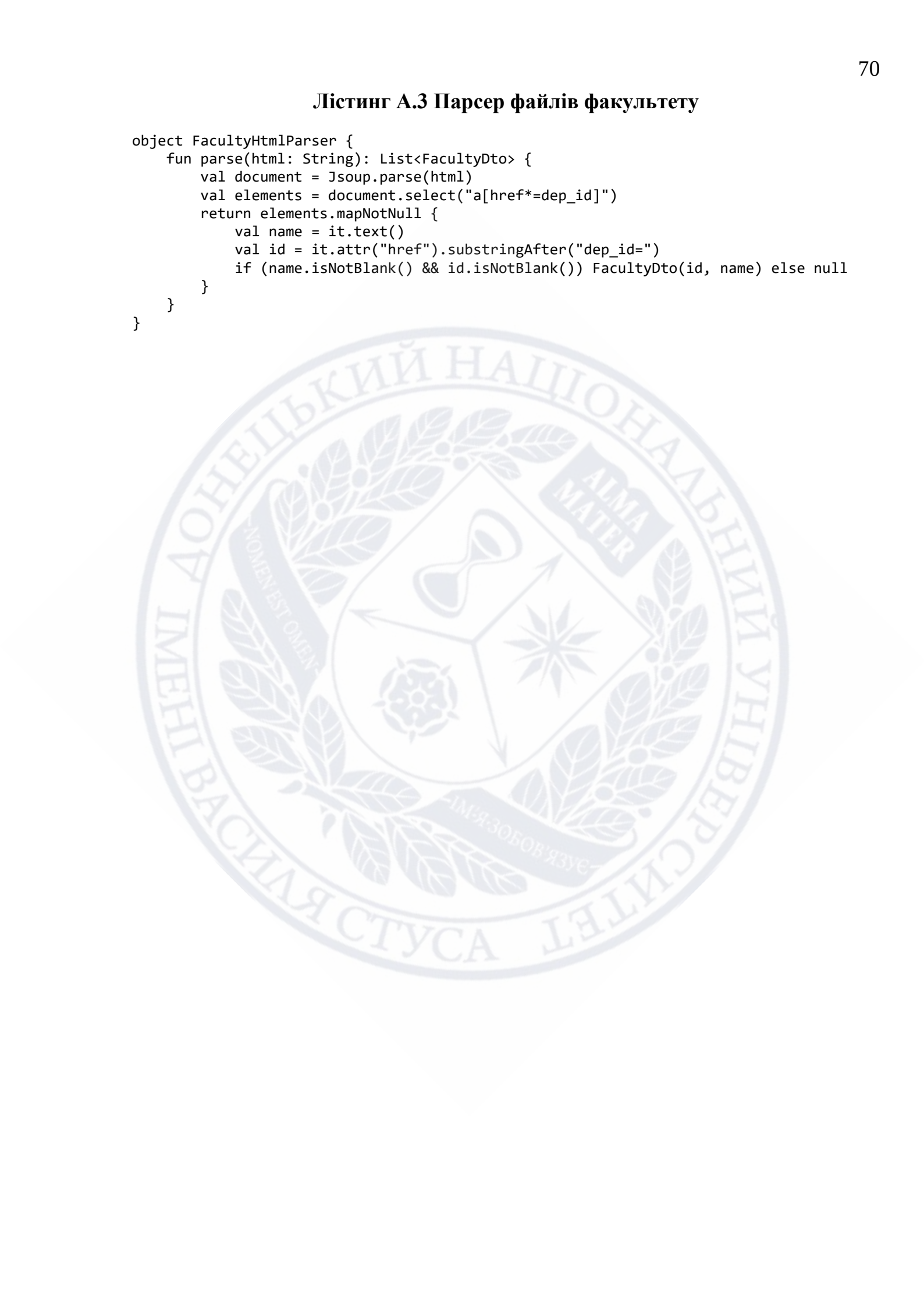
```
@Composable
fun SelectorField(
    value: String,
    onClick: () -> Unit,
    isEnabled: Boolean = true
) {
    val colors = MaterialTheme.colorScheme
    Surface(
        onClick = onClick.takeIf { isEnabled } ?: {},
        enabled = isEnabled,
        shape = RoundedCornerShape(12.dp),
        border = BorderStroke(1.dp, colors.outline),
        tonalElevation = if (isEnabled) 1.dp else 0.dp,
        modifier = Modifier
            .fillMaxWidth()
            .height(60.dp)
    ) {
        Row(
            modifier = Modifier
                .fillMaxSize()
                .padding(horizontal = 20.dp),
            verticalAlignment = Alignment.CenterVertically
        ) {
            Text(
                text = value,
                style = MaterialTheme.typography.bodyLarge,
                color = if (isEnabled) colors.onSurface else colors.onSurfaceVariant,
                maxLines = 1,
                modifier = Modifier.weight(1f)
            )
            Icon(
                imageVector = Icons.Default.ArrowDropDown,
                contentDescription = "Розгорнути",
                tint = if (isEnabled) colors.primary else colors.onSurfaceVariant
            )
        }
    }
}
```

Лістинг А.2 Парсер файлів курсу

```
object CourseHtmlParser {
    fun parse(html: String): List<Course> {
        val document = Jsoup.parse(html)
        val rows = document.select("table tbody tr")
        val courses = mutableListOf<Course>()
        var currentType: ScheduleType? = null
        var i = 0
        while (i < rows.size) {
            val row = rows[i]
            val strongText = row.selectFirst("strong")?.text()?.trim()
            when {
                strongText?.contains("Розклад занять") == true -> {
                    currentType = ScheduleType.REGULAR
                    i++
                    continue
                }
                strongText?.contains("Розклад екзаменаційної сесії") == true -> {
                    currentType = ScheduleType.EXAM
                    i++
                    continue
                }
            }
            if (currentType != null) {
                val columns = row.select("td")
                if (columns.size >= 1) {
                    val fullTimeColumn = columns.getOrNull(0)
                    val partTimeColumn = columns.getOrNull(1)
                    courses += extractCoursesFromColumn(
                        fullTimeColumn,
                        currentType,
                        StudyForm.FULL_TIME
                    )
                    courses += extractCoursesFromColumn(
                        partTimeColumn,
                        currentType,
                        StudyForm.PART_TIME
                    )
                }
            }
            i++
        }
        return courses
    }
    private fun extractCoursesFromColumn(
        column: Element?,
        type: ScheduleType,
        form: StudyForm
    ): List<Course> {
        if (column == null) return emptyList()
        return column.select("li a").map { link ->
            val name = link.text().trim()
            val url = link.attributes()["href"]
            Course(name, url, type, form)
        }
    }
}
```

