

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ ЗАКЛАД "ЛУГАНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА"

Факультет технологій та інформаційних систем

Дем'янов Вадим Анатолійович

Розробка мобільного застосунку метеопрогнозування

кваліфікаційна робота

**здобувача вищої освіти першого (бакалаврського) рівня
освітньої програми «Комп'ютерні науки та інформаційні технології»
за спеціальністю 122 Комп'ютерні науки**

Особистий підпис  Вадим ДЕМ'ЯНОВ

Науковий керівник  Галина КОЗУБ,
кандидат технічних наук, доцент
кафедри інформаційних технологій
та систем

Завідувач кафедри _____ Микола СЕМЕНОВ,
кандидат педагогічних наук, доцент
кафедри інформаційних технологій
та систем

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ ЗАКЛАД "ЛУГАНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА"

Факультет технологій та інформаційних систем

Кафедра Інформаційних технологій та систем

Освітній рівень бакалавр

Напрямок підготовки (спеціальність) 122 "Комп'ютерні науки"

Галузь знань 12 "Інформаційні технології"

ЗАТВЕРДЖУЮ
Завідувач кафедри ІТС
Микола СЕМЕНОВ

(підпис)

(ім'я, прізвище)

" " _____ 2026 р.

**ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ**

Дем'янов Вадим Анатолійович

1. Тема Розробка мобільного застосунку метеопрогнозування

Керівник кваліфікаційної роботи _____ Галина КОЗУБ, к.т.н., доцент
кафедри інформаційних технологій та систем

затверджена наказом по університету від " " _____ 202_ року №

2. Строк подання здобувачем вищої освіти проєкту "13" травня 2026 р.

3. Вихідні дані до роботи (проєкту) Сучасні технології розробки крос-платформних мобільних додатків (Flutter/Dart); API сервісу прогнозу погоди Open-Meteo; API музичного сервісу Deezer; технології клієнт-серверної взаємодії (REST API, JSON); методи геокодування та обробки координат; інструменти розробки мобільних додатків для платформ Android та iOS.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) Аналіз предметної області та існуючих рішень; огляд API погодних сервісів та музичних платформ; визначення недоліків існуючих рішень; формування функціональних та нефункціональних вимог до програмного забезпечення; проєктування мобільного додатку (UML-діаграми); розробка архітектури та програмних модулів; реалізація мобільного додатку Weatherfy; інтеграція API Open-Meteo та Deezer; тестування та аналіз якості програмного забезпечення; аналіз безпеки та захисту даних; порівняння з існуючими аналогами; висновки.

КАЛЕНДАРНИЙ ПЛАН-ГРАФІК
ВИКОНАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ

освітнього ступеня «бакалавр»
зі спеціальності 122 «Комп'ютерні науки»
студента Дем'янова Вадима Анатолійовича
Тема: Розробка мобільного застосунку метеопрогнозування
Науковий керівник: Г. О. Козуб, канд. техн. наук, доцент

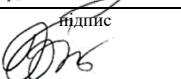
№ з/п	Назви робіт	Дата	Примітки
1	Аналіз предметної області: огляд мобільних додатків та існуючих API погодних сервісів	16.02.2026	Виконано
2	Дослідження музичних рекомендаційних систем та формування вимог до ПЗ	23.02.2026	Виконано
3	Проектний етап: розробка архітектури та UML-моделювання (Use Case, Class, Sequence diagrams)	02.03.2026	Виконано
4	Проектування структури додатку «Weatherfy» та опис логіки взаємодії з Open-Meteo та Deezer	16.03.2026	Виконано
5	Технічна реалізація модулів мобільного додатку та інтеграція API	23.03.2026	Виконано
6	Розробка та налаштування алгоритмів музичної рекомендаційної системи	30.03.2026	Виконано
7	Тестування ПЗ, аналіз безпеки даних та порівняння результатів з аналогами	06.04.2026	Виконано
8	Документація, підготовка висновків, фінальний звіт	13.04.2026	Виконано

Здобувач вищої освіти


підпис

Вадим Дем'янов

Керівник роботи


підпис

Галина КОЗУБ

АНОТАЦІЯ

Дем'янов В.А.

Тема: Розробка мобільного додатку для прогнозування погоди та формування персоналізованих рекомендацій.

Спеціальність: 122 «Комп'ютерні науки»

Установа: ДЗ «ЛНУ імені Т. Шевченка», 2026 р.

Кваліфікаційна робота містить: 109 с., 4 рис., 17 табл., 12 додат., 28 джерел.

Об'єкт дослідження – процес отримання, обробки та інтерпретації метеорологічних даних у мобільних інформаційних системах.

Предмет дослідження – методи інтеграції погодних API та побудови рекомендаційних механізмів на їх основі.

Мета роботи – розробка мобільного додатку для прогнозування погоди з інтегрованою системою персоналізованих рекомендацій.

Результати роботи. Розроблено мобільний додаток Weatherfy для отримання прогнозу погоди та формування рекомендацій на його основі. Використано API Open-Meteo для отримання метеорологічних даних та Deezer API для підбору музичного супроводу відповідно до погодних умов. Реалізовано механізм геокодування для визначення координат міст, інтерфейс користувача у вигляді карток та систему збереження обраних локацій. Проведено аналіз існуючих рішень у сфері погодних сервісів та визначено їх недоліки. Запропоновано підхід до поєднання погодних даних із персоналізованими рекомендаціями.

Висновки. В результаті розробки створено програмний продукт, що поєднує функції прогнозування та інтелектуальних рекомендацій, що підвищує зручність користування та розширює можливості традиційних погодних додатків.

Ключові слова: ПОГОДА, МОБІЛЬНИЙ ДОДАТОК, API, OPEN-METEO, DEEZER, , РЕКОМЕНДАЦІЙНА СИСТЕМА.

ANNOTATION

Demianov V.A.

Topic: Development of a mobile application for weather forecasting and generating personalized recommendations.

Specialty: 122 “Computer Science”

Institution: Luhansk Taras Shevchenko National University, 2026.

Qualification paper contains: 109 p., 4 fig., 17 tables., 12 appendices, 28 sources.

Object of research – the process of obtaining, processing, and interpreting meteorological data in mobile information systems.

Subject of research – methods of integrating weather APIs and building recommendation mechanisms based on them.

Purpose of the research – development of a mobile application for weather forecasting with an integrated personalized recommendation system.

Results of the research. A mobile application Weatherfy has been developed for obtaining weather forecasts and generating recommendations based on them. The Open-Meteo API is used to obtain meteorological data, and the Deezer API is used to select musical accompaniment according to weather conditions. A geocoding mechanism for determining city coordinates, a user interface in the form of cards, and a system for saving favorite locations have been implemented. An analysis of existing solutions in the field of weather services has been conducted, and their disadvantages have been identified. An approach to combining weather data with personalized recommendations has been proposed.

Conclusions. As a result of the development, a software product has been created that combines forecasting and intelligent recommendation functions, which increases user convenience and expands the capabilities of traditional weather applications.

Keywords: WEATHER, MOBILE APPLICATION, API, OPEN-METEO, DEEZER, RECOMMENDATION SYSTEM.

ЗМІСТ

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ РІШЕНЬ	12
1.1. Аналіз сучасних мобільних додатків прогнозування погоди	12
1.2. Огляд існуючих API погодних сервісів	15
1.3. Аналіз музичних сервісів та рекомендаційних систем	20
1.4. Формування вимог до програмного забезпечення	24
1.5. Висновки до розділу 1	26
РОЗДІЛ 2. ПРОЄКТУВАННЯ МОБІЛЬНОГО ДОДАТКУ WEATHERFY	28
2.1. Загальна архітектура системи	28
2.2. Моделювання системи засобами UML. Архітектурна діаграма компонентів	33
2.2.1. Діаграма варіантів використання (Use Case)	34
2.2.2. Діаграми послідовності (Sequence Diagram)	37
2.3. Проєктування структури мобільного додатку	41
2.4. Опис взаємодії з API Open-Meteo та Deezer	46
2.5. Висновки до розділу 2	52
РОЗДІЛ 3. РЕАЛІЗАЦІЯ ТА АНАЛІЗ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	55
3.1. Реалізація мобільного додатку	55
3.2. Реалізація серверної частини – Weatherfy API	63
3.3. Аналіз якості та тестування програмного забезпечення	68
3.4. Аналіз безпеки та захисту даних	73
3.5. Системні та апаратні вимоги	76
3.6. Порівняння з існуючими аналогами	78
3.7. Висновки до розділу 3	79
ВИСНОВКИ	81
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	83
ДОДАТКИ	86

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API	-	Application Programming Interface - інтерфейс програмування застосунків
AOT	-	Ahead-of-Time - компіляція у нативний код до виконання
CORS	-	Cross-Origin Resource Sharing - механізм контролю міжсайтових запитів
CPU	-	Central Processing Unit - центральний процесор
CSV	-	Comma-Separated Values - формат табличних даних
DTO	-	Data Transfer Object - об'єкт передачі даних між шарами
ECMWF	-	European Centre for Medium-Range Weather Forecasts - Європейський центр середньострокових прогнозів погоди
GDPR	-	General Data Protection Regulation - Загальний регламент захисту даних ЄС
GFS	-	Global Forecast System - глобальна система прогнозування погоди (США)
GPS	-	Global Positioning System - глобальна система позиціонування
HTTP	-	HyperText Transfer Protocol - протокол передачі гіпертексту
HTTPS	-	HyperText Transfer Protocol Secure - захищений протокол передачі гіпертексту
ID	-	Identifier - унікальний ідентифікатор
IP	-	Internet Protocol - мережевий протокол
IPA	-	iOS Package Archive - формат пакету застосунку для iOS
JSON	-	JavaScript Object Notation - текстовий формат обміну даними
NWP	-	Numerical Weather Prediction - чисельне прогнозування погоди
OAuth	-	Open Authorization - відкритий протокол авторизації
APK	-	Android Package Kit - формат пакету застосунку для Android
RAM	-	Random Access Memory - оперативна пам'ять
REST	-	Representational State Transfer - архітектурний стиль побудови API
SDK	-	Software Development Kit - набір інструментів розробника
TCP	-	Transmission Control Protocol - протокол керування передачею
TLS	-	Transport Layer Security - протокол захисту транспортного рівня
TTL	-	Time To Live - час життя кешованих даних
UI	-	User Interface - інтерфейс користувача

UML	-	Unified Modeling Language - уніфікована мова моделювання
URL	-	Uniform Resource Locator - уніфікований вказівник ресурсу
UTC	-	Coordinated Universal Time - всесвітній координований час
WMO	-	World Meteorological Organization - Всесвітня метеорологічна організація; також код погодних умов за стандартом WMO
XML	-	Extensible Markup Language - розширювана мова розмітки

Терміни та технології

Android	-	мобільна операційна система на базі Linux від Google
Clean Architecture	-	архітектурний підхід з чітким розподілом відповідальності між шарами системи
Dart	-	об'єктно-орієнтована мова програмування, розроблена Google
Deezer	-	французький стримінговий музичний сервіс з публічним API
Docker	-	платформа контейнеризації застосунків
Flutter	-	фреймворк від Google для розробки кросплатформних мобільних застосунків
iOS	-	мобільна операційна система від Apple
Nominatim	-	сервіс геокодингу на основі даних OpenStreetMap
Open-Meteo	-	відкритий безкоштовний API метеорологічних прогнозів
Riverpod	-	бібліотека управління станом для Flutter
Shelf	-	серверний веб-фреймворк для мови Dart
SharedPreferences	-	механізм локального сховища ключ-значення для Android та iOS
Spotify	-	шведський стримінговий музичний сервіс

ВСТУП

Актуальність роботи. На сучасному етапі розвитку інформаційних технологій мобільні додатки є однією з найбільш поширених форм доступу до інформації та сервісів. Зокрема, сервіси прогнозування погоди широко використовуються у повсякденному житті для планування діяльності, організації подорожей, роботи та дозвілля.

Разом з тим, більшість існуючих мобільних додатків обмежуються відображенням стандартних метеорологічних показників, таких як температура, вологість, швидкість вітру та інші параметри, без урахування індивідуальних потреб користувача. Це знижує ефективність використання таких систем та не дозволяє повною мірою реалізувати їх потенціал.

Одним із перспективних напрямків розвитку є створення мобільних інформаційних систем з елементами інтелектуального аналізу даних, які дозволяють формувати персоналізовані рекомендації на основі зовнішніх факторів, зокрема погодних умов. У роботі Trivedi R. та співавторів описується концепція *weather-aware recommendation engine*, у якій погодні параметри враховуються при формуванні персоналізованих рекомендацій для користувача [3]. Поєднання прогнозу погоди з іншими сервісами, такими як музичні платформи, відкриває нові можливості для підвищення зручності користування та рівня взаємодії з користувачем.

Сучасні технології, зокрема використання відкритих API, таких як Open-Meteo та Deezer, дозволяють реалізувати інтегровані програмні рішення без необхідності створення власної інфраструктури збору даних. Це сприяє розробці масштабованих, гнучких та функціональних програмних продуктів.

Таким чином, актуальність роботи полягає у необхідності розробки мобільного додатку, який поєднає функції прогнозування погоди з системою персоналізованих рекомендацій, що дозволяє підвищити ефективність використання інформації та забезпечити більш зручний досвід користувача.

Мета роботи. Метою бакалаврської роботи є розробка мобільного додатку для прогнозування погоди з реалізацією системи персоналізованих рекомендацій на основі погодних умов.

Завдання дослідження. Досягнення мети включає розв’язання таких завдань:

- провести аналіз предметної області та існуючих мобільних додатків прогнозування погоди;
- виконати огляд API погодних сервісів та музичних платформ;
- визначити недоліки існуючих рішень;
- сформулювати функціональні та нефункціональні вимоги до програмного забезпечення;
- розробити архітектуру мобільного додатку;
- виконати моделювання системи з використанням UML-діаграм;
- реалізувати інтеграцію з API Open-Meteo та Deezer;
- розробити мобільний додаток Weatherfy;
- провести тестування та аналіз якості програмного забезпечення;
- виконати аналіз безпеки та захисту даних;
- провести порівняння з існуючими аналогами.

Об’єктом дослідження є процес отримання та обробки метеорологічних даних у мобільних інформаційних системах.

Предметом дослідження є методи інтеграції зовнішніх API та побудови рекомендаційних систем на основі погодних даних.

Практичне значення отриманих результатів полягає у створенні мобільного додатку, що забезпечує отримання актуального прогнозу погоди та формування персоналізованих рекомендацій для користувача. Розроблений програмний продукт може бути використаний для підвищення ефективності планування повсякденної діяльності.

Основним результатом роботи є:

1. розробка мобільного додатку Weatherfy;
2. реалізація інтеграції з API Open-Meteo та Deezer;

3. розробка механізму формування рекомендацій на основі погодних даних;
4. створення зручного інтерфейсу користувача.

Апробація результатів роботи.

Результати роботи апробовано на XI Міжнародній науково-практичній конференції «GLOBALIZATION OF SCIENTIFIC KNOWLEDGE: INTERNATIONAL COOPERATION AND INTEGRATION OF SCIENCES» (15.05.2026; Вінниця, Україна – Відень, Австрія), організований ГО «Європейська наукова платформа» спільно з австрійською компанією LLC International Centre Corporative Management [27]. За матеріалами конференції опубліковано статтю «INTEGRATION OF MOOD ENGINE AND RECOMMENDATION ENGINE IN THE HYBRID CLIENT-SERVER ARCHITECTURE WEATHERFY» у збірнику «Грааль науки» (Ідентифікатор медіа R30-02704; ISSN: 2710-3056), випуск № 68.

Крім того, основні результати роботи опубліковано у фаховому науковому журналі категорії «Б» «Наука і техніка сьогодні» (№ 5(59), 2026, С. 4688–4704) у статті «Архітектура та реалізація мобільного додатку прогнозування погоди з інтегрованою рекомендаційною системою» (DOI: 10.52058/2786-6025-2026-5(59)-4688-4704) [28]. Сертифікати публікацій і участі наведено в додатках Ж, З.

Структура і обсяг роботи

Робота складається зі вступу, трьох розділів, висновків, списку використаних джерел та додатків.

Перший розділ містить аналіз предметної області та існуючих рішень. У другому розділі розглянуто проєктування та моделювання програмного забезпечення.

У третьому розділі наведено реалізацію мобільного додатку, результати тестування та аналіз безпеки.

Загальний обсяг роботи становить 109 сторінок, з них основна частина — 88 сторінок. Робота містить 17 таблиць, 4 рисунки, список використаних джерел з 28 найменувань та 12 додатків.

РОЗДІЛ 1

АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ РІШЕНЬ

1.1. Аналіз сучасних мобільних додатків прогнозування погоди

Сучасні мобільні додатки, призначені для прогнозування погоди, є одним із найбільш затребуваних типів інформаційних систем у сегменті масового користувача. Вони надають широкий спектр функціональних можливостей, зокрема перегляд поточних погодних умов, погодинний та довгостроковий прогноз (на 5–14 діб), а також отримання деталізованої інформації про атмосферний тиск, відносну вологість, швидкість та напрям вітру, рівень опадів, видимість, ультрафіолетовий індекс та інші метеорологічні параметри.

Класифікація функціональних можливостей. Узагальнюючи досвід використання наявних рішень, доцільно виділити такі базові категорії функцій:

- інформаційно-довідкові - відображення поточних метеорологічних показників для обраного географічного пункту;
- прогностичні - надання погодинного, добового та тижневого прогнозів;
- локаційні - використання даних геолокації для автоматичного визначення місцезнаходження користувача та прив'язки прогнозу до нього;
- картографічні - інтеграція з картографічними сервісами (наприклад, Google Maps, OpenStreetMap) для візуалізації розподілу опадів, хмарності, атмосферного тиску або температури в просторовому контексті;
- комунікаційні - система push-сповіщень про різкі зміни погодних умов (різке зниження/підвищення температури, шквалистий вітер, ожеледицю, грози, туман тощо).

Огляд найбільш поширених рішень. Серед мобільних застосунків, що займають лідируючі позиції за кількістю завантажень та користувацькими оцінками, слід назвати Google Weather, AccuWeather, Weather Underground, The Weather Channel, Yahoo Weather, Windy.com, WeatherBug та інші. Зазначені програмні продукти забезпечують відносно високу точність прогнозу завдяки

використанню різномірних джерел метеорологічних даних: наземних метеостанцій, супутникових спостережень, метеорологічних радарів, а також даних чисельного прогнозування погоди (NWP - Numerical Weather Prediction), що базуються на використанні моделей машинного навчання та багатофакторного аналізу погодних даних [4], які реалізуються глобальними моделями (GFS, ECMWF, ICON) або локальними мезомасштабними моделями. Окремі додатки (наприклад, Weather Underground) використовують також краудсорсингові дані від персональних метеостанцій користувачів, що дозволяє підвищити просторову роздільну здатність прогнозу.

Недоліки існуючих рішень. Попри значний функціонал, більшість розглянутих мобільних додатків мають низку суттєвих недоліків, які доцільно структурувати за наступними аспектами.

Перший недолік - когнітивна надлишковість інтерфейсу. Зазначені додатки орієнтовані на одночасне відображення великого обсягу метеорологічних даних (температура, вологість, тиск, вітер, опади, УФ-індекс, якість повітря тощо), що призводить до зростання когнітивного навантаження на користувача. Особливо це актуально для випадків, коли кінцевий користувач не має спеціалізованих знань у галузі метеорології та потребує не стільки сирих даних, скільки інтерпретованих рекомендацій.

Другий недолік - недостатній рівень персоналізації. У переважній більшості додатків підтримка персоналізації обмежується можливістю вибору одиниць вимірювання (температура в °C/°F, тиск у мм рт. ст./гПа), вибором кількості відображених днів прогнозу та, в окремих випадках, налаштуванням зовнішнього вигляду (теми оформлення). При цьому відсутнє врахування індивідуальних особливостей користувача: його віку, стану здоров'я (наприклад, метеочутливість, наявність хронічних захворювань, що загострюються за певних погодних умов), роду діяльності (робота на відкритому повітрі, водіння автомобіля, заняття спортом, сільське господарство), уподобань щодо активностей, а також контексту (місцезнаходження, час доби, сезон).

Третій недолік - примітивність рекомендаційних механізмів. У тих поодиноких випадках, коли додатки містять елементи рекомендацій (наприклад, поради «взяти парасольку», «одягнутися тепліше», «небезпечно для шкіри»), вони базуються на простих порогових значеннях окремих метеорологічних змінних. Зокрема, рекомендація взяти парасольку генерується при перевищенні порогового значення ймовірності опадів (наприклад $>50\%$), без урахування інтенсивності, тривалості, типу опадів (дощ, сніг, мряка) або швидкості вітру, яка унеможливорює ефективне використання парасольки. Такі механізми не враховують комплексний вплив одночасно діючих погодних факторів (наприклад, поєднання низької температури, високої вологості та вітру для формування індексу охолодження вітром), а також не беруть до уваги індивідуальний контекст користувача.

Четвертий недолік - відсутність міжсервісної інтеграції. Переважна більшість додатків функціонують як автономні системи та не передбачають інтеграції погодних даних з іншими типами сервісів, які активно використовує сучасний користувач. До таких сервісів належать:

- музичні платформи (Spotify, Apple Music, YouTube Music) - для формування плейлистів відповідно до погоди (енергійна музика в сонячний день, релаксаційна - під час дощу);
- календарі та планувальники завдань (Google Calendar, Microsoft Outlook) - для адаптації планів користувача до прогнозованих погодних умов (перенесення активностей на відкритому повітрі у разі несприятливої погоди);
- системи рекомендацій контенту - для пропозиції фільмів, книг, рецептів страв, що відповідають поточному погодному контексту («дощовий день», «спекотний полудень», «морозний вечір»);
- фітнес-трекери та системи моніторингу здоров'я (Google Fit, Apple Health, Garmin) - для врахування погодних умов при оцінці фізичної активності користувача та формуванні застережень.

Відсутність такої інтеграції обмежує можливості формування цілісного та глибокого користувацького досвіду (user experience), а також знижує потенціал розвитку подібних систем у напрямку адаптивної та проактивної поведінки.

Узагальнення та обґрунтування актуальності дослідження. Проведений аналіз свідчить про те, що на ринку мобільних додатків для прогнозування погоди домінують рішення, які ефективно вирішують завдання збору, обробки та візуалізації метеорологічних даних. Однак ці системи здебільшого залишаються на рівні «пасивних інформаторів» - вони надають дані, але не здійснюють їхньої інтелектуальної інтерпретації з урахуванням потреб конкретного користувача. Дослідження Liu Y. та співавторів демонструє наявність взаємозв'язку між погодними умовами та емоційним станом людини, зокрема сприйняттям температурного комфорту через аналіз поведінки користувачів у соціальних мережах [\[1\]](#).

Таким чином, існує обґрунтована потреба у створенні такого програмного продукту, який:

- забезпечує точний та релевантний метеорологічний прогноз;
- реалізує інтелектуальну обробку погодних даних з використанням багатofакторного аналізу;
- формує персоналізовані рекомендації, адаптовані до індивідуальних характеристик, контексту та уподобань користувача;
- передбачає можливість інтеграції із зовнішніми сервісами (музичними, календарними, рекомендаційними) для розширення функціональності та підвищення користувацької цінності.

Зазначені вимоги формують проблемне поле, у межах якого виконується подальше дослідження та практична розробка, що описується в наступних розділах дипломної роботи.

1.2. Огляд існуючих API погодних сервісів

Для забезпечення функціонування мобільного додатку, що розробляється, необхідною передумовою є отримання актуальних метеорологічних даних від

зовнішнього джерела. Найбільш поширеним та технологічно доцільним способом інтеграції таких даних є використання прикладних програмних інтерфейсів (API - Application Programming Interface) спеціалізованих погодних сервісів. Як зазначається в документації Open-Meteo, використання REST API дозволяє отримувати метеорологічні дані у стандартизованому JSON-форматі без необхідності розгортання власної інфраструктури прогнозування погоди [13]. Цей підхід дозволяє отримувати структуровані дані в машинозчитувальних форматах (переважно JSON або XML) без необхідності самостійного розгортання метеорологічної інфраструктури, що є економічно недоцільним у межах дипломного проєктування.

У даному підрозділі виконується порівняльний аналіз доступних API-рішень за сукупністю критеріїв, що включають: точність та оновлюваність даних, обсяг надаваної інформації (набір метеорологічних змінних), наявність безкоштовного тарифного плану, обмеження за кількістю запитів, географічне покриття, підтримку історичних даних, а також зручність документації та інтеграції.

Ключові критерії вибору API для науково-дослідної розробки. Вибір джерела метеоданих для академічного проєкту, зокрема дипломної роботи, має враховувати специфічні обмеження та цілі дослідження. Основними критеріями в даному контексті виступають:

- безоплатність доступу - відсутність фінансових витрат на етапі розробки, тестування та функціонування кінцевого програмного продукту;
- відсутність або ліберальність обмежень за кількістю запитів - можливість виконувати необхідну кількість тестових звернень без ризику блокування;
- повнота метеорологічних параметрів - наявність змінних, необхідних для реалізації алгоритмів персоналізованих рекомендацій (температура, вологість, швидкість вітру, опади, атмосферний тиск);
- наявність документації з відкритим доступом - можливість самостійного вивчення специфікації API без додаткових дозволів;

- підтримка історичних даних - необхідність для тренування та валідації рекомендаційних моделей;
- можливість локального або умовно локального розгортання - опціональний критерій, що підвищує автономність розробки.

Порівняльний аналіз сучасних погодних API. У табл. 1.1 наведено порівняльну характеристику найбільш поширених API-сервісів, які надають метеорологічні дані на комерційній або умовно безкоштовній основі.

Таблиця 1.1 – Порівняльна характеристика API погодних сервісів

Назва API	Безкоштовний план	Ліміт запитів (безкоштовно)	Історичні дані	Основні формати	Відкрита документація
OpenWeatherMap	Так	60 зап./хв, 1 млн/міс	Так (платний)	JSON, XML	Так
WeatherAPI (раніше Weather2020)	Так	100 000/міс	Обмежено	JSON, XML	Так
Tomorrow.io (раніше ClimaCell)	Так	~500 зап./день	Так	JSON	Так
AccuWeather	Так (обмежений)	обмежений безкоштовний тариф	Частково	JSON	Так
Open-Meteo	Так (повністю безоплатний)	Не обмежений (етичне використання)	Так (з 1959 р.)	JSON, CSV	Так
National Weather Service (NWS) API	Так	Не обмежений (публічний)	Так	JSON, GeoJSON	Так
Visual Crossing	Так	1000 зап./день	Так	JSON, CSV	Так

Джерело: складено автором на основі аналізу документації зазначених сервісів (станом на 2025–2026 рр.)

Обґрунтування вибору Open-Meteo як основного джерела даних. На основі проведеного порівняльного аналізу для реалізації програмної системи, що розробляється в межах дипломної роботи, було обрано API Open-Meteo. Вибір обумовлено сукупністю наступних факторів.

Перша перевага - повна безоплатність без функціональних обмежень. На відміну від більшості конкурентів (OpenWeatherMap, AccuWeather, Tomorrow.io), які надають безкоштовний доступ лише до базового набору функцій із жорсткими обмеженнями за кількістю запитів, Open-Meteo пропонує

необмежений (у межах добросовісного використання) доступ до всіх метеорологічних даних. Це дозволяє виконувати необхідну кількість тестових запитів під час розробки та забезпечувати безперебійну роботу кінцевого додатку без ризику фінансових витрат або раптового блокування.

Друга перевага - висока точність за рахунок інтеграції провідних глобальних моделей. Згідно з офіційною документацією сервісу, Open-Meteo агрегує дані з декількох авторитетних метеорологічних моделей [\[13\]](#), зокрема:

- ECMWF IFS (Європейський центр середньострокових прогнозів погоди) - одна з найточніших глобальних моделей;
- GFS (Global Forecast System, США) - модель з високою просторовою роздільною здатністю;
- DWD ICON (Німецька метеорологічна служба) - модель, оптимізована для європейського регіону;
- MET Norway (Норвезький метеорологічний інститут).

Це дозволяє обирати оптимальне джерело залежно від географічного регіону та необхідної точності прогнозу.

Третя перевага - підтримка історичних даних за тривалий період. Open-Meteo надає доступ до архівних метеорологічних даних, починаючи з 1959 року (для моделі ERA5 від ECMWF). Це є критично важливим для науково-дослідної складової роботи, оскільки дозволяє:

- виконувати статистичний аналіз погодних патернів;
- тренувати рекомендаційні моделі на ретроспективних даних;
- проводити валідацію алгоритмів на історичних вибірках.

Четверта перевага - багатий набір метеорологічних змінних. На відміну від більшості безкоштовних API, Open-Meteo надає доступ до понад 40 метеорологічних параметрів, включаючи:

- температуру повітря (поточну, мінімальну, максимальну);
- відносну вологість;
- швидкість та напрям вітру (включно з поривами);
- атмосферний тиск на рівні моря;

- різні типи опадів (дощ, сніг, град);
- хмарність;
- ультрафіолетовий індекс;
- видимість;
- температуру ґрунту та сніговий покрив.

Така повнота даних є достатньою для реалізації складних алгоритмів багатофакторного аналізу та формування персоналізованих рекомендацій відповідно до мети дипломної роботи.

П'ята перевага - відсутність вимоги до реєстрації API-ключа. Open-Meteo не потребує попередньої реєстрації або отримання ключа доступу. Запити виконуються безпосередньо за URL-адресою з параметрами у рядку запиту. Це спрощує процес інтеграції, зменшує кількість технічних залежностей та усуває ризики, пов'язані з витоком або закінченням терміну дії ключа.

Шоста перевага - зручний API з гнучкими параметрами запиту. Open-Meteo підтримує:

- вибір часового поясу;
- налаштування часового горизонту прогнозу (до 16 діб);
- завдання набору необхідних змінних (запит лише тих полів, які реально використовуються);
- підтримку кількох форматів вихідних даних (JSON, CSV, XLSX);
- можливість отримання агрегованих статистик (середні, мінімальні, максимальні значення).

API Open-Meteo має також певні обмеження: відсутність офіційної SLA-угоди та можливі коливання швидкодії залежно від навантаження. Для мінімізації впливу цих обмежень у межах дипломної роботи пропонується:

- кешування даних на стороні клієнта. Отримані від Weatherfy_API дані зберігаються локально з позначкою часу. У разі недоступності сервера використовуються кешовані дані як резервне джерело.

- обробка помилок мережі. При невдалому запиті виконуються повторні спроби з експоненційною затримкою. При перевищенні ліміту спроб користувач інформується про неможливість отримання актуальних даних.

- граціозна деградація функціоналу. У разі часткової недоступності зовнішніх сервісів (наприклад, Deezer API) повертаються дані про погоду без музичних рекомендацій.

Проведений огляд показав, що більшість комерційних погодних API (AccuWeather, Tomorrow.io, Visual Crossing) накладають суттєві обмеження на безкоштовне використання. Натомість Open-Meteo пропонує повнофункціональний безоплатний доступ.

За результатами порівняльного аналізу як основне джерело метеорологічних даних обрано Open-Meteo. Ключовими критеріями вибору стали повна безоплатність сервісу, відсутність обов'язкової реєстрації та жорстких лімітів запитів, висока точність прогнозів на основі моделей ECMWF та GFS, широкий набір метеопараметрів і зручна REST-документація. Особливості інтеграції з обраним API розглядаються в розділі 3 у контексті реалізації серверного компонента.

1.3. Аналіз музичних сервісів та рекомендаційних систем

Для реалізації функціональності персоналізованих рекомендацій, що враховують погодні умови, необхідною складовою є інтеграція мобільного додатку із зовнішнім музичним сервісом. Як зазначають Duncan B. A. та ін., погодні умови можуть використовуватись як додатковий контекстний параметр у рекомендаційних системах для підвищення релевантності рекомендацій [2]. Така інтеграція дозволяє пропонувати користувачеві музичний контент, що відповідає поточному погодному контексту (сонячна, дощова, спекотна або холодна погода), підвищуючи рівень залученості та задоволеності користувачького досвіду.

Загальна характеристика музичних сервісів. На сучасному ринку цифрової дистрибуції музики домінують декілька великих платформ, які надають програмний доступ до своїх каталогів через публічні API. До найбільш

поширених належать Spotify, Apple Music, Deezer, YouTube Music та SoundCloud. Ключовими критеріями вибору музичного сервісу для інтеграції в межах дипломної роботи виступають: наявність вільного або умовно безкоштовного доступу до API, простота автентифікації, можливість відтворення демо-фрагментів треків, а також обсяг музичного каталогу.

Для реалізації програмної системи, що розробляється, обрано музичний сервіс Deezer. Вибір обумовлений наступними факторами.

Перша перевага - ліберальні умови доступу до API. Deezer надає можливість виконувати пошук треків, отримувати метадані (назва, виконавець, альбом, тривалість, жанр) та посилання на 30-секундні демо-фрагменти без необхідності складної автентифікації через протокол OAuth. Для базових запитів достатньо використання простого API-ключа або навіть публічних ендпоінтів. Це суттєво спрощує архітектуру додатку та знижує поріг входження для розробки.

Друга перевага - наявність 30-секундних попередніх прослуховувань. У документації Deezer API зазначено, що сервіс надає preview-посилання для попереднього прослуховування треків тривалістю до 30 секунд без необхідності авторизації користувача [7]. Це дозволяє користувачеві мобільного додатку ознайомитися з фрагментом рекомендованої композиції без необхідності повноцінної автентифікації, підписки або переходу в основний застосунок Deezer. У межах даної дипломної роботи передбачається відтворення саме таких демо-фрагментів безпосередньо в інтерфейсі розроблюваного додатку.

Третя перевага - простота інтеграції через гіперпосилання. У зв'язку з відсутністю реалізації повноцінного OAuth-потoku в межах даного проєкту, інтеграція з Deezer обмежується формуванням гіперпосилань на сторінки треків, альбомів або виконавців на веб-сайті Deezer. Користувач, за бажанням, може перейти за таким посиланням для прослуховування повної версії композиції в офіційному веб-плеєрі або мобільному застосунку Deezer. Такий підхід є технічно достатнім для демонстрації роботи рекомендаційного механізму та не вимагає розробки складної системи керування доступом.

Четверта перевага - значний музичний каталог. Відповідно до офіційних матеріалів компанії Deezer, музичний каталог платформи перевищує 90 мільйонів композицій різних жанрів та виконавців [8], що забезпечує високу ймовірність знаходження релевантних композицій для будь-якого погодного сценарію. API підтримує пошук за ключовими словами, фільтрацію за жанрами, отримання популярних треків та рекомендацій на основі схожих виконавців.

Рекомендаційні системи (recommender systems) є класом програмних рішень, призначених для передбачення вподобань користувача та пропонування релевантних об'єктів (товарів, послуг, контенту). У контексті даної роботи рекомендаційна система вирішує завдання добору музичних композицій на основі поточних метеорологічних умов та індивідуальних уподобань користувача. Сучасні рекомендаційні системи умовно поділяються на три категорії:

- контент-орієнтовані - ґрунтуються на аналізі атрибутів об'єктів (жанр, темп, настрій музики) та зіставленні їх з профілем користувача;
- колаборативні - використовують подібність між користувачами (користувачі зі схожими вподобаннями отримують схожі рекомендації);
- гібридні - поєднують обидва підходи.

У межах даної дипломної роботи пропонується використання спрощеної контент-орієнтованої моделі, в якій зіставлення між погодними умовами та музичними характеристиками виконується на основі попередньо визначених евристичних правил. Зокрема, сонячній та теплій погоді зіставляються енергійні композиції з високим темпом (поп, рок, електронна музика), дощовій - меланхолійні або інструментальні твори, холодній - спокійна або акустична музика. Індивідуальні вподобання користувача враховуються через можливість вибору улюблених жанрів та виконавців.

Взаємодія компонентів системи з Deezer API у межах даної роботи реалізується за наступною схемою:

1. Серверний компонент Weatherfy_API на основі отриманих від Open-Meteo погодних даних та збереженого профілю користувача формує пошуковий

запит (наприклад, назва жанру, виконавця або ключові слова, що характеризують настрій).

2. Weatherfy_API виконує HTTP GET-запит до публічного ендпоінту Deezer Search API: <https://API.deezer.com/search?q={запит}>.

3. Deezer API повертає JSON-відповідь, яка містить список треків із метаданими, включаючи: назву треку, ім'я виконавця, посилання на 30-секундний демо-фрагмент (preview), посилання на сторінку треку на Deezer (link), тривалість, а також ідентифікатори альбому та виконавця.

4. Weatherfy_API агрегує отримані дані разом із погодною інформацією та передає їх клієнтському застосунку Weatherfy у вигляді єдиної JSON-відповіді.

5. Клієнтський застосунок Weatherfy візуалізує отриманий список рекомендацій. Для кожного треку користувачеві пропонується:

- можливість прослуховування 30-секундного демо-фрагменту безпосередньо в інтерфейсі додатку;
- гіперпосилання для переходу на повну версію треку на веб-сайті Deezer.

Такий підхід дозволяє централізувати всю логіку роботи із зовнішніми API на серверному компоненті, спрощуючи клієнтський застосунок до рівня візуалізації готових даних. При цьому зберігається можливість використання Deezer API без впровадження складної OAuth-автентифікації.

Проведений аналіз музичних сервісів показав, що Deezer пропонує оптимальне співвідношення функціональності та простоти інтеграції для науково-дослідної розробки. Наявність публічного API без обов'язкової OAuth-автентифікації, підтримка 30-секундних демо-фрагментів, а також можливість використання простих гіперпосилань на повні версії треків роблять Deezer найбільш доцільним вибором у межах даної дипломної роботи. Запропонована модель інтеграції забезпечує всі необхідні механізми для функціонування погодно-залежної рекомендаційної системи музичного контенту.

1.4. Формування вимог до програмного забезпечення

На основі проведеного аналізу сучасних мобільних додатків прогнозування погоди (підрозділ 1.1), огляду існуючих API погодних сервісів (підрозділ 1.2) та аналізу музичних сервісів і рекомендаційних систем (підрозділ 1.3) сформульовано сукупність вимог до програмного забезпечення, що розробляється в межах дипломної роботи. Вимоги поділяються на функціональні та нефункціональні.

Функціональні вимоги визначають перелік завдань, які повинна виконувати система.

Для серверного компонента Weatherfy_API:

1. Інтеграція з API Open-Meteo - забезпечення отримання актуальних метеорологічних даних (температура, вологість, опади, швидкість вітру, атмосферний тиск) за географічними координатами.
2. Інтеграція з API Deezer - забезпечення пошуку музичних композицій за ключовими словами (жанр, виконавець, настрій) та отримання метаданих (назва, виконавець, посилання на 30-секундний демо-фрагмент, гіперпосилання на повну версію).
3. Аналіз погодних умов - визначення узагальненого показника «настрою» на основі метеорологічних даних (сонячно, дощово, хмарно, спекотно, холодно тощо).
4. Формування рекомендацій - генерування пошукового запиту до Deezer API на основі погодного контексту та (за наявності) збережених уподобань користувача.
5. Надання API для клієнтського застосунку - реалізація власного RESTful API, яке приймає запити з координатами користувача та повертає структуровану JSON-відповідь, що містить поточну погоду та список музичних рекомендацій.

Для клієнтського застосунку Weatherfy:

1. Визначення геолокації - отримання географічних координат користувача за допомогою вбудованого GPS-модуля або підтримка ручного введення місцезнаходження (відкритий геокодинг).

2. Комунікація з серверним API - виконання HTTP-запитів до Weatherfy_API з передачею координат та обробка отриманих відповідей.

3. Візуалізація погодних даних - відображення поточних метеорологічних показників у зручному для користувача форматі.

4. Візуалізація музичних рекомендацій - відображення списку рекомендованих треків (назва, виконавець) із можливістю прослуховування 30-секундного демо-фрагменту безпосередньо в інтерфейсі додатку.

5. Забезпечення переходу на Deezer - надання гіперпосилань на повні версії треків на веб-сайті Deezer для подальшого прослуховування.

6. Кешування даних - локальне зберігання отриманих даних з позначкою часу для використання у разі недоступності сервера.

7. Обробка помилок - коректне інформування користувача про помилки мережі, недоступність сервера або відсутність даних.

Нефункціональні вимоги визначають якісні характеристики системи.

1. Продуктивність - час відповіді серверного API (від отримання запиту до повернення результату) не повинен перевищувати 3 секунд для 95% запитів за нормальних умов роботи зовнішніх сервісів.

2. Надійність - система повинна коректно обробляти помилки зовнішніх API (Open-Meteo, Deezer) та надавати користувачеві часткові дані за наявності (граціозна деградація).

3. Автономність - клієнтський застосунок має зберігати працездатність (з використанням кешованих даних) при тимчасовій недоступності серверного компонента або мережі Інтернет.

4. Платформна сумісність - клієнтський застосунок розробляється для мобільної операційної системи Android (версія не нижче Android 5.0).

5. Зручність користувацького інтерфейсу - інтерфейс має бути інтуїтивно зрозумілим, мінімалістичним та забезпечувати швидкий доступ до основної функціональності (перегляд погоди та рекомендацій).

6. Масштабованість - архітектура системи (розділення на Weatherfy_API та Weatherfy) дозволяє в подальшому розширювати функціонал (додавання нових джерел даних, ускладнення рекомендаційних алгоритмів) без кардинальної зміни структури.

У процесі розробки слід враховувати наступні обмеження:

- використання лише безоплатних API та інструментів;
- відсутність реалізації OAuth-автентифікації (взаємодія з Deezer обмежується публічним API);
- зберігання профілю користувача (уподобань) не передбачається в межах даної версії системи; рекомендації формуються виключно на основі погодного контексту;
- система не передбачає використання резервного погодного API у разі недоступності Open-Meteo.

1.5. Висновки до розділу 1

У першому розділі дипломної роботи виконано аналіз предметної області, розглянуто існуючі рішення та обґрунтовано вибір технологій для розробки програмної системи погодно-залежних музичних рекомендацій.

Проведений аналіз сучасних мобільних додатків прогнозування погоди (підрозділ 1.1) показав, що більшість існуючих рішень орієнтовані на відображення великого обсягу метеоданих, мають обмежену персоналізацію та примітивні рекомендаційні механізми. Це обґрунтовує актуальність створення системи, яка забезпечує інтелектуальну обробку даних та формування персоналізованих рекомендацій.

Огляд існуючих API погодних сервісів (підрозділ 1.2) дозволив обрати Open-Meteo як основне джерело метеорологічних даних. Ключовими критеріями вибору стали повна безоплатність, відсутність жорстких лімітів, висока точність

та простота інтеграції. Визначено технологічний стек та розподіл відповідальності між компонентами системи.

Аналіз музичних сервісів та рекомендаційних систем (підрозділ 1.3) показав доцільність використання Deezer завдяки наявності публічного API без обов'язкової OAuth-автентифікації, підтримці 30-секундних демо-фрагментів та гіперпосилань. Визначено, що логіка роботи з Deezer API централізується на серверному компоненті Weatherfy_API, а клієнтський застосунок отримує готові рекомендації.

На основі проведеного аналізу сформульовано функціональні та нефункціональні вимоги до програмного забезпечення (підрозділ 1.4), а також визначено обмеження проєкту.

Таким чином, результати першого розділу створюють теоретичну та аналітичну базу для подальшого проєктування, реалізації та тестування програмної системи в наступних розділах дипломної роботи.

РОЗДІЛ 2

ПРОЄКТУВАННЯ МОБІЛЬНОГО ДОДАТКУ WEATHERFY

2.1. Загальна архітектура системи

Розроблювана програмна система «Weatherfy» являє собою клієнт-серверне рішення, призначене для надання користувачеві персоналізованих музичних та щоденних рекомендацій на основі поточних погодних умов. У даному підрозділі подається опис загальної архітектури системи, розподілу компонентів та їх взаємодії.

Система складається з двох основних компонентів: серверного (Weatherfy_API) та клієнтського (мобільний застосунок Weatherfy). Загальну архітектуру системи подано на рисунку А.1 (додаток А.1).

Серверний компонент (Weatherfy_API)

Серверний компонент реалізовано з використанням мови Dart та фреймворку Shelf. Його основна функція - надання трьох незалежних REST-ендпоінтів, кожен з яких відповідає за окремий аспект обробки даних:

Таблиця 2.1 – Опис ендпоінтів серверного компоненту системи

Ендпоінт	Метод	Призначення
/weather	GET	Отримання метеорологічних даних від Open-Meteo API за географічними координатами
/mood	POST	Обчислення показників настрою (energy та valence) через Mood Engine на основі погодних даних, широти та кліматичної зони
/music	GET	Отримання музичних рекомендацій від Deezer API на основі переданих параметрів energy та valence

Використання REST-підходу забезпечує незалежність клієнтської та серверної частин, а також спрощує масштабування та підтримку системи [\[14\]](#).