Лістинг А.3 Парсер файлів факультету

```
object FacultyHtmlParser {
    fun parse(html: String): List<FacultyDto> {
        val document = Jsoup.parse(html)
        val elements = document.select("a[href*=dep_id]")
        return elements.mapNotNull {
            val name = it.text()
            val id = it.attr("href").substringAfter("dep_id=")
            if (name.isNotBlank() && id.isNotBlank()) FacultyDto(id, name) else null
        }
    }
}
```



Лістинг А.4 Реалізація HTML парсера для розкладу

```
fun parseSchedule(htmlContent: String): List<Lesson> {
    val doc = Jsoup.parse(htmlContent)
    val scheduleTable = doc.selectFirst("table.c_body") ?: return emptyList()
    val groupNames = extractGroupNames(scheduleTable)
    val lessons = mutableListOf<Lesson>()
    var currentDay = ""
    scheduleTable.select("tr").drop(2).forEach { row ->
        row.selectFirst("td.dowh")?.text()?.trim()?.let { currentDay = it }
        val cells = row.select("td.c_lsn")
        cells.forEachIndexed { index, cell ->
            val (groupInfo, weekParity) = groupNames.getOrNull(index) ?:
return@forEachIndexed
            val (groupName, _) = groupInfo
            parseLessonsFromCell(
                cell,
                groupName,
                currentDay,
                WeekParity.fromLabel(weekParity),
                lessons
            )
        }
    }
    return lessons
}
private fun extractGroupNames(scheduleTable: Element): List<Pair<Pair<String, String>,
String>> {
    val headerCells =
        scheduleTable.select("tr").first()?.select("td.dow_h") ?: return emptyList()
    return headerCells.flatMap { header ->
        val group = header.selectFirst("b")?.text()?.trim() ?: ""
        listOf(Pair(group, "|") to "нижній", Pair(group, "||") to "верхній")
    }
}
private fun parseLessonsFromCell(
    cell: Element,
    groupName: String,
    currentDay: String,
    weekParity: WeekParity,
    lessons: MutableList<Lesson>
) {
    cell.select("table.c_cell tr").forEach { lessonRow ->
                    val numberCell = lessonRow.selectFirst("td.dow big
                        b")?.text()?.trim()?.toIntOrNull()
        val lessonCell = lessonRow.selectFirst("td.lsn, td.lsn_dbl") ?: return@forEach
        lessonCell.select("div.l_width").forEach { lessonInfo ->
            val html = lessonInfo.html()
            val subgroupMatches = Regex(

"""<b>(.+?)\s*(?:,)?\s*(ПЗ|ЛР|ЛК|ЛЕК)</b><i><br>(.+?)</i>\s*<b>\s*([\d,\s]+)</b>\s*(\dп
\.?)"""
            ).findAll(html).toList()
            if (subgroupMatches.isNotEmpty()) {
                val allSubjectsSame =
                    subgroupMatches.all { it.groupValues[1].trim() ==
subgroupMatches.first().groupValues[1].trim() }
                val allTypesSame =
```

```
                    subgroupMatches.all { it.groupValues[2].trim() ==
subgroupMatches.first().groupValues[2].trim() }
                val subject = if (allSubjectsSame) {
                    subgroupMatches.first().groupValues[1].trim()
                } else {
                    buildString {
                        subgroupMatches.forEachIndexed { index, match ->
                            val subSubject = match.groupValues[1].trim()
                            appendLine("${index + 1} підгрупа — $subSubject")
                        }
                    }.trim()
                }
                val type = if (allTypesSame) {
                    mapLessonType(subgroupMatches.first().groupValues[2].trim())
                } else {
                    LessonType.UNKNOWN
                }
                val subgroupInfo = buildString {
                    subgroupMatches.forEach { match ->
                        val lessonType =
mapLessonType(match.groupValues[2].trim()).label
                        val teacher = Jsoup.parse(match.groupValues[3]).text().trim()
                        val rooms = match.groupValues[4].trim().replace(",", ", ")
                            .replace(Regex("\\s+"), " ")
                        val parity = match.groupValues[5].trim()
                        appendLine("$parity: $lessonType, \uD83D\uDC64 $teacher,
\uD83C\uDFEB ауд. $rooms")
                    }
                }
                lessons.add(
                    Lesson(
                        groupName = groupName,
                        dayOfWeek = currentDay,
                        pairNumber = numberCell ?: 0,
                        time = pairTimes[numberCell] ?: "",
                        subject = subject,
                        lessonType = type,
                        teacher = "",
                        room = "",
                        weekParity = weekParity,
                        subgroup = subgroupInfo.trim(),
                        dayOfWeekIndex = dayIndex[currentDay] ?: 0
                    )
                )
            } else if (numberCell != null && pairTimes.containsKey(numberCell)) {
                lessons.add(
                    parseNonSubgroupLesson(
                        lessonInfo,
                        groupName,
                        currentDay,
                        numberCell,
                        weekParity
                    )
                )
            }
        }
    }
}
```

```
private fun parseNonSubgroupLesson(

    lessonInfo: Element,
    groupName: String,
    currentDay: String,
    pairNumber: Int,
    weekParity: WeekParity
): Lesson {
    val boldTexts = lessonInfo.select("b")
    val subjectTypeText = boldTexts.first()?.text()?.trim() ?: ""
    val match = Regex("""^(.+?)\s*,?\s*(ПЗ|ЛР|ЛК|ЛЕК)$""").find(subjectTypeText)
    val subject = match?.groupValues?.get(1)?.trim() ?: subjectTypeText
    val lessonType = mapLessonType(match?.groupValues?.get(2)?.trim())
    val teacher = lessonInfo.selectFirst("i")?.text()?.trim() ?: ""
    val room = boldTexts.last()?.text()?.trim() ?: ""
    return Lesson(
        groupName = groupName,
        dayOfWeek = currentDay,
        pairNumber = pairNumber,
        time = pairTimes[pairNumber] ?: "",
        subject = subject,
        lessonType = lessonType,
        teacher = teacher,
        room = room,
        weekParity = weekParity,
        subgroup = null,
        dayOfWeekIndex = dayIndex[currentDay] ?: 0
    )
}

private fun mapLessonType(short: String?): LessonType = LessonType.fromAbbr(short)
```