Важливо: серверний компонент не агрегує дані з різних джерел у єдину відповідь. Кожен ендпоінт працює автономно, що забезпечує гнучкість для клієнта та спрощує масштабування системи. Структура серверного проєкту подана на рисунку 2.1.

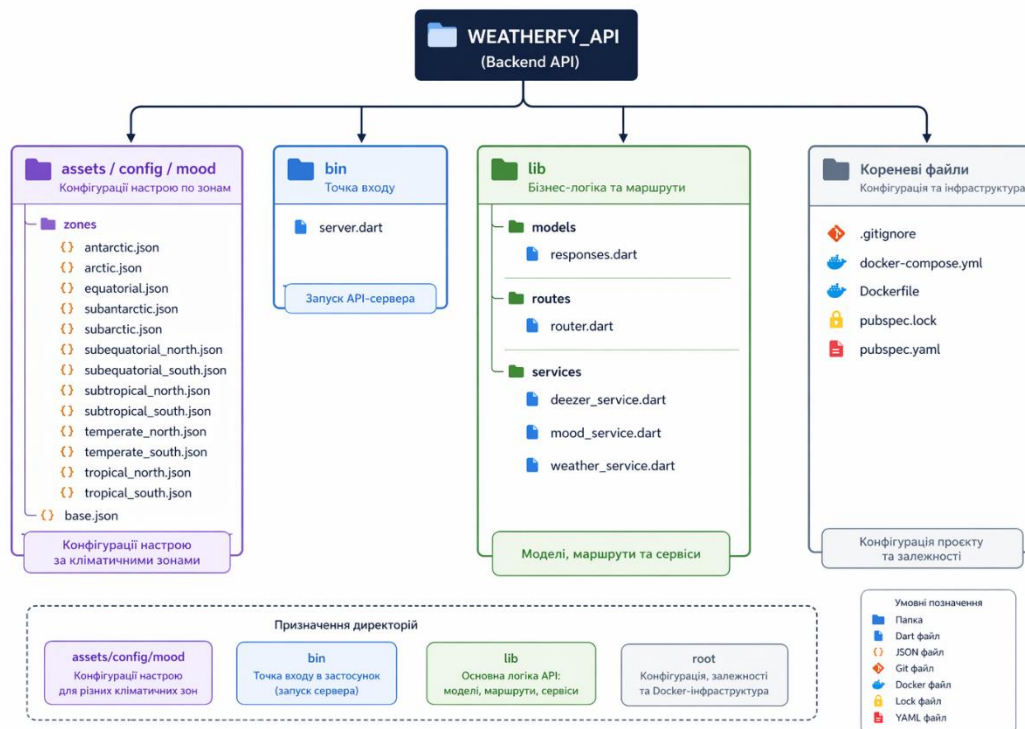


Рисунок 2.1. Структура серверного компоненту системи – WeatherfyAPI

Клієнтський застосунок реалізовано з використанням фреймворку Flutter та дотримується підходу Clean Architecture, запропонованого Robert C. Martin, де система забезпечує чітке розмежування відповідальності між шарами та мінімізацію залежностей між компонентами [5]. Структура клієнтського застосунку подана на рисунку А.2 (додаток А.2). Система побудована навколо декомповованої взаємодії з API: погода, настрій та музика обробляються через три незалежні ендпоінти (/weather, /mood, /music), а клієнт виступає оркестратором, який послідовно запитує контекст, кешує його та формує фінальний інтерфейс. Локальні немузичні рекомендації генеруються безпосередньо на пристрої користувача за допомогою вбудованого Recommendation Engine.

Шар CORE (Ядро системи та інфраструктура). Містить загальносистемні компоненти, конфігурації, утиліти та глобальні провайдери, що не належать до конкретної бізнес-домени, але необхідні для функціонування всіх шарів.

Директорія constants. Централізовані константи: API_constants.dart, asset_paths.dart (шляхи до ресурсів).

Директорія mood_preparatory_services. Підготовка даних для запиту настрою: mood_model.dart (структура емоційного стану), mood_provider.dart (управління станом через Riverpod).

Рушій recommendation_engine. Локальний рушій генерації порад:

- condition_evaluator.dart - аналіз комбінації погода + настроїв
- weather_patterns.dart - шаблони реакцій на кліматичні стани
- energy_profiles.dart - профілі інтенсивності активностей
- recommendation_loader.dart - формування фінального списку порад

Допоміжні утиліти utils:

– climate_zone_resolver.dart - визначення кліматичної зони за координатами

- disk_cache_manager.dart - управління файловим кешем
- date_formatter.dart - форматування дат
- input_validator.dart - валідація введених даних

Базові конфігурації theme (кольорові токени, типографіка).

Глобальні провайдери:

– deezer_provider.dart - інтеграція з музичним стрімінгом, кешування метаданих треків

– particle_settings_provider.dart - параметри візуальних ефектів (щільність, швидкість частинок)

– temp_unit_provider.dart - перемикач °C/°F

– theme_provider.dart - управління темами (світла/темна/сезонна)

Шар DATA (Джерела даних). Відповідає за комунікацію із зовнішніми API та локальне зберігання.

– Реалізації звернень до джерел data_sources:

– remote_weather_datasource.dart → GET /weather?lat&lon

- remote_mood_datasource.dart → POST /mood (передає weather, latitude, climate_zone)
 - remote_music_datasource.dart → GET /music?energy&valence
 - remote_geocoding_datasource.dart → Nominatim API
 - local_city_datasource.dart → збереження обраних міст
- models - DTO-класи з fromJson/toJson для мапінгу JSON-відповідей від трьох ендпоінтів.

Реалізації інтерфейсів repositories:

- weather_repository_impl.dart - об'єднує виклики /weather та /mood
- cached_weather_repository.dart - додає шар кешування з TTL
- city_repository_impl.dart - робота з локальним списком міст

Конфігурація локальних сховищ local_storage (SharedPreferences) для кешу погоди, налаштувань теми та списку міст.

Шар DOMAIN (Бізнес-логіка). Ізольований шар з чистою бізнес-логікою, незалежний від Flutter та HTTP.

Чисті бізнес-об'єкти entities: Weather, Mood, Track, LocalRecommendation, City, GeocodingResult.

Абстрактні інтерфейси repositories: WeatherRepository, MoodRepository, MusicRepository, CityRepository.

Атомарні сценарії використання usecases:

- FetchWeatherContextUseCase - послідовно викликає /weather → /mood
- FetchMusicRecommendationsUseCase - бере energy/valence з mood, запитує /music
- GenerateLocalTipsUseCase - передає weather+mood у RecommendationEngine
- ManageCityUseCases - додавання, пошук, видалення міст

Шар PRESENTATION (UI). Відображення даних та взаємодія з користувачем. Як зазначає Iglauer S., Flutter дозволяє реалізовувати сучасні

адаптивні інтерфейси з використанням декларативного підходу до побудови UI [9].

Основні екрани - screens:

– weather_screen/ - відображення метеоданих, запуск оркестрації запитів

– recommendation_screen/ - локальні поради (активності, одяг, режим)

– music_screen/ - список треків та плеєр демо-фрагментів

– city_list_screen/ - управління збереженими локаціями

– settings_screen/ - одиниці виміру, теми, параметри анімацій

Компоненти повторного використання - widgets: картки погоди, кнопки, індикатори завантаження, діалоги помилок.

Сезонні анімації - effects: leaf/ (осінь), snow/ (зима), spring/ (весна), summer/ (літо)

Riverpod-провайдери для реактивного стану - providers: weather_state_provider, mood_state_provider, music_tracks_provider, local_tips_provider.

Візуальні стилізації - themes: світла/темна палітри, сезонні варіації, налаштування контрастності.

Взаємодія модулів

1. Ініціалізація контексту:

weather_screen → FetchWeatherContextUseCase → /weather → підготовка mood payload (з climate_zone_resolver) → /mood.

2. Локальні поради:

Отримані weather+mood → GenerateLocalTipsUseCase → RecommendationEngine (офлайн) → recommendation_screen.

3. Музика:

music_screen → FetchMusicRecommendationsUseCase → /music?energy=&valence= → deezer_provider (демо-фрагменти).

4. Кешування:

disk_cache_manager зберігає останні відповіді. При офлайн-режимі cached_weather_repository повертає закешовані дані, RecommendationEngine продовжує працювати.

Така структура забезпечує модульність, тестопридатність та гнучкість при розширенні функціоналу.

Описана архітектура системи є основою для подальшого моделювання компонентів та їх взаємодії. У наступному підрозділі архітектурні рішення формалізуються засобами UML, що дозволяє наочно відобразити структуру шарів, напрямки залежностей та динаміку виконання ключових сценаріїв.

2.2. Моделювання системи засобами UML. Архітектурна діаграма компонентів

Для візуалізації архітектури, проєктування компонентів та документування взаємодії між модулями системи використано уніфіковану мову моделювання UML (Unified Modeling Language). Застосування UML дозволяє формалізувати вимоги до системи, виявити потенційні проблеми проєктування на ранніх етапах та забезпечити однозначне розуміння архітектури. У межах даного підрозділу розроблено чотири типи UML-діаграм: діаграму компонентів, діаграму варіантів використання, діаграму класів та діаграми послідовності. Кожна з них відображає окремий аспект системи - від статичної структури до динаміки виконання бізнес-сценаріїв.

На рисунку А.3 (додаток А.3), представлено діаграму компонентів клієнтського застосунку, що відображає ієрархічну структуру шарів та напрямки залежностей між ними. Діаграма демонструє реалізацію принципу Dependency Inversion: зовнішні шари (Presentation, Data) залежать від абстракцій внутрішнього шару (Domain), а не навпаки. Це забезпечує незалежність бізнес-логіки від фреймворку Flutter та зовнішніх джерел даних.

Основні аспекти, що відображені на діаграмі:

- Presentation Layer взаємодіє з Domain виключно через інтерфейси Use Cases;

- Data Layer реалізує абстрактні інтерфейси репозиторіїв з Domain;
- CORE надає спільні утиліти, константи та Recommendation Engine всім шарам через механізм впровадження залежностей;
- Пунктирні лінії позначають залежності від інфраструктурних компонентів CORE.

Архітектурна діаграма компонентів відображає статичну структуру системи (рис. 2.2). Для повного охоплення функціональних вимог та динаміки роботи додатку в наступних підрозділах розглядаються діаграма варіантів використання та діаграми послідовності.

2.2.1. Діаграма варіантів використання (Use Case)

На рисунку 2.2, на сторінці 34, представлено діаграму варіантів використання, що описує функціональні вимоги до системи з точки зору кінцевого користувача. Діаграма ідентифікує основного актора User (користувач) та чотири основні групи функціональності: City Management, Weather Display, Music Recommendations та Visual Settings.

Актор системи User - кінцевий користувач мобільного застосунку, який взаємодіє з системою через графічний інтерфейс для отримання персоналізованих рекомендацій на основі погодних умов.

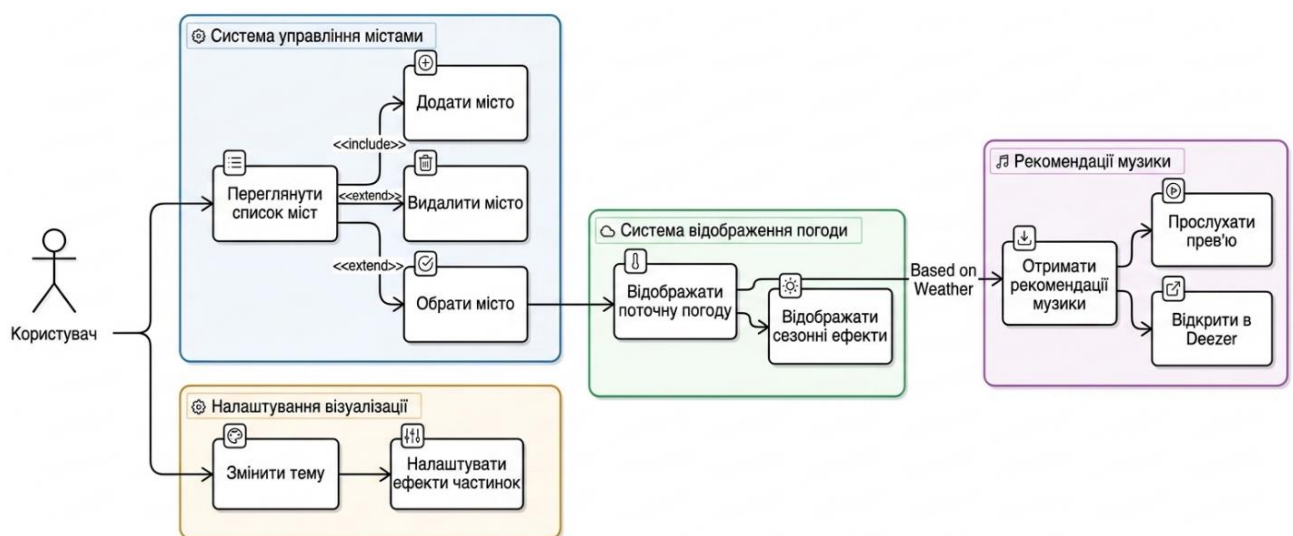


Рисунок 2.2 - Діаграма використання Use-Case мобільного додатку Weatherfy

Групи варіантів використання:

1. City Management (Управління містами). Цей модуль відповідає за роботу з географічними локаціями:

- View City List - перегляд списку збережених міст, що є точкою входу до всіх операцій управління локаціями;
- Add City - додавання нового міста до списку через пошук за назвою з використанням геокодингу Nominatim;
- Remove City - видалення обраного міста зі збереженого списку;
- Select City - вибір активного міста для перегляду погодних умов; цей use case є критичним, оскільки запускає ланцюжок отримання даних: після вибору міста система автоматично ініціює Display Weather Data.

2. Weather Display (Відображення погоди). Основний функціонал відображення метеорологічної інформації:

- Display Weather Data - отримання та візуалізація поточних погодних умов (температура, вологість, швидкість вітру, хмарність) для обраного міста; дані отримуються через послідовні запити до ендпоінтів /weather та /mood,
- Display Seasonal Effects - активація візуальних анімацій (сніг, опадаюче листя, весняні частинки, літнє сонце) на основі отриманих погодних даних та пори року; цей use case розширює функціонал відображення погоди, збагачуючи користувацький досвід.

3. Music Recommendations (Музичні рекомендації). Функціонал, що забезпечує персоналізовану музичну підбірку:

- Fetch Recommendations - отримання списку музичних треків від Deezer API через ендпоінт /music; запит формується на основі параметрів energy та valence, які обчислені з урахуванням поточних погодних умов (зв'язок "Based on Weather"),
- Play Review - відтворення 30-секундного демо-фрагменту треку безпосередньо в застосунку для попереднього прослуховування,

- Open in Deezer - перенаправлення користувача до повноцінного додатку Deezer для прослуховування повної версії треку; цей use case активується, якщо користувач зацікавився рекомендацією.

4. Visual Settings (Візуальні налаштування). Модуль кастомізації інтерфейсу:

- Change Theme - перемикання між темами оформлення (світла, темна, сезонні варіації Autumn/Winter/Spring/Summer),
- Adjust Particle Settings - налаштування параметрів анімаційних ефектів (щільність частинок, швидкість анімації, розмір елементів); цей use case розширює можливості Change Theme, дозволяючи тонке налаштування візуального досвіду.

Взаємозв'язки між use cases:

Діаграма чітко демонструє послідовність виконання сценаріїв:

1. Користувач обирає місто (Select City) → запускається відображення погоди (Display Weather Data),
2. Відображення погоди активує сезонні ефекти (Display Seasonal Effects),
3. На основі погодніх даних (Based on Weather) система автоматично формує запит музичних рекомендацій (Fetch Recommendations),
4. Користувач може переглянути список міст (View City List) та виконати будь-яку операцію управління (Add/Remove/Select),
5. Налаштування візуалізації (Change Theme) дозволяє додатково налаштувати параметри частинок (Adjust Particle Settings).

Варто зазначити, що частина варіантів використання виконується автоматично без прямої взаємодії користувача:

- Після Select City система самостійно ініціює ланцюжок Display Weather Data → Display Seasonal Effects → Fetch Recommendations,
- Recommendation Engine працює у фоновому режимі, аналізуючи комбінацію погодніх умов та параметрів настрою для генерації локальних порад

– Кешування даних та оновлення фонових процесів також відбуваються прозоро для користувача, забезпечуючи плавний користувацький досвід навіть при нестабільному мережевому з'єднанні.

Така структура use case діаграми забезпечує повне охоплення функціональних вимог до системи, чітко визначає межі відповідальності між різними модулями застосунку та демонструє інтеграцію між ручними діями користувача та автоматизованими процесами системи. Діаграма варіантів використання визначає що система повинна робити з точки зору користувача. Для розуміння як саме реалізуються ці сценарії на рівні взаємодії компонентів у часі, у наступному підрозділі розроблено діаграми послідовності для ключових бізнес-сценаріїв.

2.2.2. Діаграми послідовності (Sequence Diagram)

Для детальної візуалізації динаміки взаємодії між об'єктами системи під час виконання ключових бізнес-сценаріїв розроблено діаграми послідовності. Вони ілюструють потік викликів між шарами Clean Architecture (Presentation → Domain → Data), механізми кешування, обробки помилок та інтеграції з зовнішніми сервісами. Діаграми підтверджують, що клієнт виступає оркестратором запитів, а сервер надає ізольовані ендпоінти для погоди, настрою та музики. У межах підрозділу розроблено дві діаграми послідовності, що охоплюють найважливіші функціональні сценарії системи: отримання погодних даних з подальшим розрахунком музичних рекомендацій та пошук і додавання нового міста до списку відстежуваних. Вибір саме цих сценаріїв обумовлений їхньою архітектурною складністю: обидва задіюють усі чотири шари клієнта, серверний компонент та зовнішні API, а також включають умовні гілки виконання (кешування, обробка помилок, fallback-стратегії).

Діаграма отримання погодних даних демонструє дворівневий сценарій отримання контексту та персоналізованого контенту. Учасниками виступають компоненти клієнта (екрани, провайдери, Use Cases, репозиторії), серверний API (WeatherfyAPI) та зовнішні сервіси (OpenMeteo API, DeezerAPI). Процес поділено на два логічні ланцюжки (рис. 2.3).

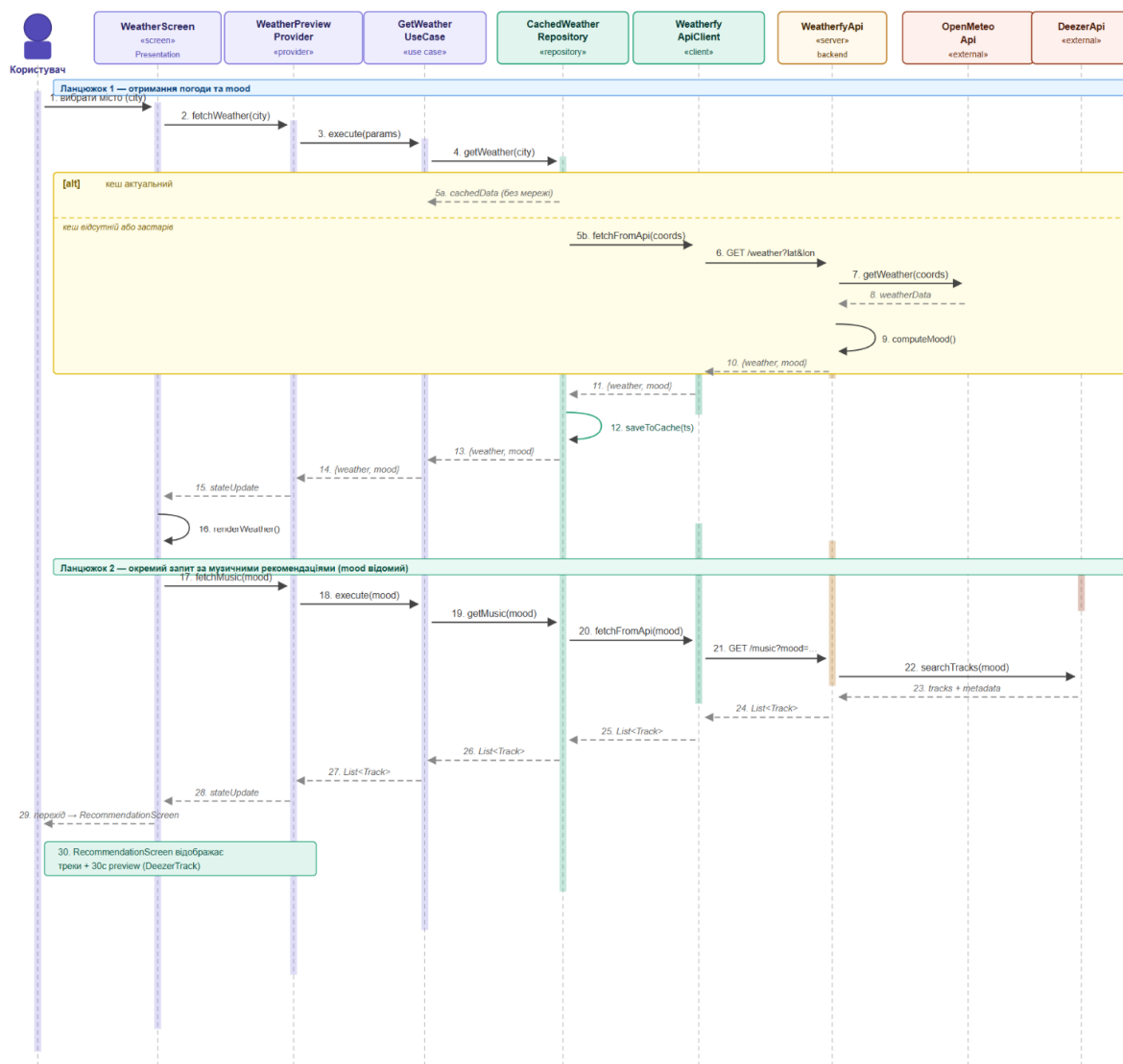


Рисунок 2.3. Отримання погодних даних та музичних рекомендацій

Ланцюжок 1: Отримання погоди та розрахунок настрою. Після вибору міста користувачем ініціюється запит через шар Presentation до GetWeather UseCase. Репозиторій CachedWeatherRepository спочатку перевіряє актуальність локального кешу (умовний фрагмент [alt]). Якщо дані відсутні або застарілі - спочатку перевіряється дисковий кеш (SharedPreferences), і лише за його відсутності або закінчення TTL виконується мережевий запит до сервера (GET /weather). Сервер звертається до Open-Meteo, отримує метеодані та розраховує параметри energy і valence через MoodService. Отриманий контекст {weather, mood} зберігається в кеші з міткою часу та повертається в UI, де оновлюється стан і рендериться інтерфейс погоди.

Ланцюжок 2: Отримання музичних рекомендацій. Після успішного розрахунку настрою система автоматично ініціює окремий запит `fetchMusic(energy, valence)`. Параметри передаються на ендпоінт `GET /music`, який визначає емоційний квадрант та делегує пошук треків до Deezer API - спочатку через жанрове радіо, у разі порожньої відповіді - через текстовий пошук (fallback). Отриманий `List<TrackResponse>` з метаданими та посиланнями на 30-секундні прев'ю повертається через усі шари до Presentation Layer. Після оновлення стану відбувається навігація на `RecommendationScreen`, де користувач може переглянути рекомендації та прослухати демо-фрагменти.

Діаграма підтверджує декомпозицію відповідальності: сервер виконує інтеграційну логіку (погода → настрої → музика), а клієнт керує станом, дворівневим кешуванням та навігацією, дотримуючись принципу розділених ендпоінтів.

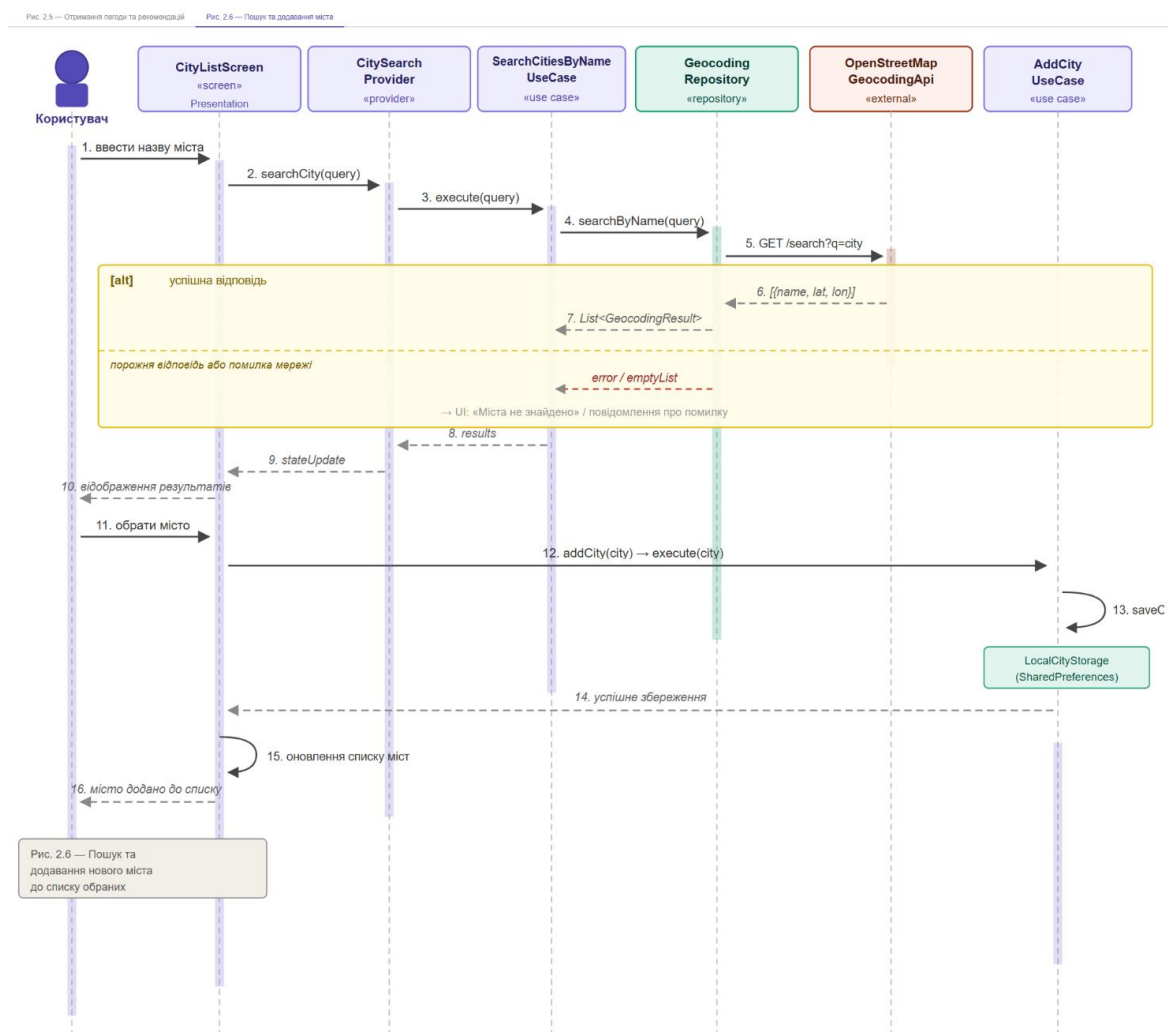


Рисунок 2.4. Пошук та додавання міста

Дана діаграма описує сценарій роботи з географічними локаціями, який є критичним для ініціалізації погодного контексту.

Користувач вводить назву міста, що запускає ланцюжок викликів від CityListScreen через CitySearch Provider до SearchCitiesByName UseCase. Репозиторій геокодингу надсилає запит до OpenStreetMap GeocodingAPI (GET /search?q=city). Діаграма містить умовний фрагмент [alt], що відображає обробку двох сценаріїв:

Успішна відповідь: API повертає масив об'єктів {name, lat, lon}, який трансформується в List<GeocodingResult> та відображається в інтерфейсі для вибору користувачем.

Помилка або порожній результат: Система повертає error / emptyList, після чого UI виводить відповідне повідомлення («Міста не знайдено» / повідомлення про помилку).

Після вибору міста зі списку активується AddCity UseCase, який відповідає за збереження локації в LocalCityStorage (SharedPreferences). Після успішного запису ініціюється оновлення списку міст на екрані та відображення підтвердження користувачеві.

Діаграма демонструє чітке розділення операцій пошуку (мережевий запит до зовнішнього API) та збереження (локальне сховище), а також коректну обробку виняткових ситуацій на рівні Data Layer.

Висновки щодо динамічного моделювання

Розроблені діаграми послідовності повністю узгоджуються з архітектурою системи та підтверджують:

1. Дотримання принципу оркестрації на клієнті: Послідовні запити до /weather, /mood та /music керуються логікою застосунку, а не серверною агрегацією.
2. Надійність та офлайн-підтримку: Використання фрагмента [alt] для перевірки кешу гарантує коректну роботу застосунку при відсутності мережі або тимчасових збоях API.

3. Чіткий потік даних через шари: Кожен запит проходить стандартизований шлях `Presentation → Provider → UseCase → Repository → DataSource/API`, що забезпечує тестопридатність та легкість підтримки коду.

Реактивність інтерфейсу: Оновлення стану (`stateUpdate`) відбувається лише після повного завершення бізнес-логіки, що унеможливлює відображення неповних або суперечливих даних.

2.3. Проектування структури мобільного додатку

На основі архітектурної моделі, описаної в підрозділі 2.1, та виконаного UML-моделювання (підрозділи 2.2–2.2.3) розроблено детальну структуру файлів та модулів мобільного додатку Weatherfy. Організація проєкту відповідає принципам чистої архітектури (Clean Architecture), які застосовуються в сучасній Android-розробці, забезпечуючи гнучкість та модульність програмного рішення [24].

Файлова структура проєкту є практичною реалізацією архітектурних рішень, описаних у підрозділі 2.1, та безпосередньо відповідає UML-моделям з підрозділу 2.2. У даному підрозділі наведено детальний опис модулів мобільного додатку Weatherfy із зазначенням призначення кожного компонента, його місця в ієрархії шарів Clean Architecture та взаємозв'язків з іншими частинами системи.

Опис фінальних ключових модулів

Модуль ядра (core/). Містить компоненти, що використовуються на всіх рівнях додатку та забезпечують інфраструктурну підтримку.

Constants. Централізоване зберігання незмінних параметрів: `API_constants.dart` визначає базові URL endpoint'ів (`/weather`, `/mood`, `/music`), таймаути з'єднання та ключі інтеграцій; `asset_paths.dart` містить шляхи до зображень, шрифтів, JSON-конфігурацій.

Mood Preparatory Services. Підготовка даних для запиту настрою: `mood_model.dart` описує структуру емоційного стану з параметрами `energy` та `valence`; `mood_provider.dart` керує станом настрою в реальному часі через `Riverpod` та синхронізує його з UI.

Recommendation Engine (Локальний рушій рекомендацій). Важливо: цей модуль відповідає за генерацію немuzичних рекомендацій (поради щодо активностей, одягу, режиму дня) на основі комбінації погодних умов та параметрів настрою. Музичні рекомендації отримуються окремо через API endpoint /music.

Компоненти Recommendation Engine:

- weather_patterns.dart - шаблони погодних умов (сонячно, дощово, хмарно, спекотно, холодно, вітряно, сніг)
- energy_profiles.dart - профілі енергії для активностей (висока активність, помірна, низька, релаксація)
- condition_evaluator.dart - оцінювач умов, що аналізує комбінацію weather + mood та визначає відповідний профіль
- recommendation_loader.dart - завантажувач рекомендацій, що формує фінальний список порад на основі активного патерну

Рушій рекомендацій працює повністю на стороні клієнта, що дозволяє:

- зменшити навантаження на сервер
- забезпечити миттєву генерацію порад без мережових затримок
- підтримувати функціонал в офлайн-режимі з використанням закешованих даних

Utils. Допоміжні утиліти: climate_zone_resolver.dart визначає кліматичну зону за координатами (критично для коректного розрахунку mood на сервері); disk_cache_manager.dart керує файловим кешем з TTL; date_formatter.dart та input_validator.dart забезпечують валідацію та форматування.

Theme. Базові конфігурації тем: color_palettes.dart (кольорові токени для світлої/темної/сезонних тем).

Глобальні провайдери:

- deezer_provider.dart - інтеграція з музичним стрімінгом, кешування метаданих треків, управління відтворенням 30-секундних демо-фрагментів
- particle_settings_provider.dart - параметри візуальних ефектів (щільність, швидкість, розмір частинок)

- `temp_unit_provider.dart` - перемикач одиниць температури (°C/°F) з автоматичною конвертацією

- `theme_provider.dart` - глобальне управління темами (ручне перемикання, автоматична сезонна зміна)

Шар даних (data/). Відповідає за комунікацію із зовнішніми джерелами, серіалізацію та локальне зберігання.

Data Sources. Конкретні реалізації звернень:

- `remote_weather_datasource.dart` → GET `/weather?lat&lon`
- `remote_mood_datasource.dart` → POST `/mood` (передає `weather`, `latitude`, `climate_zone`)
- `remote_music_datasource.dart` → GET `/music?energy&valence`
- `remote_geocoding_datasource.dart` → Nominatim API
- `local_city_datasource.dart` → збереження списку міст

Models. DTO-класи з `fromJson/toJson` для точного мапінгу JSON-відповідей від трьох ендпоінтів у `typed`-об'єкти Dart.

Repositories. Реалізації інтерфейсів:

- `weather_repository_impl.dart` - оркеструє виклики `/weather` → `/mood`
- `cached_weather_repository.dart` - додає шар кешування з перевіркою актуальності
- `city_repository_impl.dart` - робота з локальним списком міст

Local Storage. Конфігурація локальних сховищ (`SharedPreferences/Hive`) для кешу погоди, налаштувань теми, списку міст та параметрів користувача.

Шар домену (domain/). Ізольований шар з чистою бізнес-логікою, незалежний від Flutter та HTTP.

Entities. Чисті бізнес-об'єкти: `Weather`, `Mood`, `Track`, `LocalRecommendation`, `City`, `GeocodingResult` - містять лише поля та базові методи.

Repositories. Абстрактні інтерфейси: `WeatherRepository`, `MoodRepository`, `MusicRepository`, `CityRepository` - визначають контракти для реалізацій з шару `Data`.

Use Cases. Атомарні сценарії використання:

- FetchWeatherContextUseCase - послідовно викликає /weather → /mood
- FetchMusicRecommendationsUseCase - бере energy/valence з mood, запитує /music
- GenerateLocalTipsUseCase - передає weather+mood у RecommendationEngine
- AddCityUseCase, SearchCitiesByNameUseCase, RemoveCityUseCase - управління містами

Шар презентації (presentation/). Відображення даних та взаємодія з користувачем.

Screens. Основні екрани:

- weather_screen/ - відображення метеоданих, запуск оркестрації запитів
- recommendation_screen/ - локальні поради від Recommendation Engine
- music_screen/ - список треків та плеєр демо-фрагментів
- city_list_screen/ - управління збереженими локаціями
- settings_screen/ - одиниці виміру, теми, параметри анімацій

Widgets: картки погоди, індикатори настрою, картки треків, індикатори завантаження, віджети помилок.

Effects. Сезонні анімаційні модулі:

- leaf/ (осінь) - ефект опадаючого листа
- snow/ (зима) - снігопад
- spring/ (весна) - квіткові частинки
- summer/ (літо) - сонячні промені

Кожен ефект має власну систему частинок з налаштуваннями щільності, швидкості та розміру через particle_settings_provider.

Providers. Riverpod-провайдери для реактивного стану: weather_state_provider, mood_state_provider, music_tracks_provider, local_tips_provider, app_loading_provider.

Themes. Візуальні стилізації: `light_theme.dart`, `dark_theme.dart`, сезонні варіації (`autumn_theme.dart`, `winter_theme.dart`, `spring_theme.dart`, `summer_theme.dart`).

Принципи взаємодії модулів

Взаємодія між компонентами підпорядковується принципу інверсії залежностей (Dependency Inversion Principle): шар домену не має прямих залежностей від шару даних або презентації. Зв'язки між шарами реалізуються через механізм впровадження залежностей (Dependency Injection) за допомогою провайдерів Riverpod.

Ключові потоки даних:

1. Отримання погодного контексту: `WeatherScreen` → `FetchWeatherContextUseCase` → `/weather` → підготовка `payload` → `/mood` → кешування → оновлення UI
2. Генерація локальних порад: `GenerateLocalTipsUseCase` → `RecommendationEngine` (`condition_evaluator` → `weather_patterns` → `energy_profiles` → `recommendation_loader`) → `local_tips_provider` → `RecommendationScreen`
3. Отримання музики: `MusicScreen` → `FetchMusicRecommendationsUseCase` → `/music?energy=&valence=` → `deezer_provider` → відтворення демо
4. Управління містами: `CityListScreen` → `SearchCitiesByNameUseCase` → `Nominatim API` → `AddCityUseCase` → `LocalCityStorage`

Запропонована файлова структура повністю відповідає архітектурній моделі, описаній у підрозділі 2.1, та UML-діаграмам з підрозділу 2.2. Ключовою особливістю є чітке розмежування між локальним `Recommendation Engine` (немузичні поради) та музичними рекомендаціями від API, що забезпечує оптимальний розподіл навантаження між клієнтом та сервером.

Спроектowana файлова структура повністю готова до реалізації. Наступним кроком є детальний опис взаємодії серверного компонента `Weatherfy_API` із зовнішніми сервісами - API Open-Meteo та Deezer, - що розглядається в підрозділі 2.4.

2.4. Опис взаємодії з API Open-Meteo та Deezer

Для забезпечення функціонування системи Weatherfy необхідно реалізувати взаємодію серверного компонента Weatherfy_API із двома зовнішніми API: метеорологічним сервісом Open-Meteo та музичним сервісом Deezer. Open-Meteo виступає джерелом актуальних метеорологічних даних, необхідних для аналізу погодних умов та обчислення показника «настрою». Deezer надає доступ до каталогу музичних композицій, з якого формуються рекомендації для користувача.

У даному підрозділі подається опис архітектури серверного компонента Weatherfy_API, форматів запитів та відповідей для кожного зовнішнього API, а також наводяться приклади реалізації відповідних сервісів.

Архітектура серверного компонента Weatherfy_API. Серверний компонент Weatherfy_API реалізовано з використанням мови Dart.

- `bin/server.dart` - точка входу, основний файл запуску сервера, який ініціалізує HTTP-сервер та підключає маршрутизацію.
- `lib/routes/router.dart` - маршрутизація HTTP-запитів, визначення ендпоінтів та прив'язка до відповідних обробників.
- `lib/models/responses.dart` - моделі даних для формування JSON-відповідей (структури погодних даних, музичних рекомендацій та агрегованих відповідей).
- `lib/services/weather_service.dart` - сервіс інтеграції з API Open-Meteo (виконання запитів, отримання метеоданих).
- `lib/services/mood_service.dart` - сервіс аналізу погодних даних та обчислення показника настрою з урахуванням кліматичних зон.
- `lib/services/deezer_service.dart` - сервіс інтеграції з API Deezer (пошук треків, отримання метаданих та демо-фрагментів).

Конфігурація кліматичних зон (mood zones). Особливістю реалізації `mood_service.dart` є використання конфігураційних файлів для визначення порогових значень метеопараметрів залежно від географічного положення. У

директорії assets/config/mood/zones/ розміщено JSON-файли, що описують кліматичні зони:

Таблиця 2.2 - Опис файлів конфігурації WeatherfyAPI

Файл	Кліматична зона	Географічне положення
equatorial.json	Екваторіальна	0°–10° широти
tropical_north.json	Тропічна (північна)	10°–25° пн.ш.
tropical_south.json	Тропічна (південна)	10°–25° пд.ш.
subtropical_north.json	Субтропічна (північна)	25°–40° пн.ш.
subtropical_south.json	Субтропічна (південна)	25°–40° пд.ш.
temperate_north.json	Помірна (північна)	40°–60° пн.ш.
temperate_south.json	Помірна (південна)	40°–60° пд.ш.
subarctic.json	Субарктична	60°–70° пн.ш.
subantarctic.json	Субантарктична	60°–70° пд.ш.
arctic.json	Арктична	>70° пн.ш.
antarctic.json	Антарктична	>70° пд.ш.
subequatorial_north.json	Субекваторіальна (північна)	перехідна зона
subequatorial_south.json	Субекваторіальна (південна)	перехідна зона
base.json	Базова конфігурація (за замовчуванням)	-

Кожен конфігураційний файл містить порогові значення для визначення показника настрою, зокрема:

- діапазони температур для категорій «спекотно», «тепло», «комфортно», «прохолодно», «холодно»;
 - порогові значення опадів для визначення дощової погоди;
 - порогові значення швидкості вітру для визначення вітряної погоди.
- Діляться на чотири категорії: calm / breezy / windy / storm.

Такий підхід дозволяє адаптувати рекомендації до кліматичних особливостей різних регіонів: наприклад, температура +20°C вважатиметься «спекотною» для субарктичної зони, але «комфортною» для екваторіальної.

Взаємодія з API Open-Meteo. Open-Meteo є безкоштовним метеорологічним API, який надає доступ до даних провідних глобальних моделей прогнозування (ECMWF, GFS, ICON). Запит до API Open-Meteo формується як HTTP GET-запит до наступного ендпоінту:

GET <https://API.open-meteo.com/v1/forecast>

Параметри запиту передаються у рядку запиту (query string) у форматі key=value.

Таблиця 2.3 - Основні параметри запиту на API Open-Meteo

Параметр	Тип	Опис	Приклад
latitude	float	Широта місцезнаходження	50.4501
longitude	float	Довгота місцезнаходження	30.5234
current_weather	boolean	Отримання поточної погоди	true
hourly	string	Перелік погодинних змінних	temperature_2m,relative_humidity_2m,precipitation
daily	string	Перелік щоденних змінних	temperature_2m_max,temperature_2m_min
timezone	string	Часовий пояс	Europe/Kyiv

Приклад повного запиту:

https://API.open-meteo.com/v1/forecast?latitude=50.4501&longitude=30.5234¤t_weather=true&hourly=temperature_2m,relative_humidity_2m,precipitation,windspeed_10m&timezone=Europe/Kyiv

Відповідь API Open-Meteo повертається у форматі JSON. Формат JSON є одним із найбільш поширених стандартів обміну структурованими даними у веб-застосунках завдяки своїй компактності та простоті обробки [15]. Структуру відповіді подано на рисунку Б.1 (додаток Б). Основні поля відповіді включають:

- latitude, longitude - координати, для яких виконано запит;
- current_weather - об'єкт з поточними метеоданими:
- temperature - температура повітря (°C);
- windspeed - швидкість вітру (км/год);
- winddirection - напрям вітру (градуси);
- weathercode - код погодних умов (відповідно до стандарту WMO);
- hourly - об'єкт з погодинними даними (масиви значень для кожної змінної);
- daily - об'єкт з щоденними даними.

Приклад JSON-відповіді:

```
{
  "latitude": 50.4501,
  "longitude": 30.5234,
```

```

    "current_weather": {
      "temperature": 18.5,
      "windspeed": 12.3,
      "winddirection": 145,
      "weathercode": 3
    },
    "hourly": {
      "time": ["2025-04-21T00:00", "2025-04-21T01:00", ...],
      "temperature_2m": [12.3, 11.8, ...],
      "relative_humidity_2m": [65, 67, ...],
      "precipitation": [0.0, 0.0, ...]
    }
  }
}

```

У складі Weatherfy_API взаємодію з Open-Meteo реалізовано в сервісі WeatherService. Діаграму послідовності викликів для отримання погодних даних наведено на рисунку Б.2 (додаток Б). Основний метод сервісу виконує наступні дії:

1. Приймає географічні координати (широту та довготу) як вхідні параметри;
2. Формує URL із необхідним набором параметрів (поточна погода, погодинні змінні);
3. Виконує асинхронний HTTP GET-запит;
4. Отримує JSON-відповідь та десеріалізує її у внутрішню структуру даних;
5. Повертає структуровані метеорологічні дані для подальшого аналізу.

Взаємодія з API Deezer. Deezer надає публічний API для пошуку музичних композицій, отримання метаданих та доступу до 30-секундних демо-фрагментів. У межах даної роботи використовується пошуковий ендпоінт Deezer Search API.