Лістинг А.5 Реалізація функції оновлення розкладу з сайту та Lottie-анімації

```
// LottieRepositoryImpl.kt
class LottieRepositoryImpl @Inject constructor(
    @ApplicationContext private val context: Context
) : LottieRepository {
    override suspend fun preloadLoginAnimation(): LottieComposition? {
        val result = LottieCompositionFactory.fromRawRes(
            context,
            R.raw.loading_anim
        )
        return result.result?.value
    }
}

//LoginScreen.kt
LottieAnimation(
    composition =
rememberLottieComposition(LottieCompositionSpec.RawRes(R.raw.loading_anim)).value,
    iterations = LottieConstants.IterateForever,
    modifier = Modifier.size(400.dp),
)
```

Лістинг А.6 Реалізація набору інструментальних UI тестів

```
@RunWith(AndroidJUnit4::class)
class SettingsScreenTest {
    @get:Rule
    val composeTestRule = createAndroidComposeRule<ComponentActivity>()

    @Test
    fun settingsScreen_displaysAllButtons() {
        composeTestRule.setContent {
            SettingsScreen(
                uiState = SettingsUiState(
                    isLoggedIn = true,
                    userEmail = "test@domain.com",
                    themeMode = ThemeMode.SYSTEM
                ),
                uiEffect = emptyFlow(),
                intent = FakeIntent(),
                onNavigateToLogin = {},
                onNavigateToSetup = {}
            )
        }

        composeTestRule.onNodeWithText("Синхронізувати в Google
Calendar").assertIsDisplayed()
        composeTestRule.onNodeWithText("Повторно завантажити
розклад").assertIsDisplayed()
        composeTestRule.onNodeWithText("Змінити групу").assertIsDisplayed()
        composeTestRule.onNodeWithText("Змінити тему").assertIsDisplayed()
        composeTestRule.onNodeWithText("Вийти з акаунту").assertIsDisplayed()
        composeTestRule.onNodeWithText("Ви увійшли як:
test@domain.com").assertIsDisplayed()
    }

    @Test
    fun clickingSyncButton_callsIntent() {
        val intent = FakeIntent()
        composeTestRule.setContent {
            SettingsScreen(
                uiState = SettingsUiState(
                    isLoggedIn = true,
                    userEmail = "test@domain.com",
                    themeMode = ThemeMode.SYSTEM
                ),
                uiEffect = emptyFlow(),
                intent = intent,
                onNavigateToLogin = {},
                onNavigateToSetup = {}
            )
        }

        composeTestRule.onNodeWithText("Синхронізувати в Google
Calendar").performClick()

        assert(intent.syncCalled)
    }

    @Test
    fun clickingReimportButton_callsIntent() {
```

```
        val intent = FakeIntent()
        composeTestRule.setContent {
            SettingsScreen(
                uiState = SettingsUiState(
                    isLoggedIn = true,
                    userEmail = "test@domain.com",
                    themeMode = ThemeMode.SYSTEM
                ),
                uiEffect = emptyFlow(),
                intent = intent,
                onNavigateToLogin = {},
                onNavigateToSetup = {}
            )
        }

        composeTestRule.onNodeWithText("Повторно завантажити розклад").performClick()
        assert(intent.reimportCalled)
    }

    @Test
    fun clickingChangeGroup_callsIntent() {
        val intent = FakeIntent()
        composeTestRule.setContent {
            SettingsScreen(
                uiState = SettingsUiState(
                    isLoggedIn = true,
                    userEmail = "test@domain.com",
                    themeMode = ThemeMode.SYSTEM
                ),
                uiEffect = emptyFlow(),
                intent = intent,
                onNavigateToLogin = {},
                onNavigateToSetup = {}
            )
        }
        composeTestRule.onNodeWithText("Змінити групу").performClick()
        assert(intent.changeGroupCalled)
    }

    @Test
    fun clickingDeleteCalendarEvents_callsIntent() {
        val intent = FakeIntent()
        composeTestRule.setContent {
            SettingsScreen(
                uiState = SettingsUiState(
                    isLoggedIn = true,
                    userEmail = "test@domain.com",
                    themeMode = ThemeMode.SYSTEM
                ),
                uiEffect = emptyFlow(),
                intent = intent,
                onNavigateToLogin = {},
                onNavigateToSetup = {}
            )
        }

        composeTestRule.onNodeWithText("Видалити розклад з Google
Calendar").performClick()
```

```
        assert(intent.deleteEventsCalled)
    }

    @Test
    fun clickingLogout_callsIntent() {
        val intent = FakeIntent()
        composeTestRule.setContent {
            SettingsScreen(
                uiState = SettingsUiState(
                    isLoggedIn = true,
                    userEmail = "test@domain.com",
                    themeMode = ThemeMode.SYSTEM
                ),
                uiEffect = emptyFlow(),
                intent = intent,
                onNavigateToLogin = {},
                onNavigateToSetup = {}
            )
        }

        composeTestRule.onNodeWithText("Вийти з акаунту").performClick()
        assert(intent.logoutCalled)
    }

    @Test
    fun clickingThemeSelector_opensModal() {
        val intent = FakeIntent()
        composeTestRule.setContent {
            SettingsScreen(
                uiState = SettingsUiState(
                    isLoggedIn = true,
                    userEmail = "test@domain.com",
                    themeMode = ThemeMode.SYSTEM
                ),
                uiEffect = emptyFlow(),
                intent = intent,
                onNavigateToLogin = {},
                onNavigateToSetup = {}
            )
        }

        composeTestRule.onNodeWithText("Змінити тему").performClick()
        composeTestRule.waitForIdle()
        composeTestRule.onNodeWithText("Світла").assertIsDisplayed()
    }
}
```

Лістинг А.7 Реалізація набору модульних Unit тестів

```
@OptIn(ExperimentalCoroutinesApi::class)
class SettingsViewModelTest {

    private val logoutUseCase = mockk<LogoutUseCase>(relaxed = true)
    private val reimportUseCase = mockk<LoadScheduleUseCase>()
    private val isLoggedInUseCase = mockk<GetIsLoggedInUseCase>()
    private val saveIsLoggedInUseCase = mockk<SaveIsLoggedInUseCase>(relaxed = true)
    private val syncUseCase = mockk<SyncCalendarUseCase>()
    private val deleteUseCase = mockk<DeleteCalendarEventsUseCase>()
    private val emailUseCase = mockk<GetUserEmailUseCase>()
    private val getThemeUseCase = mockk<GetThemeUseCase>()
    private val setThemeUseCase = mockk<SetThemeUseCase>(relaxed = true)

    private lateinit var viewModel: SettingsViewModel

    @Before
    fun setup() {
        Dispatchers.setMain(StandardTestDispatcher())
        every { isLoggedInUseCase() } returns flowOf(true)
        every { getThemeUseCase() } returns flowOf(ThemeMode.SYSTEM)
        every { emailUseCase() } returns "test@example.com"

        viewModel = SettingsViewModel(
            logoutUseCase,
            reimportUseCase,
            isLoggedInUseCase,
            saveIsLoggedInUseCase,
            syncUseCase,
            deleteUseCase,
            emailUseCase,
            getThemeUseCase,
            setThemeUseCase
        )
    }

    @After
    fun tearDown() {
        Dispatchers.resetMain()
    }

    @Test
    fun `onThemeChanged calls setThemeUseCase`() = runTest {
        viewModel.onThemeChanged(ThemeMode.DARK)
        advanceUntilIdle()
        coVerify { setThemeUseCase(ThemeMode.DARK) }
    }

    @Test
    fun `onLogoutClicked updates prefs and triggers logout`() = runTest {
        viewModel.onLogoutClicked()
        advanceUntilIdle()
        coVerify { saveIsLoggedInUseCase(false) }
        coVerify { logoutUseCase() }
    }
```

```
    @Test
    fun `onReimportSchedule emits success message`() = runTest {
        coEvery { reimportUseCase() } returns Resource.Success(Unit)

        viewModel.uiEffect.test {
            viewModel.onReimportSchedule()
            awaitItem().also {
                assert(it is SettingsUiEffect.ShowMessage && it.message == "Розклад
оновлено")
            }
            cancelAndIgnoreRemainingEvents()
        }
    }

    @Test
    fun `onDeleteCalendarEvents emits success message`() = runTest {
        coEvery { deleteUseCase() } returns Resource.Success(Unit)

        viewModel.uiEffect.test {
            viewModel.onDeleteCalendarEvents()
            awaitItem().also {
                assert(it is SettingsUiEffect.ShowMessage && it.message == "Розклад
видалено з Google Calendar")
            }
            cancelAndIgnoreRemainingEvents()
        }
    }

    @Test
    fun `onDeleteCalendarEvents emits error message`() = runTest {
        val errorMessage = "Помилка видалення"
        coEvery { deleteUseCase() } returns Resource.Error(errorMessage)

        viewModel.uiEffect.test {
            viewModel.onDeleteCalendarEvents()
            awaitItem().also {
                assert(it is SettingsUiEffect.ShowMessage && it.message ==
errorMessage)
            }
            cancelAndIgnoreRemainingEvents()
        }
    }

    @Test
    fun `onChangeGroup emits NavigateToSetup`() = runTest {
        viewModel.uiEffect.test {
            viewModel.onChangeGroup()
            awaitItem().also {
                assert(it == SettingsUiEffect.NavigateToSetup)
            }
            cancelAndIgnoreRemainingEvents()
        }
    }
```

```
    @Test
    fun `onSyncToGoogleCalendar emits success message`() = runTest {
        coEvery { syncUseCase() } returns Resource.Success(Unit)

        viewModel.uiEffect.test {
            viewModel.onSyncToGoogleCalendar()
            awaitItem().also {
                assert(it is SettingsUiEffect.ShowMessage && it.message ==
"Синхронізація завершена")
            }
            cancelAndIgnoreRemainingEvents()
        }
    }

    @Test
    fun `onSyncToGoogleCalendar emits error message`() = runTest {
        coEvery { syncUseCase() } returns Resource.Error("Помилка синхронізації")

        viewModel.uiEffect.test {
            viewModel.onSyncToGoogleCalendar()
            awaitItem().also {
                assert(it is SettingsUiEffect.ShowMessage && it.message == "Помилка
синхронізації")
            }
            cancelAndIgnoreRemainingEvents()
        }
    }
}
```

ДОДАТОК Б

Рисунки роботи програми

Рисунок Б.1 Авторизація через Google OAuth Firebase.