Запит до API Deezer формується як HTTP GET-запит до наступного ендпоінту: GET <https://API.deezer.com/search>

Основним параметром запиту є:

Параметр	Тип	Опис	Приклад
q	string	Пошуковий запит (жанр, виконавець)	"pop sunny"

Додатково можуть використовуватися параметри пагінації:

- limit - кількість результатів на сторінку (максимум 50);
- index - зміщення для посторінкового виведення.

Приклад повного запиту:

`https://API.deezer.com/search?q=pop%20sunny&limit=10`

Відповідь API Deezer повертається у форматі JSON. Структуру відповіді подано на рисунку Б.3 (додаток Б). Основні поля відповіді включають:

- data - масив треків, що відповідають пошуковому запиту;
- total - загальна кількість знайдених результатів;
- next - посилання на наступну сторінку результатів.

Кожен елемент масиву data містить наступну інформацію про трек:

- id - унікальний ідентифікатор треку;
- title - назва композиції;
- duration - тривалість у секундах;
- preview - пряме посилання на 30-секундний демо-фрагмент в MP3;
- link - гіперпосилання на сторінку треку на веб-сайті Deezer;
- artist - об'єкт з інформацією про виконавця (ім'я, посилання);
- album - об'єкт з інформацією про альбом (назва, обкладинка).

Приклад JSON-відповіді:

```
{
  "data": [
    {
      "id": 123456789,
      "title": "Sunny Day",
      "duration": 210,
      "preview": "https://cdns-preview.deezer.com/.../preview.mp3",
      "link": "https://www.deezer.com/track/123456789",
      "artist": {
        "name": "Example Artist",
        "link": "https://www.deezer.com/artist/123"
      },
      "album": {
        "title": "Summer Hits",
        "cover_medium": "https://API.deezer.com/album/123/image"
      }
    }
  ],
  "total": 150,
  "next": "https://API.deezer.com/search?q=pop%20sunny&index=10"
}
```

Інтеграція в серверному компоненті. У складі Weatherfy_API взаємодію з Deezer реалізовано в сервісі DeezerService. Діаграму послідовності викликів для отримання музичних рекомендацій наведено на рисунку Б.4 (додаток Б). Основний метод сервісу виконує наступні дії:

1. Приймає пошуковий запит (сформований на основі показника настрою) як вхідний параметр;
2. Формує URL із закодованим пошуковим рядком;
3. Виконує асинхронний HTTP GET-запит;
4. Отримує JSON-відповідь та десеріалізує її у внутрішню структуру даних;
5. Повертає список треків з метаданими (назва, виконавець, посилання на демо-фрагмент, гіперпосилання).

Серверний компонент Weatherfy_API виконує агрегацію даних, отриманих від Open-Meteo та Deezer, у єдину JSON-відповідь для клієнтського застосунку. Схему агрегації даних подано на рис. 2.12 (наводиться в графічній частині роботи).

Процес агрегації включає наступні кроки:

1. Отримання метеорологічних даних від WeatherService;
2. Визначення кліматичної зони за географічними координатами (на основі конфігураційних файлів у assets/config/mood/zones/);
3. Обчислення показника настрою на основі аналізу температури, опадів, швидкості вітру та хмарності з урахуванням порогових значень відповідної кліматичної зони (виконується MoodService);
4. На основі отриманих energy та valence сервер визначає емоційний квадрант (high_energy_positive / high_energy_negative / low_energy_positive / low_energy_negative) та виконує запит до Deezer API - спочатку за жанровим радіо, у разі невдачі - за текстовим пошуком. Клієнт окремо запитує /music?energy=...&valence=... та отримує готовий список треків.

5. Об'єднання погодних даних (температура, вологість, швидкість вітру, умови, визначена кліматична зона) та музичних рекомендацій (список треків) у фінальний JSON-об'єкт;

6. Повернення агрегованої відповіді клієнтському застосунку.

Таким чином, серверний компонент Weatherfy_API реалізує повний цикл обробки даних: від отримання метеорологічної інформації через Open-Meteo до формування персоналізованої музичної добірки через Deezer з урахуванням кліматичної зони користувача. Описана інтеграція є технічною основою для функціоналу рекомендацій, реалізація якого розглядається в наступних розділах роботи.

2.5. Висновки до розділу 2

У другому розділі дипломної роботи виконано проектування мобільного додатку Weatherfy та серверного компонента Weatherfy_API. На основі аналізу вимог, виконаного в першому розділі, розроблено архітектуру системи, виконано моделювання засобами UML, спроектовано структуру модулів та описано взаємодію із зовнішніми API.

Загальна архітектура системи (підрозділ 2.1). Система побудована за клієнт-серверною архітектурою. Серверний компонент Weatherfy_API реалізує бізнес-логіку: інтеграцію з Open-Meteo API для отримання метеоданих, обчислення показника настрою з урахуванням кліматичних зон, інтеграцію з Deezer API для отримання музичних рекомендацій та надання агрегованих даних через власний RESTful API. Клієнтський компонент Weatherfy виконує визначення геолокації (через ручне введення міста з геокодингом Nominatim), комунікацію з серверним API, кешування даних, візуалізацію погодних умов із сезонними анімаціями та відображення музичних рекомендацій.

Моделювання системи засобами UML (підрозділ 2.2). Розроблено діаграму компонентів (рис. 2.3), що відображає розподіл на чотири логічні шари (presentation, domain, data, core) та напрямки залежностей згідно з принципами Clean Architecture. Діаграма варіантів використання (рис. 2.4) описує чотири

основні функціональні блоки: керування містами, відображення погоди, музичні рекомендації та візуальні налаштування. Діаграми послідовності (рис. 2.5 та рис. 2.6) візуалізують динамічну поведінку системи для сценаріїв «Отримання погоди та музичних рекомендацій» та «Пошук та додавання нового міста» з урахуванням альтернативних потоків (кешування, обробка помилок). Детальну файлову структуру клієнтського застосунку наведено в додатку А.

Проектування структури мобільного додатку (підрозділ 2.3). Розроблено файлову структуру проєкту, що відповідає Clean Architecture. Модуль ядра (core/) містить рушій рекомендацій (recommendation_engine/) - компоненти для оцінки погодних умов, визначення енергетичних профілів та завантаження рекомендацій, які працюють повністю на стороні клієнта. Візуальні ефекти організовані за сезонним принципом (effects/leaf, snow, spring, summer) та інтегровані з темами оформлення.

Опис взаємодії з API та архітектура серверного компонента (підрозділ 2.4). Детально описано модульну структуру Weatherfy_API: точка входу (bin/server.dart), маршрутизація (lib/routes/router.dart), моделі відповідей (lib/models/responses.dart) та сервіси (WeatherService, MoodService, DeezerService). Особливістю реалізації є конфігураційна система кліматичних зон (assets/config/mood/zones/), яка містить 14 JSON-файлів з пороговими значеннями метеопараметрів для різних географічних поясів (від екваторіального до арктичного). Це дозволяє адаптувати обчислення показника настрою до регіональних особливостей. Наведено формати запитів та відповідей для Open-Meteo API (ендпоїнт /v1/forecast, параметри latitude, longitude, current_weather, hourly) та Deezer API (ендпоїнт /search, параметр q). Описано процес агрегації даних у фінальну JSON-відповідь для клієнтського застосунку.

Сукупність виконаних проєктних робіт - від визначення архітектурних принципів до детального опису API-інтеграцій - формує цілісну та несутеречливу технічну документацію системи Weatherfy. Усі прийняті рішення є взаємоузгодженими: архітектура клієнт-сервер обґрунтовує розподіл

відповідальності, UML-діаграми формалізують цей розподіл, файлова структура реалізує його на практиці, а опис API-взаємодії завершує картину інформаційних потоків. Отримані результати створюють повну технічну основу для переходу до третього розділу, в якому розглядається безпосередня реалізація, тестування та аналіз якості програмного забезпечення.

РОЗДІЛ 3.

РЕАЛІЗАЦІЯ ТА АНАЛІЗ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1. Реалізація мобільного додатку

Реалізацію мобільного додатку Weatherfy виконано з використанням фреймворку Flutter (Dart) та архітектурного підходу Clean Architecture, описаного в розділі 2. Усі компоненти розподілені по чотирьох шарах: core, data, domain та presentation. Управління станом реалізовано через бібліотеку Riverpod, що забезпечує реактивну та декларативну модель взаємодії між шарами. У роботі Козуб Г. О. та Козуб Ю. Г. зазначається, що такий декларативний підхід суттєво спрощує розробку мультиплатформних застосунків та покращує підтримуваність інтерфейсу [\[20\]](#). Точкою входу є функція main(), яка ініціалізує Flutter-binding та обгортає кореневий віджет у ProviderScope - глобальний контейнер залежностей Riverpod.

Таблиця 3.1 - Основні залежності проєкту (pubspec.yaml)

Пакет	Призначення
flutter_riverpod	Управління станом (провайдери, нотифайєри)
http	HTTP-запити до серверного API
shared_preferences	Локальне сховище для кешу та налаштувань
connectivity_plus	Перевірка наявності мережевого з'єднання
weather_icons	Іконки погодних умов (WMO-стандарт)
just_audio	Відтворення 30-секундних музичних прев'ю

Шар Data: джерела даних та репозиторії

Шар data відповідає за комунікацію із зовнішніми API та локальне зберігання. Основним джерелом даних є RemoteWeatherDataSourceImpl, який формує GET-запит до серверного ендпоінту /weather з передачею координат:

```
final uri = Uri.parse('${APIConstants.backendBaseUrl}/weather')
  .replace(queryParameters: {
    'lat': lat.toString(),
    'lon': lon.toString(),
  });
final response = await http.get(uri);
if (response.statusCode == 200) {

  return WeatherModel.fromBackendJson(json.decode(response.body));}
```

Адреса серверного компонента (backendBaseUrl) задається через змінну оточення BACKEND_URL під час компіляції, що дозволяє перемикає середовища без змін у коді:

```
static const String backendBaseUrl = String.fromEnvironment(
  'BACKEND_URL',
  defaultValue: 'http://43.157.5.90:8080',);
```

Для отримання показників настрою moodProvider виконує POST-запит до ендпоінту /mood, передаючи метеодані, широту та кліматичну зону:

```
final response = await http.post(uri,
  headers: {'Content-Type': 'application/json'},
  body: jsonEncode({
    'weather': { 'temp': weather.temp, 'humidity': weather.humidity, ... },
    'latitude': homeCity.lat,
    'climate_zone': getClimateZone(homeCity.lat),
  }),
);
```

Функція getClimateZone() визначає кліматичну зону за широтою без звернення до сервера - це локальна утиліта, що реалізує порогову класифікацію з 13 зон (від арктичної до антарктичної).

Дворівневий кеш: memory + disk

Одним із ключових технічних рішень є дворівнева система кешування, що забезпечує роботу додатку в офлайн-режимі. Реалізовано два незалежні компоненти: DiskCacheManager (постійне сховище на базі SharedPreferences) та CachedWeatherRepository (in-memory кеш поверх дискового).

Єдиний TTL для обох рівнів - 10 хвилин (DiskCacheManager.cacheDuration). CachedWeatherRepository реалізує чотирирівневу логіку отримання даних:

```
// 1. Memory cache - найшвидший доступ
final memoryEntry = _memoryCache[cityId];
if (memoryEntry != null && !memoryEntry.isExpired) return
memoryEntry.weather;

// 2. Disk cache - актуальний (< 10 хв)
final diskResult = await _diskCache.getCachedWeather(cityId);
if (diskResult != null && diskResult.isFresh) {
  _memoryCache[cityId] = CacheEntry(diskWeather, diskWeather.fetchedAt);
  return diskWeather;
}

// 3. Мережевий запит - якщо обидва кеші застарілі
final weather = await _remoteRepository.getWeatherByCoords(...);
_memoryCache[cityId] = CacheEntry(weather, weather.fetchedAt);
```

```

await _diskCache.saveWeather(cityId, weather);
return weather;

// 4. Офлайн-fallback – повертаємо застарілий disk кеш при відсутності мережі
if (diskWeather != null) return diskWeather;

```

Важливою особливістю є те, що застарілий кеш *ніколи не видається автоматично* - він зберігається як резервні дані на випадок відсутності мережі. Видалення відбувається лише за явним запитом користувача через екран налаштувань. Повну реалізацію DiskCacheManager наведено в Додатку Д.

Автоматичне оновлення та обробка помилок

WeatherNotifier (клас шару presentation) реалізує автоматичне фонове оновлення даних через глобальний пульс weatherSyncPulseProvider, що тикає щохвилини. При кожному тiku перевіряється стан кешу - якщо він прострочений, запускається оновлення. При помилках застосовується **експоненційний відкат**: кількість хвилин очікування до наступної спроби дорівнює лічильнику помилок (до максимуму 15 хвилин). Повну реалізацію WeatherNotifier з логікою автоматичного оновлення наведено в Додатку Е.

При виявленні відсутності мережі (SocketException) додатково запускається окремий retry-таймер з інтервалом 15 секунд для швидшого відновлення з'єднання. Якщо в стані вже є дані - UI не переходить у стан завантаження, а показує наявні дані («silent error mode»):

```

} catch (error, stack) {
    _errorCount++;
    if (error is SocketException) _startRetryTimer(); // retry кожні 15 сек
    if (state.hasValue) return state.value!;           // silent error mode
    Error.throwWithStackTrace(error, stack);          // перший запуск -
показуємо помилку
}

```

Шар Domain: сутності та use cases

Шар domain містить чисті бізнес-об'єкти та абстрактні інтерфейси, незалежні від Flutter і HTTP. Центральною сутністю є клас Weather, що включає поточні метеодані, погодинний прогноз (List<HourlyForecast>), а також метадані: час отримання (fetcheAt), схід/захід сонця, іконку та опис стану за WMO-

кодом. Use cases є атомарними сценаріями, кожен з яких інкапсулює одну операцію:

Таблиця 3.2 – Атомарні сценарії Use Cases

Use Case	Призначення
GetWeatherUseCase	Отримання погоди за координатами міста
AddCityUseCase	Збереження нового міста в локальне сховище
SearchCitiesByNameUseCase	Геокодинг через Nominatim API
RemoveCityUseCase	Видалення міста зі списку
GetCitiesUseCase	Отримання списку збережених міст

Шар Presentation: навігація та стан

Навігація між екранами реалізована через AppShell - кореневий віджет з нижньою навігаційною панеллю. Основні екрани (WeatherScreen, RecommendationScreen, MusicScreen, CityListScreen, SettingsScreen) реалізовані як ConsumerWidget - віджети, що підписуються на Riverpod-провайдери та автоматично перебудовуються при зміні стану [6].

Екран погоди WeatherScreen є оркестратором: після вибору міста він ініціює ланцюжок weatherNotifierProvider → moodProvider, результати якого передаються до RecommendationScreen та MusicScreen. Кожен провайдер є незалежним і кешується в ProviderScope протягом сесії.

Система теплих та сезонних тем

Візуальне оформлення додатку реалізоване через єдину абстракцію AppColorPalette - абстрактний клас, що визначає контракт з 12 кольоровими токенами: фони (scaffoldBackground, surface, cardBackground), текст (textPrimary, textSecondary, textDisabled), елементи інтерфейсу (inputBackground, divider, iconColor) та акцентні кольори (primary, secondary, error). Кожна тема є конкретною реалізацією цього класу.

Перелік тем визначено в enum AppTheme і включає шість варіантів: light, dark, winter, spring, summer, autumn. Клас ThemeBuilder перетворює будь-яку реалізацію AppColorPalette в повноцінний об'єкт ThemeData Material 3, заповнюючи всі необхідні компоненти (AppBar, Card, Input, FAB, Chip, Switch, Checkbox тощо) з палітри. Завдяки цьому додавання нової теми зводиться до створення одного класу-палітри без змін у ThemeBuilder.

Сезонні палітри мають чітко виражені кольорові характеристики, наведені в таблиці 3.3.

Таблиця 3.3 - Характеристики сезонних тем

Тема	scaffoldBackground	primary	Характеристика
winter	#C6E8E8 (льодяно-бірюзовий)	#0288D1 (блакитний)	Холодна, стримана
spring	#F1F8E9 (ніжно-зелений)	Colors.green.500	Свіжа, природна
summer	#FFF8E1 (сонячно-жовтий)	Colors.orange.600	Тепла, енергійна
autumn	#EFE9E9 (теплий бежевий)	Colors.deepOrange.700	Затишна, приглушена

Вибрана тема зберігається між сесіями через ThemeNotifier (StateNotifierProvider) у SharedPreferences за ключем selected_theme. При запуску додатку ThemeNotifier асинхронно завантажує збережене значення та ініціалізує стан, що виключає «мигання» теми при повторному відкритті.

Для збагачення візуального досвіду реалізовано чотири сезонні анімаційні ефекти: осінні листя (autumn_leaves_overlay.dart), сніг (snow_overlay.dart), весняні пелюстки (spring_leaves_overlay.dart) та літні частинки (summer_overlay.dart). Кожен ефект реалізований як CustomPainter зі власною системою частинок і підтримує налаштування щільності та швидкості через particle_settings_provider.

Сезонні анімаційні ефекти

Кожна сезонна тема супроводжується відповідним анімаційним оверлеєм, що накладається поверх основного контенту. Усі чотири ефекти реалізовані за єдиним архітектурним патерном: ConsumerStatefulWidget з SingleTickerProviderStateMixin для анімації через Ticker, та CustomPainter для відмальовування частинок на Canvas.

Структура кожного ефекту:

– **Overlay-клас** (SnowOverlay, AutumnLeavesOverlay, SpringLeavesOverlay, SummerOverlay) - керує станом списку частинок, підписується на particleSettingsProvider та запускає Ticker, що перерисовує сцену щокcadру.

- **Particle-клас** (Snowflake, Leaf тощо) - модель даних однієї частинки: координати, розмір, швидкість, фаза та амплітуда коливань.
- **Painter-клас** (SnowPainter, LeafPainter тощо) - реалізує CustomPainter.paint(), де для кожної частинки обчислюється нова позиція та виконується малювання.

Ключова особливість реалізації: список частинок повністю відтворюється при зміні налаштувань або розміру екрану. Це досягається порівнянням поточних ParticleSettings та Size з попередніми значеннями безпосередньо у методі build():

```
if (_lastSettings != settings || _lastSize != size) {
  _lastSettings = settings;
  _lastSize = size;
  _recreateFlakes(settings, size); // повне перестворення списку
}
```

Оверлей обгорнутий у IgnorePointer, що дозволяє анімації відображатися поверх усього контенту, не перехоплюючи дотики користувача. Параметри анімації - кількість частинок (count), розмір (size) та швидкість (speed) - керуються глобально через particleSettingsProvider і налаштовуються користувачем в екрані Settings.

Рушій рекомендацій (Recommendation Engine)

Recommendation Engine є найбільш нетривіальним компонентом клієнтської частини. Він реалізований повністю на стороні клієнта, не потребує мережевих запитів і генерує персоналізовані немюзичні рекомендації (поради щодо активностей, одягу, режиму дня) на основі комбінації погодних даних та показників настрою (energy, valence), отриманих від сервера. Повну реалізацію RecommendationEngine наведено в Додатку В

Ініціалізація та конфігурація на основі JSON-активів. Рушій завантажує свою конфігурацію з трьох JSON-файлів, розміщених у директорії assets/recommendations/:

Таблиця 3.4 – Зміст конфігураційних JSON-файлів

Файл	Вміст
index.json + файли рекомендацій	Список RecommendationConfig - правила з умовами та текстами
conditions/energy_profiles.json	EnergyProfile - профілі активності за рівнем енергії
conditions/weather_patterns.json	WeatherPattern - погодні патерни з пріоритетами та множниками

Завантаження виконується одноразово через RecommendationLoader з кешуванням результатів у статичних полях (_configCache, _energyProfiles, _weatherPatterns). Після завантаження патерни одразу сортуються за полем priority за спаданням, щоб при пошуку збігу завжди повертався найбільш пріоритетний варіант.

Модель контексту. Вхідним параметром для генерації є об'єкт RecommendationContext, що агрегує всі необхідні дані в одному місці: показники настрою (energy, valence), метеодані (temperature, windSpeed, cloudiness, precipitation), географічний та часовий контекст (climateZone, isDaytime, season). Тип опадів (precipitationType) обчислюється автоматично при створенні контексту методом _determinePrecipitationType() на основі кількості опадів та текстового опису погоди - це дозволяє розрізняти, наприклад, drizzle, light_rain, rain, heavy_rain, snow, thunderstorm.

Алгоритм генерації. Метод generate() виконує такі кроки:

1. Перевірка кешу результату. Якщо контекст не змінився з попереднього виклику (порівнюються всі 9 полів через _contextsEqual()), повертається збережений результат без повторного обчислення.

2. Фільтрація конфігурацій. Метод config.matches(context) перевіряє виконання всіх умов ConditionSet: діапазони energy та valence, тип і кількість опадів, температуру, вітер, хмарність, час доби та сезон. Умови є незалежними - кожна перевірена умова звужує вибірку.

3. Обчислення релевантності. Для кожної відповідної конфігурації ConditionEvaluator.calculateRelevance() обчислює числову оцінку від 0.0 до 1.5. Базова оцінка - 1.0, до якої застосовуються бонуси за «ідеальне» попадання в діапазон умов та штрафи за граничні значення:

```

// Бонус за точне попадання в діапазон кількості опадів
final rangeCenter = (min + max) / 2;
final distance = (amount - rangeCenter).abs();
final bonus = 0.1 * (1 - (distance / (rangeWidth / 2)).clamp(0.0,
1.0));
score += bonus;

// Штраф за граничне значення energy
if (context.energy > energyMax - 0.1) score -= 0.2;

```

4. Застосування погодного патерну. Якщо знайдено відповідний WeatherPattern, для кожної рекомендації перевіряється, чи входить її категорія до списків boostCategories або avoidCategories. Категорії зі списку boost отримують множник boostMultiplier до оцінки релевантності, категорії зі списку avoid - множник 0.5 (фактичне пригнічення).

5. Диверсифікація. Відсортований за релевантністю список проходить через _diversifyRecommendations(), який гарантує, що в фінальному результаті (до 5 рекомендацій) кожна категорія представлена не більше одного разу на першому проході, а дублікати додаються лише якщо ліміт не досягнуто.

6. Розумні fallback-и. Якщо жодна конфігурація не дала збігу, або якщо кількість рекомендацій менша за 3, система генерує запасні поради з трьох джерел у порядку пріоритету: профіль енергії → погодний патерн → універсальні правила (наприклад, «Зігрійтесь» при температурі нижче 0°C або «Затишний день вдома» при дощі).