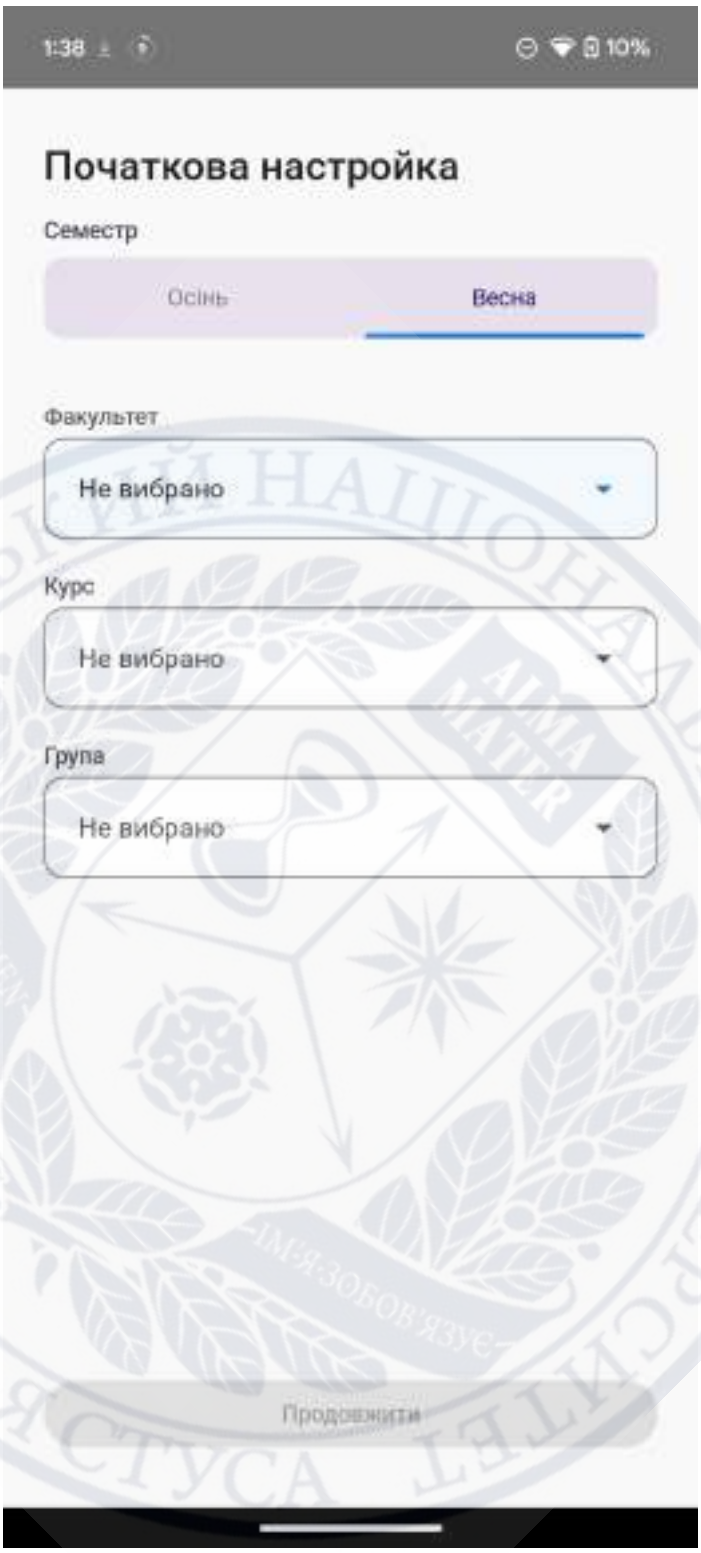


Рисунок Б.2 Екран вибору факультету, курсу та групи після авторизації.

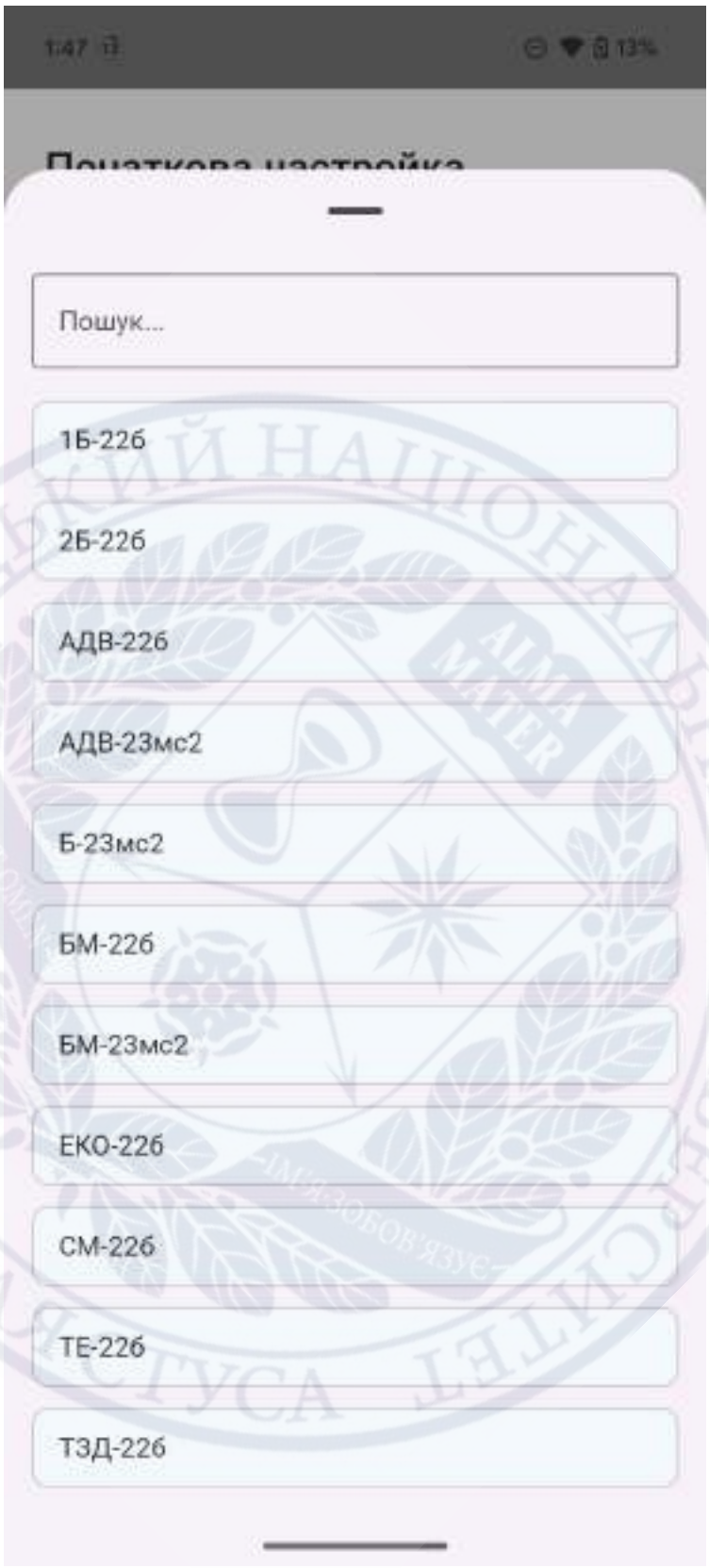


Рисунок Б.3 Екран списку завантажених груп на вибір користувача.

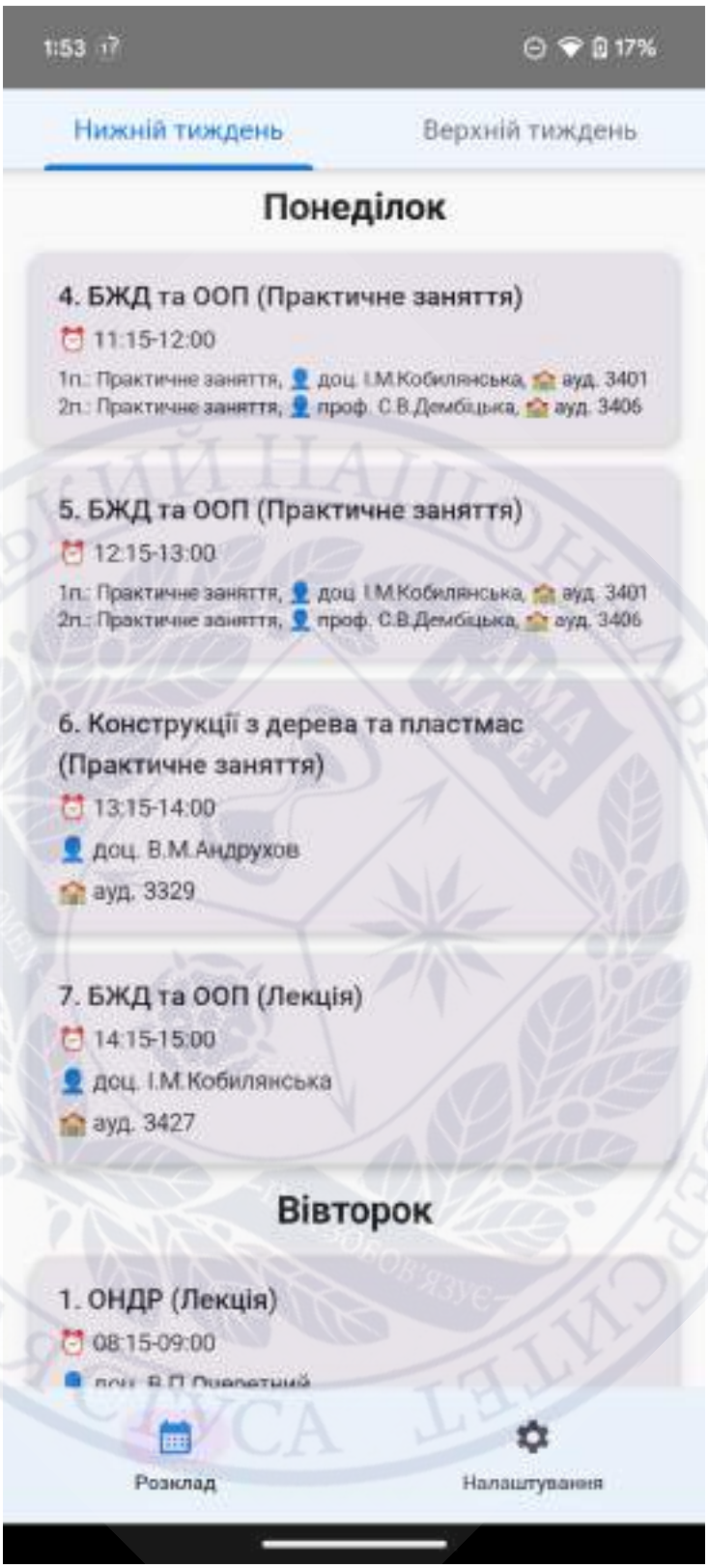


Рисунок Б.4 Екран списку розкладу відповідної групи, обраної користувачем.