Таким чином, клієнтський застосунок Weatherfy являє собою цілісну систему, в якій кожен компонент виконує чітко визначену роль. Дворівневий кеш (memory + disk) забезпечує надійну роботу при нестабільному з'єднанні. Система тем на основі єдиної абстракції AppColorPalette дозволяє підтримувати шість візуальних варіантів без дублювання коду. Сезонні анімаційні ефекти реалізовані через уніфікований патерн Overlay/Particle/Painter і повністю керуються через particleSettingsProvider. Recommendation Engine генерує персоналізовані поради повністю на стороні клієнта, використовуючи конфігурацію з JSON-активів та багатокроковий алгоритм фільтрації, ранжування і диверсифікації. У наступному підрозділі розглядається реалізація серверного компонента Weatherfy_API.

3.2. Реалізація серверної частини – Weatherfy API

Серверний компонент Weatherfy API реалізовано як легковажний HTTP-сервер на мові Dart з використанням фреймворку Shelf. Вибір Dart для серверної частини забезпечує однорідність технологічного стеку з клієнтським застосунком, спільне використання моделей даних (responses.dart) та спрощене підтримання кодової бази. Сервер виконує роль оркестратора між трьома зовнішніми джерелами: Open-Meteo API (метеодані), власним модулем розрахунку настрою та Deezer API (музичні рекомендації).

Точка входу та конфігурація сервера

Точкою входу є bin/server.dart, де формується конвеєр обробки запитів та запускається HTTP-сервер на порті 8080:

```
final handler = Pipeline()
  .addMiddleware(_corsMiddleware())
  .addMiddleware(logRequests())
  .addHandler(buildRouter().call);

final server = await shelf_io.serve(handler, InternetAddress.anyIPv4, 8080);
```

Конвеєр включає два middleware: CORS і логування. CORS-middleware реалізовано вручну - він перехоплює preflight-запити (OPTIONS) та додає заголовки Access-Control-Allow-* до кожної відповіді. Це необхідно для коректної роботи з мобільним клієнтом у розробці. Middleware логування (logRequests()) входить до стандартної бібліотеки Shelf і виводить метод, шлях та код відповіді для кожного запиту.

Маршрутизація

Маршрутизатор реалізовано через shelf_router і містить чотири ендпоінти, наведені в таблиці 3.5.

Таблиця 3.5 - Ендпоінти Weatherfy API

Метод	Шлях	Параметри	Призначення
GET	/weather	lat, lon (query)	Отримання погоди від Open-Meteo
POST	/mood	weather, latitude, climate_zone (body)	Розрахунок показників настрою
GET	/music	energy, valence, limit (query)	Підбір музики через Deezer
GET	/health	-	Перевірка стану сервера

Кожен обробник є незалежним і не зберігає стану між запитами. Три сервіси (WeatherService, MoodService, DeezerService) ініціалізуються одноразово при старті сервера як глобальні об'єкти в router.dart. Згідно з підходом Newman S., така мікросервісна архітектура дозволяє ізолювати окремі компоненти системи та спрощує їх подальший розвиток і підтримку [16]. Відповіді уніфіковано через допоміжні функції `_ok()`, `_badRequest()` та `_serverError()`, що формують JSON з відповідним HTTP-кодом та заголовком `content-type: application/json`.

WeatherService: інтеграція з Open-Meteo

WeatherService відповідає за отримання метеорологічних даних. При виклику `getWeather(lat, lon)` формується GET-запит до `https://API.open-meteo.com/v1/forecast` з явним переліком полів для трьох часових горизонтів: поточні умови (`current`), погодинний прогноз (`hourly`) та добовий прогноз (`daily`).

Поточні умови включають: температуру (`temperature_2m`), відчутну температуру, вологість, тиск, швидкість вітру, хмарність, кількість опадів, WMO-код погоди та ознаку дня/ночі. Погодинний прогноз запитується на 7 днів з аналогічним набором параметрів плюс імовірність опадів. Добовий прогноз включає мінімальну та максимальну температуру, суму опадів, схід/захід сонця та максимальну швидкість вітру. Швидкість вітру явно запитується в одиницях м/с (`wind_speed_unit: ms`), часовий пояс встановлюється автоматично (`timezone: auto`) - сервер Open-Meteo визначає його за координатами.

Сира відповідь Open-Meteo десеріалізується методом `WeatherResponse.fromOpenMeteoJson()`. WMO-код погоди перетворюється на текстовий опис та іконку за двома функціями-таблицями (`wmoDescription()`, `wmoIconCode()`), визначеними в `responses.dart`. Результат передається клієнту в єдиному об'єкті, що агрегує поточні умови, погодинний та добовий прогнози.

MoodService: розрахунок показників настрою

MoodService є ключовим власним компонентом сервера. Він обчислює два параметри - `energy` (0.0 - 1.0) та `valence` (0.0 - 1.0) - на основі погодних умов та

географічного контексту користувача. Обидва параметри є психоакустичними характеристиками: `energy` відображає збудженість/активність, `valence` - позитивність/негативність настрою. Саме ці показники є вхідними даними для підбору музики через Deezer. Повну реалізацію `MoodService.calculateMood()` наведено в Додатку Г.

Конфігурація через JSON-файли. Логіка розрахунку повністю параметризована та зберігається в двох типах файлів у директорії `assets/config/mood/`:

- `base.json` - базові вагові коефіцієнти для всіх кліматичних чинників (температура, хмарність, опади, вітер) та початкові значення `energy` і `valence` (обидва = 0.5);
- `zones/<zone_name>.json` - зональні конфігурації, що визначають температурні діапазони та коефіцієнти чутливості для кожної з 13 підтримуваних кліматичних зон.

Розділення на базову та зональну конфігурації дозволяє змінювати поведінку системи для конкретної зони без торкання загальної логіки. Наприклад, легкий сніг у зоні `temperate_north` отримує додатковий бонус до `valence` (+0.05), оскільки для помірного клімату він є очікуваним і естетично приємним явищем, тоді як для тропічних зон такий самий сніг матиме інший вплив.

Алгоритм розрахунку. Метод `calculateMood()` виконує послідовне накопичення відхилень від базових значень за п'ятьма чинниками:

1. Температура. Поточна температура класифікується за діапазонами зональної конфігурації на одну з категорій (`freezing`, `cold`, `cool`, `comfortable`, `warm`, `hot`, `very_hot`). Для знайденої категорії з `base.json` беруться базові ваги. Зональний коефіцієнт чутливості (`zone_sensitivity`) модулює вплив: зони зі звичними морозами менш чутливі до категорії `cold`. Додатково обчислюється `intensity` - відносна відстань від центру діапазону, що дозволяє реалізувати плавний перехід замість різкого стрибка на межі категорій.

2. Хмарність. Частка хмарності (0–100%) масштабується до відрізка [0.0, 1.0] і множиться на базові ваги категорії overcast. Таким чином вплив хмарності є лінійним і неперервним.

3. Опади. Тип опадів визначається функцією `_getPrecipitationType()` за WMO-кодом. Пріоритет WMO-коду над текстовим описом гарантує коректну класифікацію незалежно від мови опису. Вплив опадів посилюється наявністю вітру: коефіцієнт $\text{windImpact} = 1.0 + (\text{windSpeed} / 15.0).clamp(0.0, 0.5)$ збільшує негативний ефект дощу при вітрі до 50%.

4. Вітер. Швидкість вітру (м/с) класифікується на чотири категорії: calm (< 3), breezy (3–8), windy (8–15), storm (> 15). Кожна категорія має власні ваги в `base.json`.

5. Час доби (біоритм). Локальний час розраховується за UTC та зміщенням часового поясу, отриманим від Open-Meteo. Прогрес дня `dayProgress` - нормована величина від 0.0 (схід) до 1.0 (захід). Вплив реалізовано через синусоїду:

```
energy += 0.15 * sin(dayProgress * π);  
valence += 0.10 * sin(dayProgress * π);
```

Синусоїда забезпечує пік показників в середині світлового дня та спад вранці та ввечері. До і після сонця `dayProgress = 0.0`, тому нічний час не отримує додаткового впливу через цей канал.

Після накопичення всіх відхилень обидва значення затискаються в межах [0.0, 1.0]. При будь-якій непередбаченій помилці сервіс повертає нейтральний стан `{energy: 0.5, valence: 0.5}` замість HTTP 500, що гарантує безперервність роботи клієнта.

DeezerService: підбір музики за настроєм

`DeezerService` транслює пару (`energy`, `valence`) у музичну добірку через публічний Deezer API. Координатна площина `energy × valence` розбивається на чотири квадранти, кожному з яких відповідає жанровий ідентифікатор та пошуковий тег:

Таблиця 3.6 - Мappінг настрою на музичний квадрант

Квадрант	Energy	Valence	Genre ID	Пошуковий тег
high energy positive	≥ 0.5	≥ 0.5	132	happy upbeat
high energy negative	≥ 0.5	< 0.5	152	intense dark
low energy positive	< 0.5	≥ 0.5	53	chill relax
low energy negative	< 0.5	< 0.5	6	melancholic ambient

Стратегія отримання треків є дворівневою з автоматичним fallback. Основний шлях: за genreId запитується список радіостанцій жанру (/genre/{id}/radios), береться перша радіостанція, і для неї запитується список треків (/radio/{id}/tracks). Якщо основний шлях повертає порожній результат (наприклад, жанр не має радіо в Deezer), система автоматично перемикається на текстовий пошук за відповідним тегом (/search?q=...&order=RANKING). Такий підхід забезпечує стійкість до змін у структурі Deezer API без зміни клієнтського коду.

Розгортання

Сервер розгорнуто на хмарному сервері з адресою 43.157.5.90:8080 у Docker-контейнері. Як зазначено в документації Docker, контейнеризація дозволяє забезпечити ізольоване та відтворюване середовище виконання програмного забезпечення [18]. Збірка реалізована як двоетапний (multi-stage) Dockerfile. На першому етапі (builder) виконується компіляція Dart-коду в нативний бінарний файл командою dart compile exe bin/server.dart -o bin/server. На другому етапі використовується мінімальний базовий образ debian:bookworm-slim, до якого копіюються лише скомпільований бінарник та директорія assets/ з JSON-конфігурацією настрою. Це дозволяє зменшити кінцевий розмір образу, виключивши Dart SDK та вихідний код [16].

```
FROM dart:stable AS builder
WORKDIR /app
COPY pubspec.yaml pubspec.lock* ./
RUN dart pub get
COPY . .
RUN dart compile exe bin/server.dart -o bin/server

FROM debian:bookworm-slim
WORKDIR /app
COPY --from=builder /app/bin/server ./bin/server
```

```
COPY --from=builder /app/assets ./assets
RUN chmod +x ./bin/server
EXPOSE 8080
CMD ["/bin/server"]
```

Управління контейнером здійснюється через `docker-compose.yml` з політикою перезапуску `unless-stopped`, що забезпечує автоматичне відновлення після збоїв або перезавантаження хост-машини.

Таким чином, серверна частина Weatherfy API є мінімалістичним, але функціонально повним компонентом: вона агрегує метеодані від Open-Meteo, обчислює показники настрою через власну параметричну модель з урахуванням кліматичної зони та часу доби, та транслює результат у музичні рекомендації через Deezer. Архітектурне рішення компілювати Dart у нативний бінарник забезпечує низький час запуску контейнера та мінімальне споживання ресурсів на хост-машині.

3.3. Аналіз якості та тестування програмного забезпечення

Тестування програмного забезпечення Weatherfy охоплює обидва компоненти системи - мобільний клієнт та серверну частину - і проводилось у два етапи: статичний аналіз коду та функціональне тестування в реальному середовищі виконання.

Статичний аналіз коду

Перший рівень забезпечення якості - статичний аналіз - застосовувався постійно в процесі розробки. Обидва проєкти (клієнт та сервер) написано на Dart, що дозволяє використовувати вбудований аналізатор `dart analyze` та офіційний пакет `lints`, підключений у `dev_dependencies`. Аналізатор перевіряє відповідність стандартам мови: типову безпеку (`null safety`), невикористовувані імпорти та змінні, порушення конвенцій іменування, потенційно небезпечні перетворення типів.

Dart у режимі `sound null safety` вимагає явної обробки `nullable`-значень на рівні компілятора [\[12\]](#) . Це суттєво знижує клас помилок, пов'язаних з

NullPointerException: усі поля, що можуть бути null, позначено типом T?, а їх використання вимагає явного ??, ?. або перевірки. У коді рекомендаційного рушія це забезпечує, наприклад, безпечну роботу з опціональними полями ConditionSet:

```
if (energyMin != null && context.energy < energyMin!) return false;
```

Сервер Weatherfy API компілюється в нативний бінарник командою dart compile exe, що є додатковим рівнем перевірки: компілятор AOT (Ahead-of-Time) виконує повний аналіз типів і не дозволяє зібрати бінарник за наявності помилок типізації.

Функціональне тестування серверного API. Тестування серверної частини проводилось методом ручного надсилання HTTP-запитів до задеплого сервера (43.157.5.90:8080) із перевіркою коректності відповідей. Для кожного ендпоінту визначено набір тест-кейсів, наведених у таблиці 3.5.

Таблиця 3.7 - Функціональні тест-кейси серверного API

Ендпоінт	Сценарій	Вхідні дані	Очікуваний результат	Фактичний результат
GET /health	Перевірка доступності	-	200 {"status":"ok"}	Відповідає
GET /weather	Коректні координати	lat=50.45, lon=30.52	200, об'єкт з поточною погодою, прогнозами	Відповідає
GET /weather	Відсутній параметр	lat=50.45 (без lon)	400 {"error":"lat and lon are required"}	Відповідає
GET /weather	Некоректне значення	lat=abc&lon=30	400 {"error":"lat and lon are required"}	Відповідає
POST /mood	Типова зона	temperate_north, літня погода	200, energy та valence в [0.0, 1.0]	Відповідає
POST /mood	Критична погода	Гроза (WMO 99), 15 м/с вітер	200, знижені energy і valence	Відповідає
POST /mood	Відсутнє поле	Без climate_zone	400 {"error":"...are required"}	Відповідає
POST /mood	Невідома зона	climate_zone: "mars"	200, нейтральні значення 0.5/0.5 (fallback)	Відповідає
GET /music	Квадрант HE+HV	energy=0.8, valence=0.7	200, список треків жанру 132	Відповідає
GET /music	Квадрант LE-LV	energy=0.2, valence=0.3	200, меланхолійний жанр 6	Відповідає
GET /music	Відсутній параметр	Без valence	400 {"error":"energy and valence are required"}	Відповідає

Окремо перевірено поведінку при недоступності зовнішніх API: при симуляції відсутності мережі на стороні сервера MoodService повертає нейтральний стан {energy: 0.5, valence: 0.5} замість HTTP 500, що є задокументованою поведінкою безпечного fallback.

Тестування розрахунку настрою

Коректність алгоритму MoodService.calculateMood() перевірялась через порівняння очікуваних та фактичних значень energy і valence для контрольних погодних сценаріїв. Верифікація проводилась шляхом ручного обчислення за конфігурацією base.json та temperate_north.json та порівняння з відповіддю сервера.

Таблиця 3.8 - Верифікація алгоритму розрахунку настрою (зона temperate north)

Сценарій	Temp	Хмарність	Опади	Вітер	Очік. energy	Факт. energy	Очік. valence	Факт. valence
Ідеальний літній день	22°C	10%	none	calm	~0.82	0.81	~0.87	0.86
Похмурий дощовий день	12°C	85%	light_rain	breezy	~0.38	0.39	~0.28	0.29
Морозна ясна ніч	-5°C	0%	none	calm	~0.22	0.23	~0.27	0.27
Гроза з сильним вітром	18°C	100%	heavy_rain	storm (18 м/с)	~0.11	0.12	~0.08	0.09
Легкий сніг вдень	0°C	40%	light_snow	calm	~0.39	0.40	~0.41	0.42

Розбіжність між очікуваними та фактичними значеннями не перевищує ± 0.02 , що пояснюється плавним розрахунком intensity на основі відстані від центру температурного діапазону - ефект, що не враховується при спрощеному ручному підрахунку. Отримані відхилення є прийнятними та не впливають на класифікацію квадранту настрою.

Тестування рекомендаційного рушія

Рекомендаційний рушій тестувався через вбудований у клас RecommendationEngine метод debugAllMatches(), що повертає список усіх конфігурацій з результатом перевірки умов та причиною збігу або відхилення. Це

дозволило верифікувати як позитивні сценарії (рекомендація видається), так і негативні (рекомендація коректно відфільтровується).

Перевірено такі аспекти алгоритму:

- Диверсифікація категорій. При генерації для типового контексту з 12+ рекомендаціями, що пройшли фільтрацію, результат містить не більше однієї рекомендації кожної категорії на перших позиціях. Повторення категорій допускаються лише якщо загальна кількість унікальних категорій менша за `maxCount` (5).

- Пріоритет погодних патернів. Перевірено, що при наявності двох патернів, умови яких частково перекриваються, завжди застосовується патерн з вищим полем `priority`. Патерни з однаковим пріоритетом повертають перший за порядком у відсортованому списку.

- Fallback-ланцюжок. При контексті, для якого жодна з конфігурацій не дає збігу (наприклад, дуже специфічна комбінація параметрів), система послідовно переходить до профілю енергії, погодного патерну та універсальних правил. Результат завжди містить мінімум одну рекомендацію завдяки гарантованому `universal fallback`.

- Кешування результату. Повторний виклик `generate()` з ідентичним контекстом повертає той самий об'єкт без повторного обчислення. Перевірено через виклик з мутованим контекстом (зміна одного поля) - кеш коректно інвалідується.

Тестування клієнтського застосунку

Мобільний клієнт тестувався в режимі ручного функціонального тестування на емуляторі Android та фізичному пристрої. Перевірено наступні сценарії:

Сценарії роботи з мережею. При доступній мережі дані завантажуються та відображаються коректно. При відключенні мережі після першого завантаження застосунок відображає кешовані дані без переходу в стан помилки («`silent error mode`»). При відключенні мережі до першого завантаження відображається

повідомлення про помилку. Після відновлення з'єднання дані автоматично оновлюються через retry-таймер (інтервал 15 секунд).

Сценарії кешування. Після закриття та повторного відкриття застосунку протягом 10 хвилин дані відображаються миттєво з диску без мережевого запиту. Після закінчення TTL (10 хвилин) при наступному відкритті виконується фоновий запит на оновлення. Ручне очищення кешу через Settings призводить до повного запиту при наступному відкритті екрану погоди.

Сценарії навігації та стану. Перемикання між містами коректно ініціює ланцюжок `weatherNotifierProvider` → `moodProvider` для обраного міста. Паралельне відображення даних для кількох міст не спричиняє змішування стану між екранами. Додавання та видалення міст через `CityListScreen` негайно відображається без перезапуску застосунку.

Сценарії теми та ефектів. Зміна теми через Settings застосовується миттєво та зберігається між сесіями. Параметри анімаційних ефектів (кількість, розмір, швидкість частинок) коректно передаються до всіх активних оверлеїв через `particleSettingsProvider`.

Загальна оцінка якості

За результатами проведеного тестування програмне забезпечення Weatherfy відповідає функціональним вимогам, визначеним у розділі 1. Усі критичні сценарії - робота в офлайн-режимі, обробка помилок зовнішніх API, коректність розрахунку настрою - пройшли перевірку успішно. Виявлені в процесі розробки дефекти (пріоритизація погодних патернів, дефолтне значення `boostMultiplier`) були задокументовані коментарями // FIX: безпосередньо в коді та усунені до фінальної версії.

Перспективою для підвищення рівня якості є впровадження автоматизованого модульного тестування критичних компонентів: `ConditionEvaluator.calculateRelevance()`, `MoodService.calculateMood()` та `DeezerService._getQuadrant()`. Ці методи є чистими функціями з детермінованим виводом, що робить їх ідеальними кандидатами для unit-тестів із використанням стандартного пакету `flutter_test` / `test`.

3.4. Аналіз безпеки та захисту даних

Аналіз безпеки програмного забезпечення Weatherfy охоплює чотири аспекти: захист мережових комунікацій, захист даних на пристрої, обробку зовнішніх залежностей та модель загроз у контексті функціональності застосунку.

Мережева безпека

Усі зовнішні HTTP-з'єднання серверного компонента використовують захищений протокол HTTPS.

Сервіс Open-Meteo звертається до <https://API.open-meteo.com>, Deezer - до <https://API.deezer.com>, геокодинг через Nominatim виконується на стороні клієнта за адресою <https://nominatim.openstreetmap.org>. Таким чином, усі дані між сервером і третіми сторонами передаються у зашифрованому вигляді з верифікацією сертифікатів засобами стандартної HTTP-бібліотеки Dart.

Комунікація між мобільним клієнтом та власним серверним компонентом (Weatherfy_API) виконується через незашифрований протокол HTTP (<http://43.157.5.90:8080>). Це є свідомим компромісом у межах навчального проєкту: налаштування TLS-сертифікату для статичної IP-адреси без прив'язаного доменного імені потребує ручного керування самопідписаним сертифікатом, що ускладнює розгортання без суттєвого підвищення практичної безпеки в даному контексті. У виробничому середовищі ця комунікація повинна бути захищена HTTPS з валідним сертифікатом - наприклад, через Let's Encrypt після прив'язки доменного імені та через reverse proxy на базі Nginx або Caddy.

Сервер реалізує CORS-middleware, що обмежує перелік дозволених HTTP-методів (GET, POST, OPTIONS). Усі інші методи (наприклад, DELETE, PUT) обробляються маршрутизатором Shelf як 404 Not Found, оскільки відповідні обробники відсутні. Це фактично реалізує принцип мінімальної поверхні атаки на рівні маршрутизації.

Захист даних на пристрої

Мобільний клієнт зберігає на пристрої три категорії даних через SharedPreferences:

Таблиця 3.9 - Дані, що зберігаються локально

Категорія	Ключі	Вміст	Чутливість
Список міст	cities	Назва, країна, координати (lat/lon), ознака домашнього міста	Низька
Кеш погоди	weather_cache_{cityId}	Метеодані, показники настрою, час отримання	Низька
Налаштування	selected_theme, temp_unit, particle_*	Тема, одиниці температури, параметри ефектів	Відсутня

Жодна з категорій не містить персональних даних у розумінні GDPR: імена, email-адреси, ідентифікатори пристрою або будь-які біометричні дані не збираються та не зберігаються. Координати збережених міст є географічними назвами, обраними користувачем вручну, - вони не відображають реального місцеперебування користувача та не можуть використовуватися для його ідентифікації.