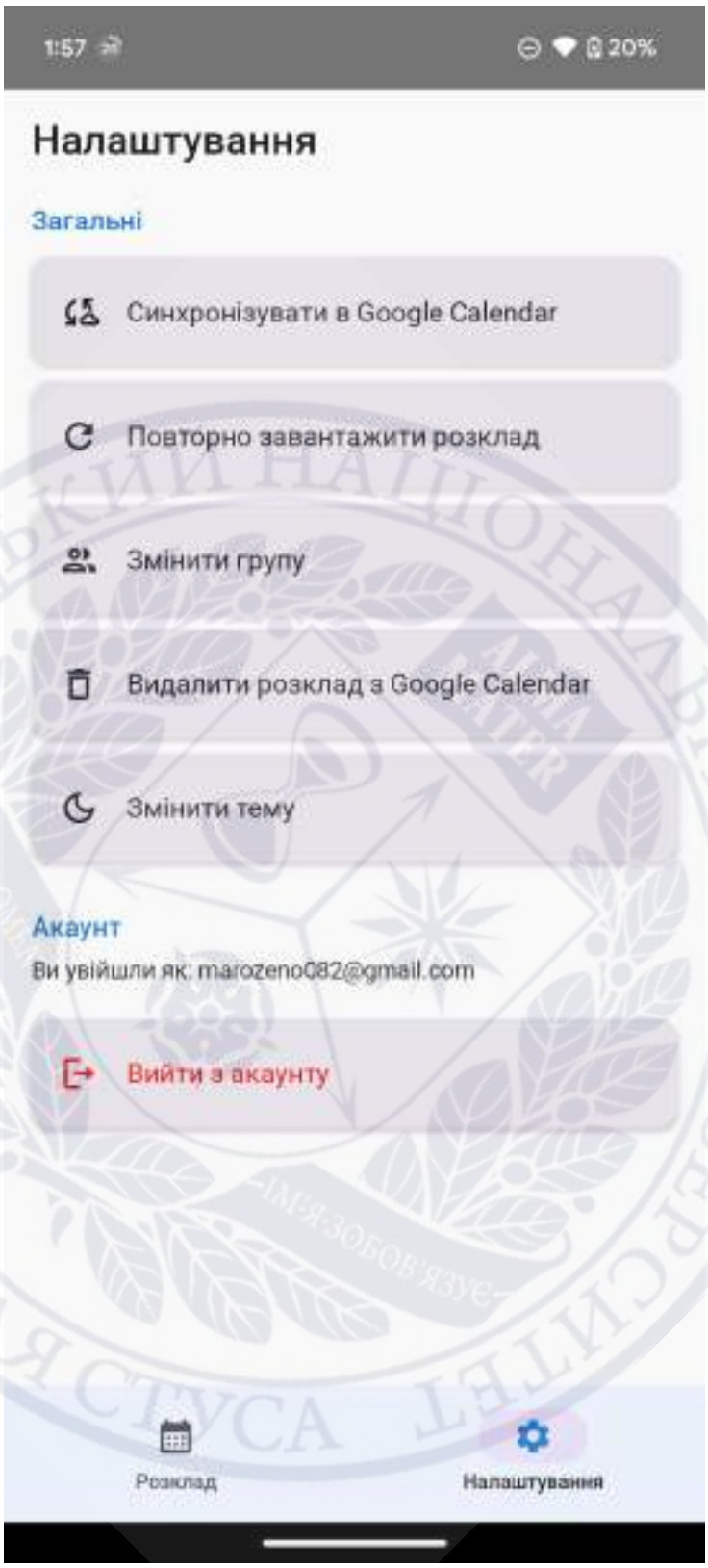


Рисунок Б.5 Екран налаштувань.

Рисунок Б.6 Результат експорту розкладу в Google Calendar.

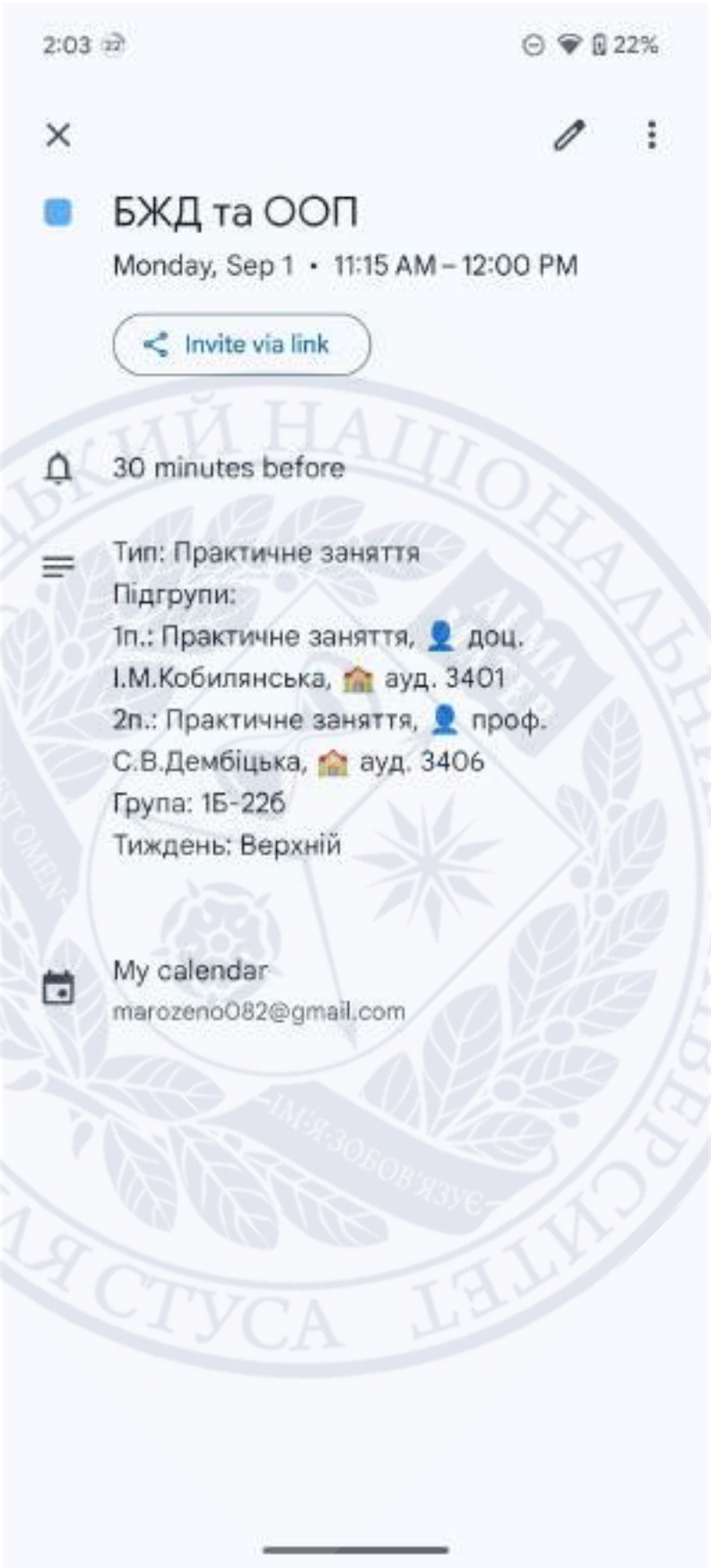


Рисунок Б.7 Результат експорту розкладу в Google Calendar.