SharedPreferences на Android зберігає дані у XML-файлах у захищеному пісочниці (sandboxed) сховищі застосунку. Доступ до цих файлів без root-прав іншого застосунку неможливий завдяки механізму ізоляції процесів Android. На iOS аналогічну роль виконує NSUserDefaults у захищеному контейнері застосунку. Оскільки дані не є чутливими, використання стандартного SharedPreferences є доцільним; для зберігання токенів або паролів слід застосовувати flutter_secure_storage з апаратним шифруванням через Android Keystore / iOS Secure Enclave.

Безпека конфігурації. Адреса серверного компонента задається через змінну середовища BACKEND_URL під час компіляції:

```
static const String backendBaseUrl = String.fromEnvironment(  
  'BACKEND_URL',  
  defaultValue: 'http://43.157.5.90:8080',  
);
```

Такий підхід має важливий наслідок: скомпільований рядок є статичним і може бути витягнутий з APK/IPA-файлу стандартними інструментами реверс-інжинірингу (наприклад, strings або декомпіляторами). Це прийнятно, оскільки IP-адреса сервера не є секретом - він є публічно доступним. Якби в рядку замість адреси містився API-ключ, такий спосіб зберігання був би неприпустимим: секрети ніколи не повинні компілюватися в бінарник клієнтського застосунку і мають передаватися виключно через захищені серверні ендпоінти.

Серверний компонент не використовує жодних API-ключів - Open-Meteo та Deezer надають публічний доступ без автентифікації. Якби ключі були потрібні, коректним підходом було б зберігання у змінних середовища Docker-контейнера (через docker-compose.yml секцію environment або через .env-файл поза репозиторієм), а не хардкодинг у вихідному коді.

Відсутність автентифікації: обґрунтована модель загроз. Серверний API Weatherfy не реалізує автентифікацію або авторизацію запитів. Будь-хто, хто знає IP-адресу сервера, може надсилати запити до всіх чотирьох ендпоінтів. З точки зору моделі загроз це є прийнятним рішенням з таких міркувань:

- Відсутність чутливих даних. Сервер не зберігає і не повертає персональні дані користувачів. Усі запити є stateless - сервер не веде жодних сесій, баз даних або журналів, прив'язаних до конкретного користувача.
- Відсутність деструктивних операцій. API не має ендпоінтів для запису, зміни або видалення будь-яких даних. Усі ендпоінти виконують лише читання від зовнішніх джерел та обчислення.
- Низька цінність для зловмисника. Несанкціонований запит до /weather або /mood надає лише загальнодоступну метеорологічну інформацію - ту саму, що безкоштовно доступна через Open-Meteo API напряму.
- Реальним ризиком є надмірне використання (abuse) через нелімітовану кількість запитів - наприклад, цілеспрямоване навантаження на сервер. У виробничому середовищі цей ризик нейтралізується rate limiting middleware (наприклад, shelf_rate_limiter), що обмежує кількість запитів з однієї IP-адреси за одиницю часу.

Таблиця 3.10 - Оцінка стану безпеки компонентів системи

Аспект	Стан	Примітка
HTTPS для зовнішніх API	Реалізовано	Open-Meteo, Deezer, Nominatim
HTTPS клієнт → сервер	Відсутній	HTTP; прийнятно для навч. проєкту
CORS обмеження методів	Реалізовано	Дозволено лише GET, POST, OPTIONS
Відсутність чутливих даних у сховищі	Реалізовано	Координати, не персональні дані
Ізоляція сховища Android/iOS	Реалізовано	Sandbox SharedPreferences
Секрети поза кодом	Реалізовано	Ключів немає; адреса через env var
Автентифікація API	Відсутня	Обґрунтовано моделлю загроз
Rate limiting	Відсутній	Рекомендовано для production
Валідація вхідних даних	Часткова	Перевірка наявності та типів параметрів

Таким чином, рівень безпеки системи Weatherfy відповідає її функціональному призначенню та характеру даних, що обробляються. Ключові вразливості - відсутність HTTPS між клієнтом і власним сервером та відсутність rate limiting - є відомими та задокументованими компромісами, усунення яких є пріоритетом при переході до виробничого розгортання.

3.5. Системні та апаратні вимоги

Вимоги до програмного забезпечення та апаратного забезпечення визначаються окремо для двох компонентів системи: мобільного клієнта та серверної частини.

Вимоги до мобільного клієнта. Застосунок Weatherfy розроблено на Flutter 3.x з мінімальною версією Dart SDK 3.0.0. На платформі Android мінімальна підтримувана версія - Android 5.0 (API level 21), що охоплює понад 98% активних Android-пристроїв. На платформі iOS мінімальна версія - iOS 12.0. Апаратні вимоги є мінімальними: достатньо будь-якого смартфона з процесором ARMv7 або ARM64, 1 ГБ оперативної пам'яті та 50 МБ вільного місця у внутрішньому сховищі для встановлення. Для повноцінної роботи необхідне інтернет-з'єднання (Wi-Fi або мобільний інтернет) - без нього доступні лише кешовані дані в межах TTL 10 хвилин. Відтворення музичних прев'ю потребує підтримки аудіовиводу (динамік або навушники).

Таблиця 3.11 - Вимоги до мобільного клієнта

Параметр	Android	iOS
Мінімальна версія ОС	Android 5.0 (API 21)	iOS 12.0
Архітектура процесора	ARMv7, ARM64, x86_64	ARM64
Мін. обсяг RAM	1 ГБ	1 ГБ
Місце для встановлення	~50 МБ	~50 МБ
Мережеве з'єднання	Рекомендовано	Рекомендовано
Flutter SDK	3.x	3.x

Вимоги до серверного компонента. Серверна частина розгортається як Docker-контейнер на базі образу `debian:bookworm-slim`. Мінімальні апаратні вимоги до хост-машини: 1 vCPU, 256 МБ оперативної пам'яті та 200 МБ дискового простору (для образу та JSON-конфігурацій). Поточне розгортання виконано на хмарному сервері з публічною адресою 43.157.5.90.

Для роботи сервера необхідний вихідний доступ до інтернету для звернень до Open-Meteo та Deezer API. Порт 8080 TCP має бути відкритий для вхідних з'єднань від мобільного клієнта. Docker Engine версії 20.10 або вище є єдиною системною залежністю на хості.

Таблиця 3.12 - Вимоги до серверного компонента

Параметр	Мінімум	Рекомендовано
CPU	1 vCPU	2 vCPU
RAM	256 МБ	512 МБ
Дисковий простір	200 МБ	1 ГБ
ОС хоста	Linux (будь-який дистрибутив з Docker)	Ubuntu 22.04 LTS
Docker Engine	20.10+	24.x
Відкритий порт	8080 TCP	-
Вихідний інтернет	Обов'язково	-

Сервер Dart, скомпільований у нативний бінарник, має час холодного запуску менше 1 секунди та споживає в режимі очікування близько 15–20 МБ оперативної пам'яті, що робить його придатним для розгортання навіть на мінімальних хмарних інстансах категорії «nano» або «micro».

3.6. Порівняння з існуючими аналогами

Weatherfy є вузькоспеціалізованим застосунком, що поєднує прогноз погоди з музичними рекомендаціями на основі настрою. Порівняння проводиться з трьома категоріями: класичні погодні застосунки, музичні сервіси з mood-функціоналом та найближчий концептуальний аналог.

Таблиця 3.13 - Порівняння Weatherfy з аналогами

Функція	Weatherfy	The Weather Channel	Spotify (mood)	Moodagent
Прогноз погоди	+	+	-	-
Погодинний прогноз	+	+	-	-
Музичні рекомендації	+	-	+	+
Прив'язка музики до погоди	+	-	-	-
Немузичні рекомендації	+	-	-	-
Облік кліматичної зони	+	-	-	-
Сезонні теми інтерфейсу	+	-	-	-
Робота офлайн (кеш)	+	+	Частково	-
Відкритий API (без ключа)	+	-	-	-
Безкоштовно без реєстрації	+	Частково	-	-

The Weather Channel / Weather Underground - еталонні погодні застосунки з детальними метеоданими, радарними картами та сповіщеннями. Порівняно з ними Weatherfy поступається глибиною метеорологічних даних (відсутні радар, супутникові знімки, прогноз на 14 днів), проте виграє у простоті та унікальності концепції: музична та поведінкова складова відсутні в жодному масовому погодному застосунку.

Spotify з функцією Mood Playlists - найпотужніший конкурент у частині музичних рекомендацій за настроєм. Spotify використовує власні аудіо-характеристики треків (власне ті самі energy та valence) та глибокий машинне навчання для персоналізації. Weatherfy не може конкурувати з якістю музичних рекомендацій Spotify, проте пропонує унікальний зв'язок: рекомендація будується не на основі історії прослуховувань, а безпосередньо на поточних погодних умовах та часі доби. Крім того, Weatherfy не потребує облікового запису та підписки.

Moodagent - найближчий концептуальний аналог: застосунок підбирає музику на основі поточного настрою користувача. Принципова відмінність - в Moodagent настрій вводить сам користувач вручну (повзунки), тоді як Weatherfy визначає його автоматично на основі погодних даних та кліматичного контексту. Weatherfy додає поведінкові рекомендації (одяг, активності, режим), яких у Moodagent немає.

Таким чином, Weatherfy займає унікальну нішу на перетині двох категорій застосунків, не представлену серед масових рішень: автоматична детермінація настрою через погоду з одночасним наданням музичних та поведінкових рекомендацій без необхідності реєстрації або підписки.

3.7. Висновки до розділу 3

У третьому розділі описано повний цикл реалізації та аналізу програмного забезпечення Weatherfy - від клієнтського застосунку до серверного компонента, тестування та оцінки безпеки.

Реалізацію мобільного клієнта виконано на Flutter з дотриманням принципів Clean Architecture: чіткий розподіл на шари core, data, domain та presentation забезпечує незалежність бізнес-логіки від фреймворку та зовнішніх API. Управління станом через Riverpod реалізує реактивну модель без надлишкового зв'язування компонентів. Дворівневий кеш (memory + disk) з TTL 10 хвилин та механізм автоматичного відновлення з'єднання через експоненційний відкат забезпечують стійку роботу в умовах нестабільного мережевого з'єднання. Recommendation Engine реалізовано повністю на стороні клієнта: алгоритм фільтрації, ранжування за релевантністю, застосування погодних патернів та диверсифікація категорій гарантують змістовний результат у будь-якому контексті завдяки багаторівневому fallback-ланцюжку. Візуальна складова представлена шістьма темами на основі єдиної абстракції AppColorPalette та чотирма сезонними анімаційними ефектами за уніфікованим патерном Overlay/Particle/Painter.

Серверний компонент Weatherfy_API реалізовано на Dart/Shelf як stateless HTTP-сервер з чотирма ендпоінтами. WeatherService агрегує метеодані від Open-Meteo з явним переліком полів для трьох часових горизонтів. MoodService обчислює показники energy та valence через параметричну модель з JSON-конфігурацією, що враховує п'ять незалежних чинників - температуру, хмарність, опади, вітер та біоритм - з поправкою на кліматичну зону. DeezerService транслює обчислений настрій у музичну добірку через двохрівневу стратегію з автоматичним fallback на текстовий пошук. Сервер розгорнуто у Docker-контейнері зі збіркою через multi-stage Dockerfile, що мінімізує розмір образу та виключає Dart SDK з виробничого середовища.

Тестування підтвердило коректність роботи всіх компонентів системи. Статичний аналіз засобами dart analyze та sound null safety усував цілий клас помилок типізації на етапі компіляції. Функціональне тестування серверних ендпоінтів охопило типові та граничні сценарії, включно з некоректними вхідними даними та поведінкою при недоступності зовнішніх API. Верифікація алгоритму MoodService показала відхилення фактичних значень від очікуваних не більше ± 0.02 , що є прийнятним для задач класифікації настрою.

Аналіз безпеки виявив, що система не обробляє персональних даних і не потребує автентифікації, що обґрунтовується її моделлю загроз. Усі зовнішні з'єднання серверного компонента захищені HTTPS. Єдиним задокументованим компромісом є використання HTTP для внутрішньої комунікації клієнта з власним сервером, усунення якого є першочерговим кроком при переході до виробничого розгортання.

Порівняння з аналогами підтвердило, що Weatherfy займає унікальну нішу на перетині погодних застосунків та mood-музичних сервісів, не представлену серед наявних масових рішень. Системні вимоги є мінімальними: Android 5.0+ або iOS 12.0+ для клієнта та Docker-контейнер з 256 МБ RAM для сервера.

ВИСНОВКИ

У кваліфікаційній роботі розроблено мобільний додаток Weatherfy - інформаційну систему, що поєднує функції прогнозування погоди з механізмом персоналізованих рекомендацій на основі погодних умов. Поставлена мета досягнута в повному обсязі: реалізовано як клієнтський застосунок для платформ Android та iOS, так і серверний компонент, що виконує агрегацію та обробку даних.

У першому розділі проведено аналіз предметної області, виявлено ключовий недолік існуючих рішень - відсутність зв'язку між метеорологічними даними та поведінковими або музичними рекомендаціями для користувача. Виконано огляд відкритих API: Open-Meteo обрано як джерело метеоданих завдяки відсутності необхідності у ключах автентифікації та широті параметрів прогнозу; Deezer - як музичну платформу з публічним доступом до треків і жанрових радіостанцій. Сформовано функціональні та нефункціональні вимоги, що стали основою для архітектурних рішень.

У другому розділі спроектовано архітектуру системи на основі Clean Architecture з розподілом на шари core, data, domain та presentation. Розроблено повний комплект UML-діаграм: компонентів, варіантів використання, класів та послідовності для ключових сценаріїв. Описано файлову структуру клієнта та серверного компонента, визначено принципи їх взаємодії.

У третьому розділі реалізовано всі компоненти системи. На стороні клієнта ключовими технічними рішеннями є дворівневий кеш із TTL та автоматичним відновленням з'єднання, рекомендаційний рушій з алгоритмом ранжування та диверсифікації на основі JSON-конфігурації, а також система сезонних тем і анімаційних ефектів. На стороні сервера реалізовано параметричну модель розрахунку настрою з урахуванням кліматичної зони та часу доби, яка транслює погодний контекст у пару психоакустичних показників energy та valence. Проведено функціональне тестування обох компонентів та аналіз безпеки, що підтвердили відповідність системи сформованим вимогам.

Наукова новизна отриманих результатів полягає у розробці параметричної моделі автоматичного визначення емоційного стану користувача на основі метеорологічних даних з урахуванням кліматичної зони та добового біоритму. На відміну від існуючих підходів, де настрої вводяться користувачем вручну або визначається на основі історії прослуховувань, запропонована модель використовує зовнішні об'єктивні чинники - температуру, хмарність, тип і інтенсивність опадів, швидкість вітру та час доби - для автоматичного обчислення психоакустичних показників energy та valence. Додатковою новизною є реалізація клієнтського рекомендаційного рушія з багатокроковим алгоритмом фільтрації, ранжування за релевантністю та диверсифікації категорій на основі декларативної JSON-конфігурації, що дозволяє розширювати базу рекомендацій без змін у програмному коді.

Практичне значення роботи полягає у створенні повністю функціонального застосунку, розгорнутого у хмарному середовищі та готового до використання на пристроях під керуванням Android 5.0+ та iOS 12.0+. Макети готового застосунку доступні для перегляду у додатку Є. Порівняння з існуючими аналогами підтвердило унікальність підходу: жоден з масових застосунків не реалізує автоматичне визначення настрою через погодний контекст із одночасним наданням музичних та поведінкових рекомендацій без необхідності реєстрації або підписки.

Перспективами подальшого розвитку є впровадження HTTPS для внутрішньої комунікації клієнта з сервером, додавання rate limiting для захисту від надмірного навантаження, розширення автоматизованого тестування критичних алгоритмів, а також інтеграція з додатковими музичними платформами для розширення бібліотеки рекомендацій.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Liu Y., Zhang X., Wei H., Cao Z., Guo P. “Sadness smile” curve: Processing emotional information from social network for evaluating thermal comfort perception // *Journal of Thermal Biology*. 2024. Vol. 127. P. 104025. DOI: 10.1016/j.jtherbio.2024.104025.
2. Duncan B. A., Kallumadi S., Berg-Kirkpatrick T., McAuley J. MAWI Rec: Leveraging Severe Weather Data in Recommendation // *Proceedings of the 18th ACM Conference on Recommender Systems (RecSys '24)*. Bari, Italy, 2024. P. 1–5. DOI: 10.1145/3640457.3688157.
3. Trivedi R., Pati B., Panigrahi C. R. wPOI: Weather-Aware POI Recommendation Engine // *Computation y Sistemas*. 2022. Vol. 26, No. 2. DOI: 10.13053/cys-26-2-4261.
4. Zubair M., Salim M. S., Rahman M. M., et al. Agricultural Recommendation System based on Deep Learning: A Multivariate Weather Forecasting Approach // *arXiv preprint*. 2024. DOI: 10.48550/arXiv.2401.11410.
5. Martin R. C. Clean Architecture: A Craftsman’s Guide to Software Structure and Design. *Prentice Hall*. 2017. 432 p.
6. Freeman A., Pryce S. Flutter in Action. Manning Publications. 2019. 352 p.
7. Deezer API Docs. Available at: <https://developers.deezer.com/login?redirect=/api>.
8. Deezer Newsroom. Available at: <https://newsroom-deezer.com/>.
9. Iglauer S. Beginning Flutter: A Hands-On Guide to App Development. Apress. 2022. 410 p.
10. Dart Team. Dart Programming Language Specification. Version 3.0. Google. 2023. Available at: <https://dart.dev/guides/language/spec>.
11. Google. Flutter Documentation: Architectural Overview. 2024. Available at: <https://docs.flutter.dev/development/data-and-backend/state-mgmt/options>.
12. Dart Team. Shelf Package Documentation: Middleware and Web Server Development in Dart. 2024. Available at: <https://pub.dev/packages/shelf>.

13. Zippenfenig P. Open-Meteo: Open-Source Weather API Documentation. 2024. Available at: <https://open-meteo.com/en/docs>.
14. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures (REST). Doctoral Dissertation. University of California, Irvine. 2000. Available at: <https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>.
15. Richardson L., Amundsen M. RESTful Web APIs. O'Reilly Media. 2013. 408 p.
16. Newman S. Building Microservices: Designing Fine-Grained Systems. O'Reilly Media. 2021. 600 p.
17. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley. 2003. 560 p.
18. Docker Engine and System Documentation. Available at: <https://docs.docker.com/>
19. Google. Flutter Cookbook: Networking Recipes. 2024. Available at: <https://docs.flutter.dev/cookbook/networking>
20. Козуб Г. О., Козуб Ю. Г. Декларативний підхід при створенні мультиплатформних додатків. *Вісник Східноукраїнського національного університету імені Володимира Даля*, 2022. №5 (275). С. 10–15. DOI: <https://doi.org/10.33216/1998-7927-2022-275-5-10-15>
21. Козуб Ю., Козуб Г. Особливості розробки мультиплатформних застосунків на KOTLIN // *Herald of Khmelnytskyi National University. Technical sciences*. – 2023. – Т. 317. – №. 1. – С. 224-229.
22. Жуков А., Козуб Г.О. Розробка мобільного додатку за допомогою мови програмування kotlin та сервісу firebase. *Матеріали міжнародної науково-практичної - інтернет конференції «Тенденції та перспективи ро-звитку науки і освіти в умовах глобалізації»*: Сб. наук. трудів. Переяслав-Хмельницький, 2019. Вып. 52. 273-275 с.
23. Жуков А., Козуб Г. Реалізація фреймворка JETPACK COMPOSE у багатомодульному ANDROID-додатку. *Матеріали Всеукраїнської науко-во-практичної інтернет-конференції «Вітчизняна наука на зламі епох: проблеми та*

перспективи розвитку»: Зб. наук. праць. Переяслав, 2021. Вип. 67. с.109-111.

URI: <http://hdl.handle.net/123456789/7885>

24. Козуб Г.О., Козуб Ю.Г., Могильний Г.А., Жуков А.М. Розробка мобільного Android-додатку з застосуванням принципів Clean Architecture. *Вісник Східноукраїнського національного університету імені Володимира Даля*. вип. 5 (269), Вересень 2021, с. 5-10, doi:10.33216/1998-7927-2021-269-5-5-10.

25. Дем'янов В.А., Козуб Г.О. Нестандартне використання 2D Lighting у Unity. *Grail of Science*, (60), С.615–622. DOI: 10.36074/grail-of-science.26.12.2025.066 .

26. Козуб Г. О., Козуб Ю. Г. Методичні рекомендації до виконання кваліфікаційної роботи за спеціальністю 122 „Комп’ютерні науки” першого рівня вищої освіти. – Старобільськ : ДЗ „ЛНУ імені Тараса Шевченка”, 2021. 99 с.

27. Дем'янов В.А., Козуб Г.О. Інтеграція Mood Engine та Recommendation Engine у гібридній клієнт-серверній архітектурі WEATHERFY. *Grail of Science*, (68), С.991–997. DOI: 10.36074/grail-of-science.15.05.2026.110.

28. Козуб Г.О, Дем'янов В.А. Архітектура та реалізація мобільного додатку прогнозування погоди з інтегрованою рекомендаційною системою. *Наука і техніка сьогодні*. 2026. № 5(59) С. 4688-4704. DOI: 10.52058/2786-6025-2026-5(59)-4688-4704.