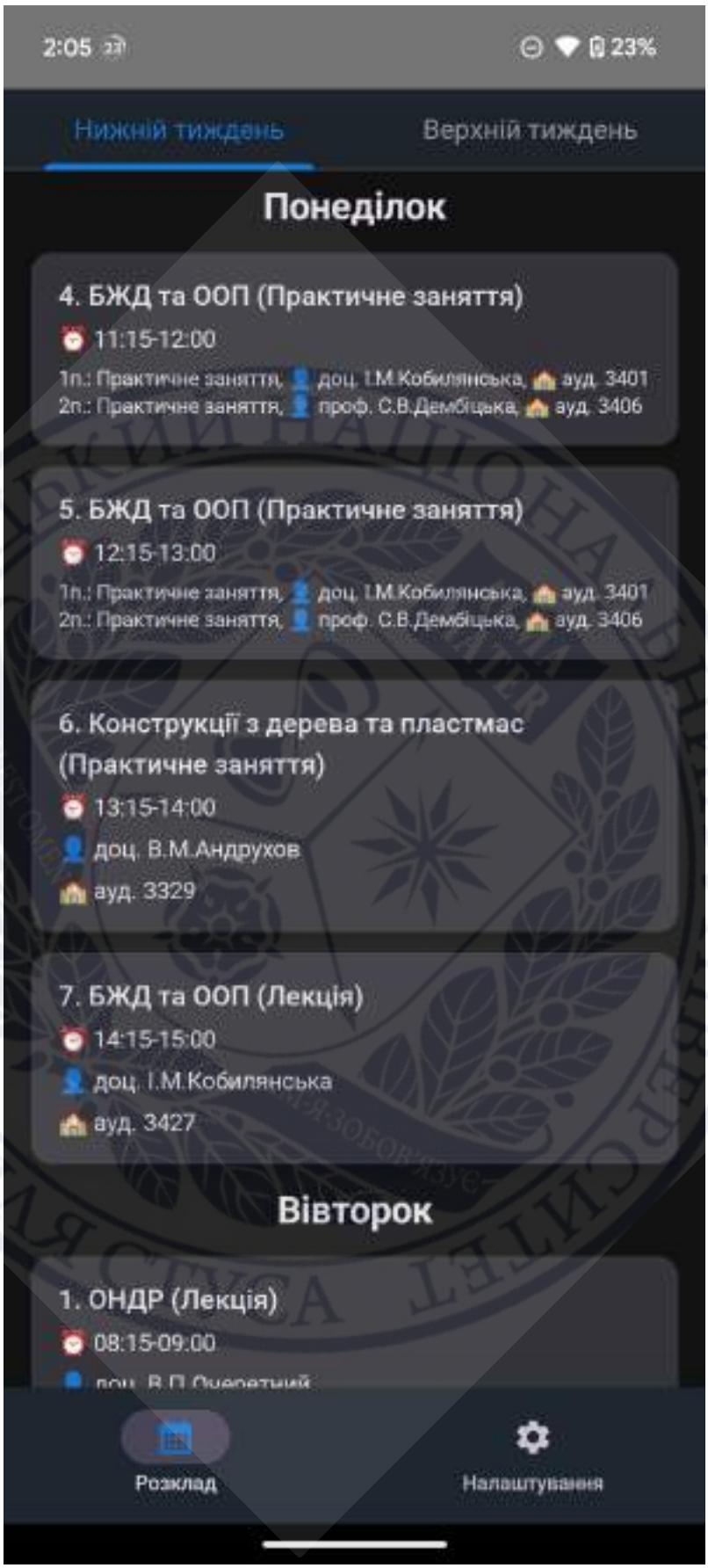


Рисунок Б.8 Екран розкладу із застосуванням темної теми.

ДЕКЛАРАЦІЯ

про дотримання академічної доброчесності

Я, _____
_____
_____
_____
_____

Повністю вказується ПІБ та статус (освітня (освітньо-наукова) програма – для здобувачів вищої освіти, назва кваліфікаційної роботи)

що нижче підписалась/підписався, розуміючи та підтримуючи загальновизнані засади справедливості, доброчесності та законності,

ЗОБОВ'ЯЗУЮСЬ:

дотримуватися принципів та правил академічної доброчесності, що визначені законодавством України, локальними нормативними актами Донецького національного університету імені Василя Стуса, положеннями, правилами, умовами, визначеними іншими суб'єктами, та не допускати їх порушення.

ПІДТВЕРДЖУЮ:

що мені відомі положення статті 42 Закону України «Про освіту»;

що у даній роботі не представляла/представляв чиїсь роботи повністю або частково як свої власні. Там, де я скористалася/скористався працею інших, я зробила/зробив відповідні посилання на джерела інформації;

що дана робота не передавалась іншим особам і подається вперше, не порушує авторських та суміжних прав закріплених статтями 21-25 Закону України «Про авторське право та суміжні права», а дані та інформація не отримувались в недозволений спосіб.

УСВІДОМЛЮЮ:

що ця робота може бути перевірена університетом на плагіат або інші порушення академічної доброчесності, в тому числі з використанням спеціалізованих сервісів;

що у разі порушення академічної доброчесності, до мене можуть бути застосовані процедури, передбачені законодавством України та Кодексом академічної доброчесності та корпоративної етики Донецького національного університету імені Василя Стуса, іншими локальними нормативними актами університету, та я можу бути притягнута/притягнутий до академічної відповідальності.

_____ _____

(дата) (підпис)