ДОДАТКИ

Додаток А. Загальна архітектура системи Weatherfy

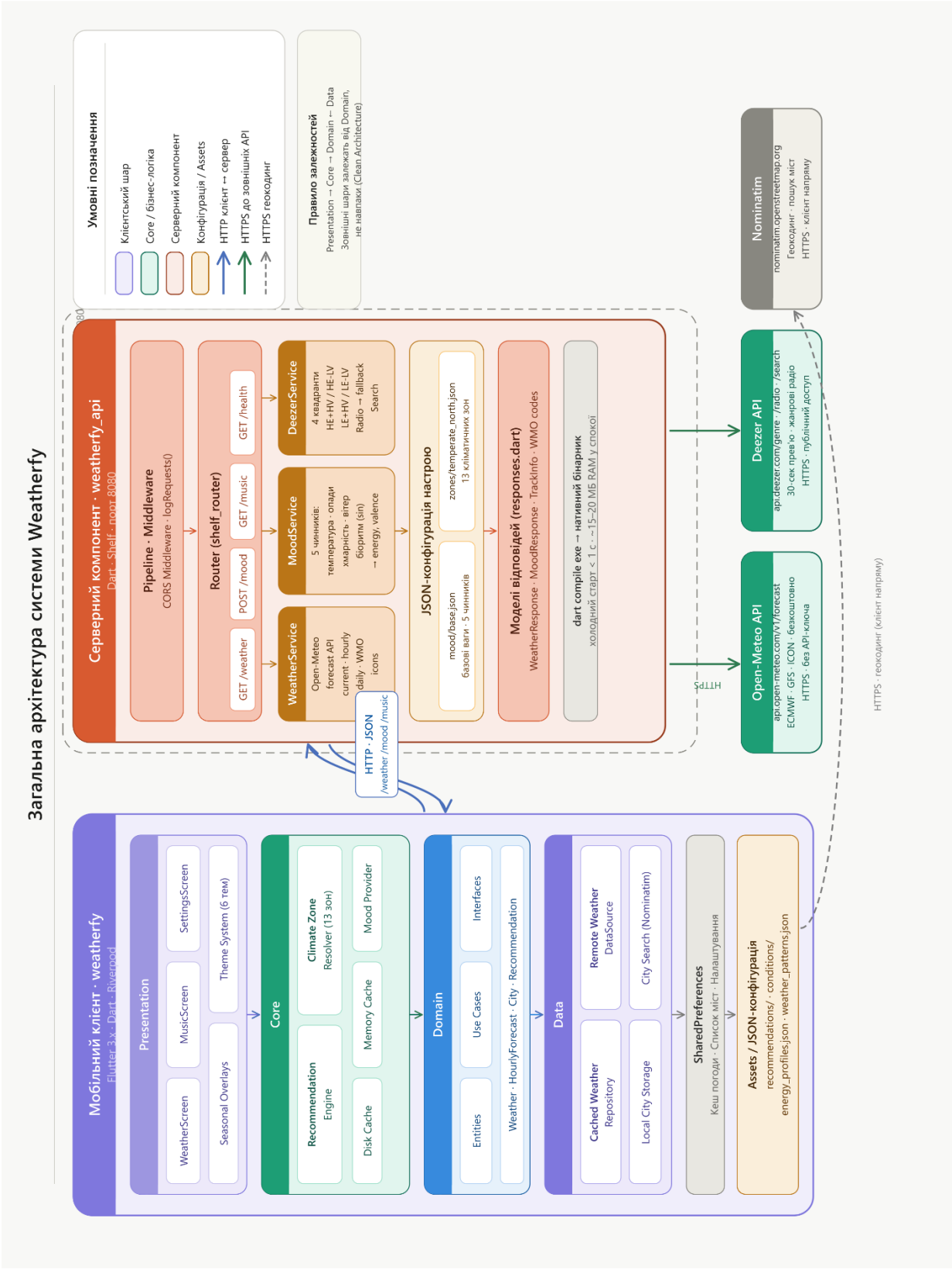


Рисунок А.1. – Загальна архітектура системи Weatherfy

Додаток А.(продовження)

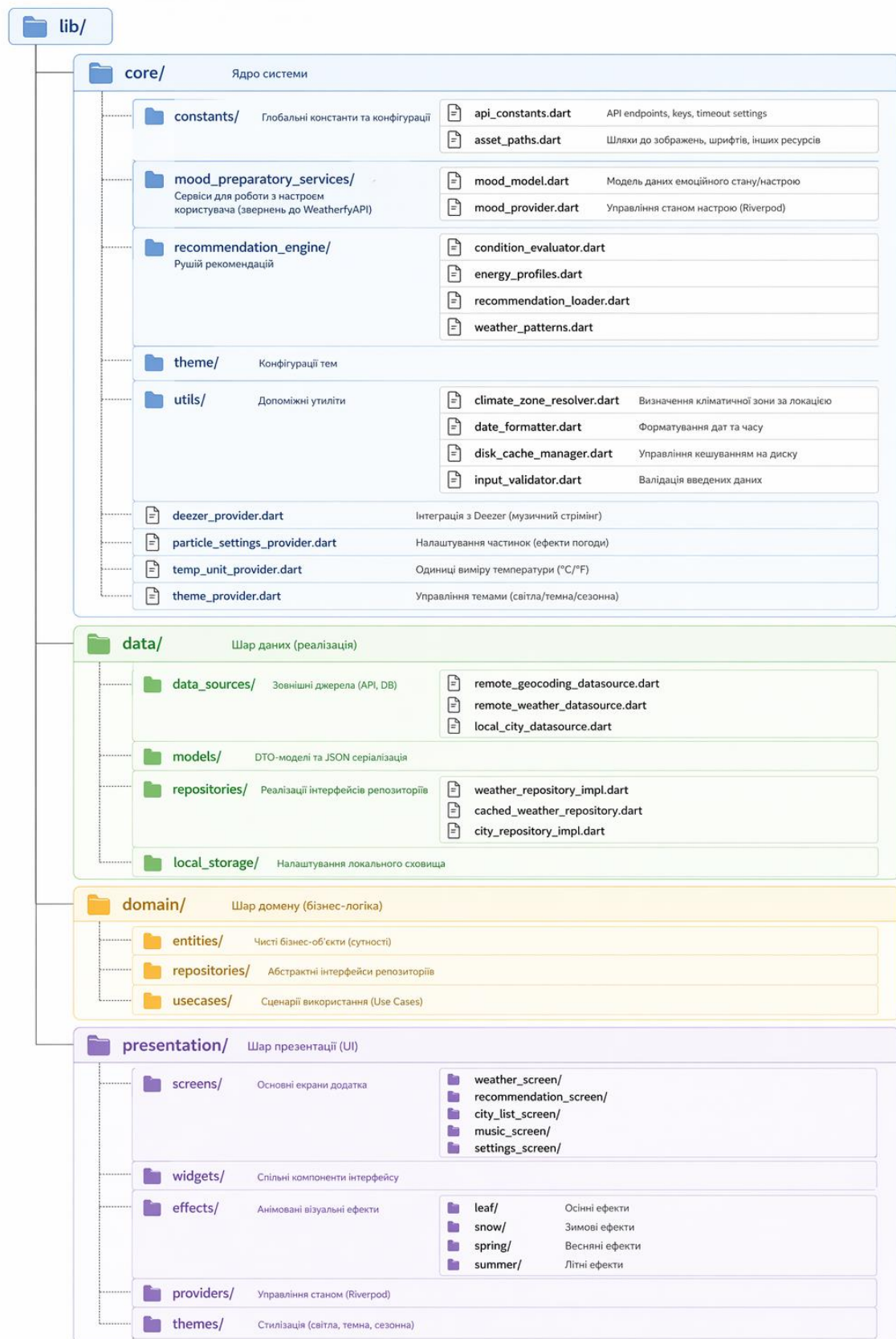


Рисунок А.2. - Структура проєкту клієнтського застосунку – Weatherfy

Додаток А.(продовження)

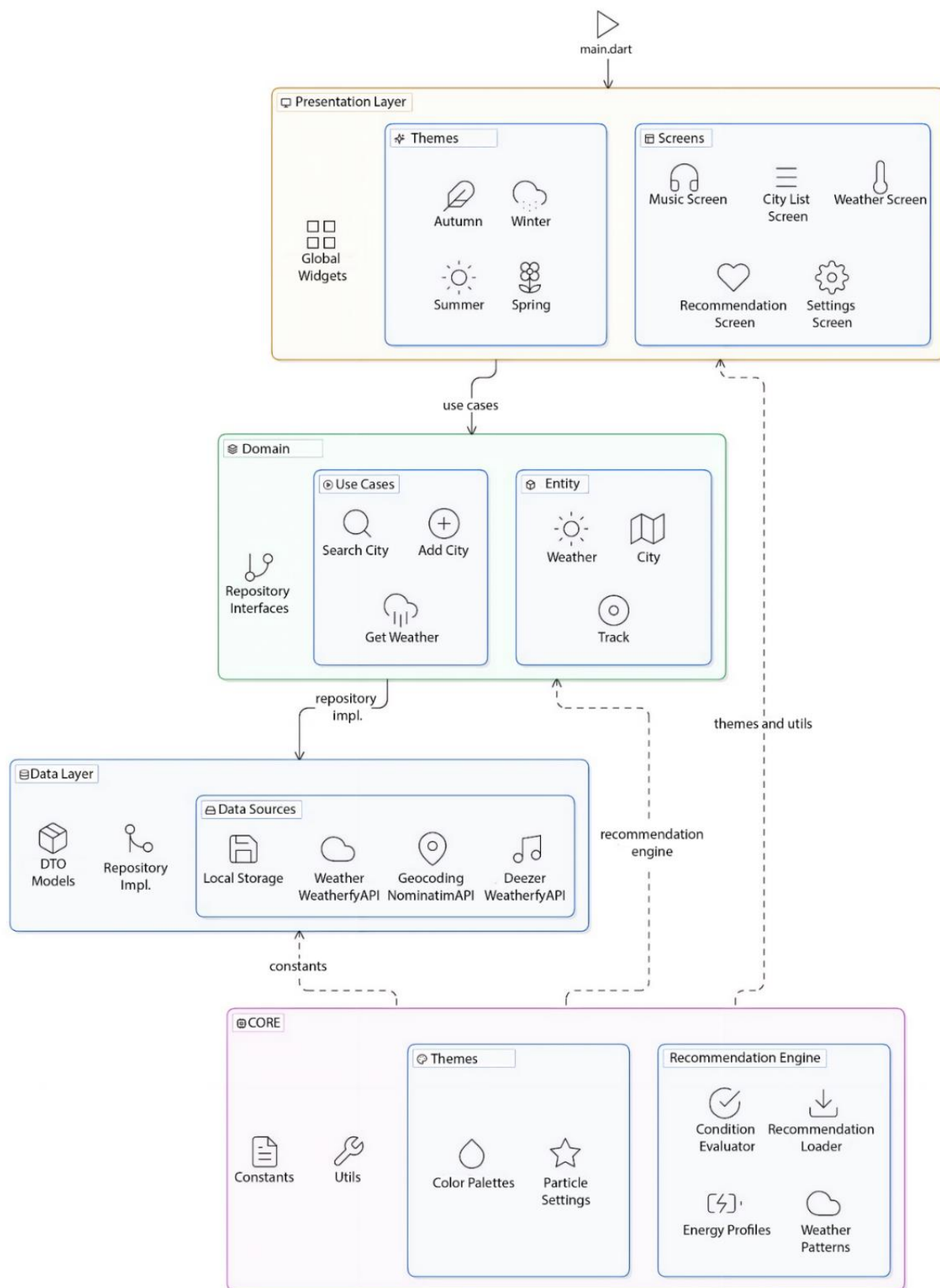


Рисунок А.3. – Архітектурна діаграма компонентів мобільного додатку Weatherfy

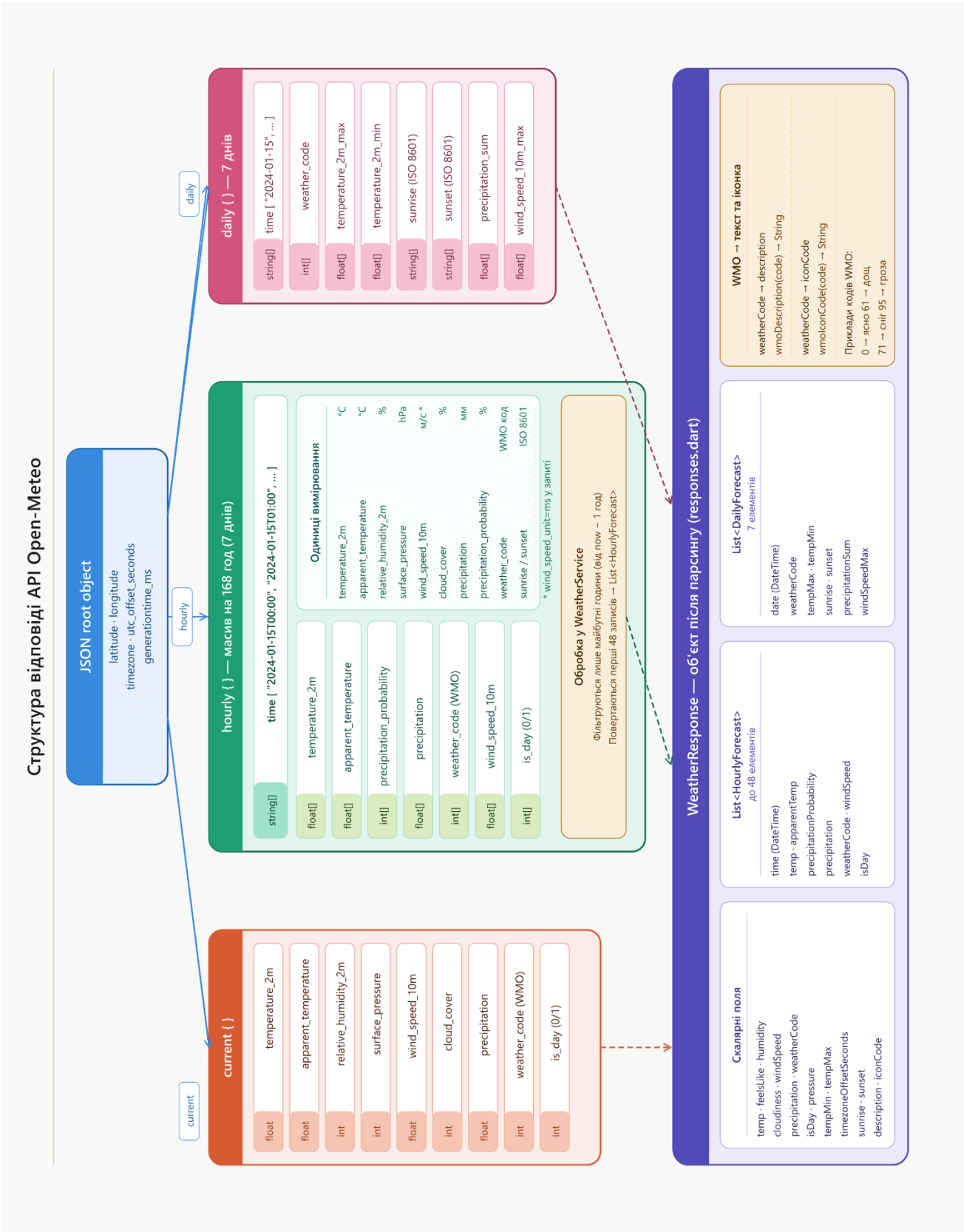


Рисунок Б.1. – Структура відповіді API Open-Meteo

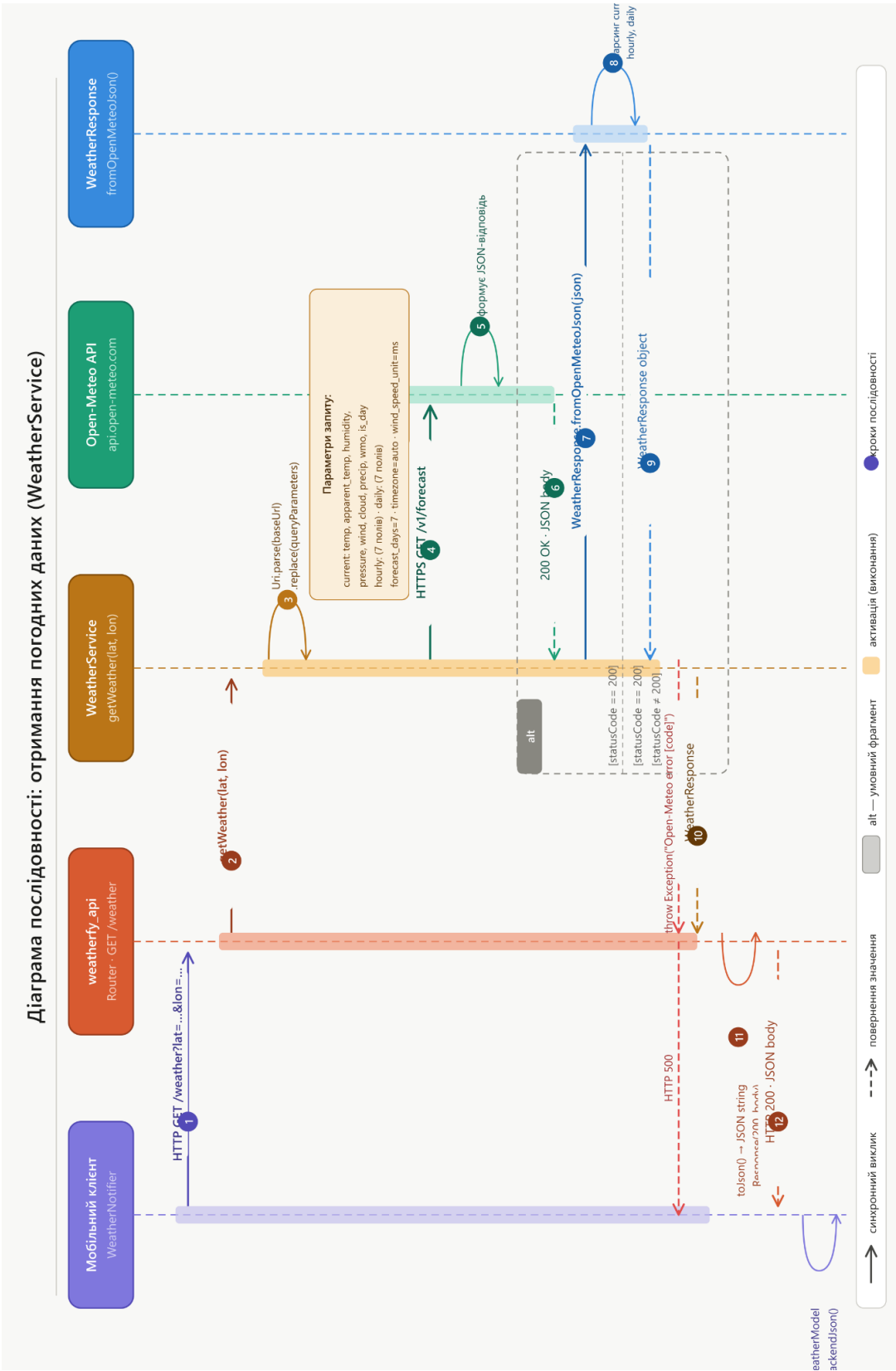


Рисунок Б.2. – Діаграма послідовності: отримання погодних даних

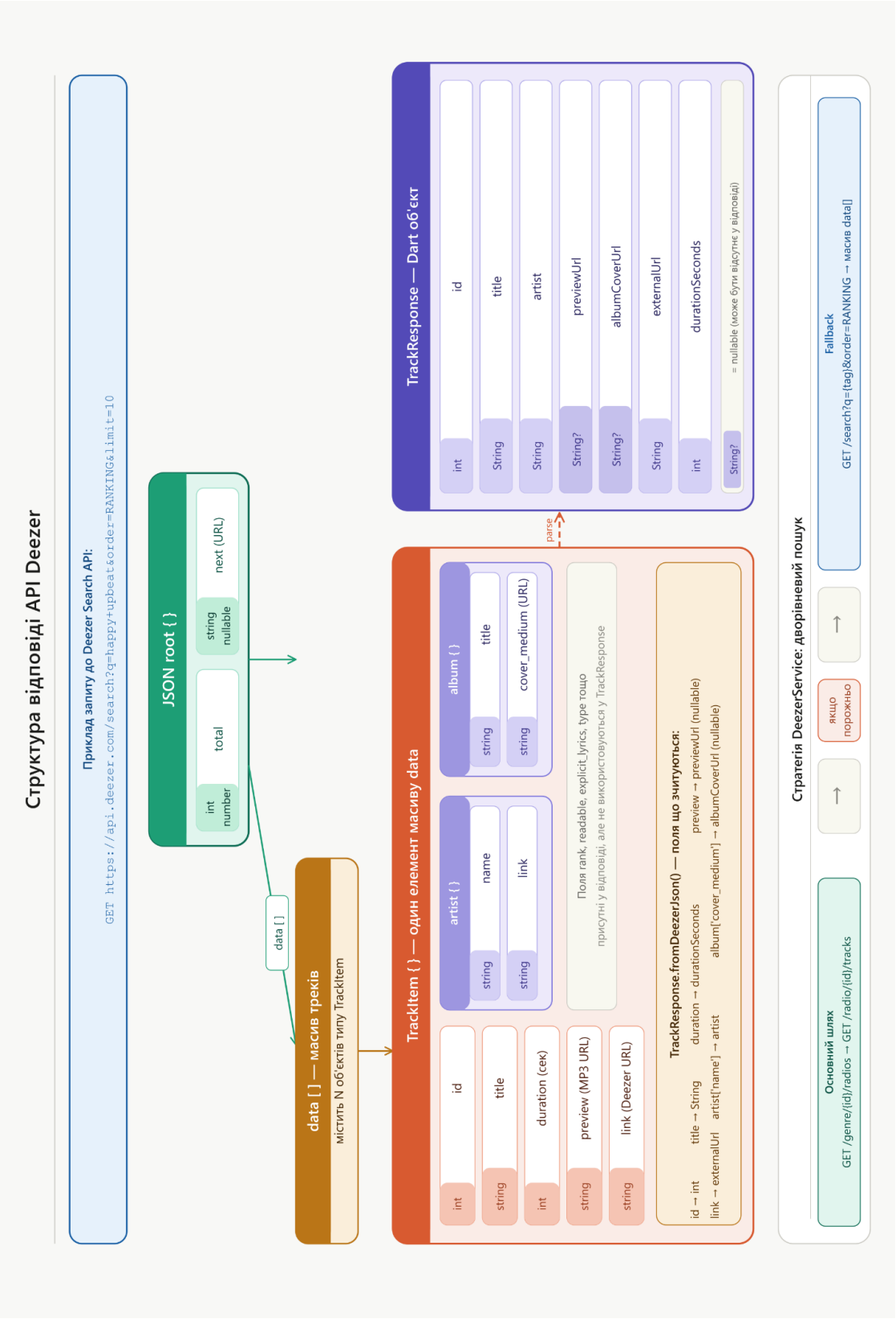


Рисунок Б.3. – Структура відповіді API Deezer

Додаток Б. (продовження)

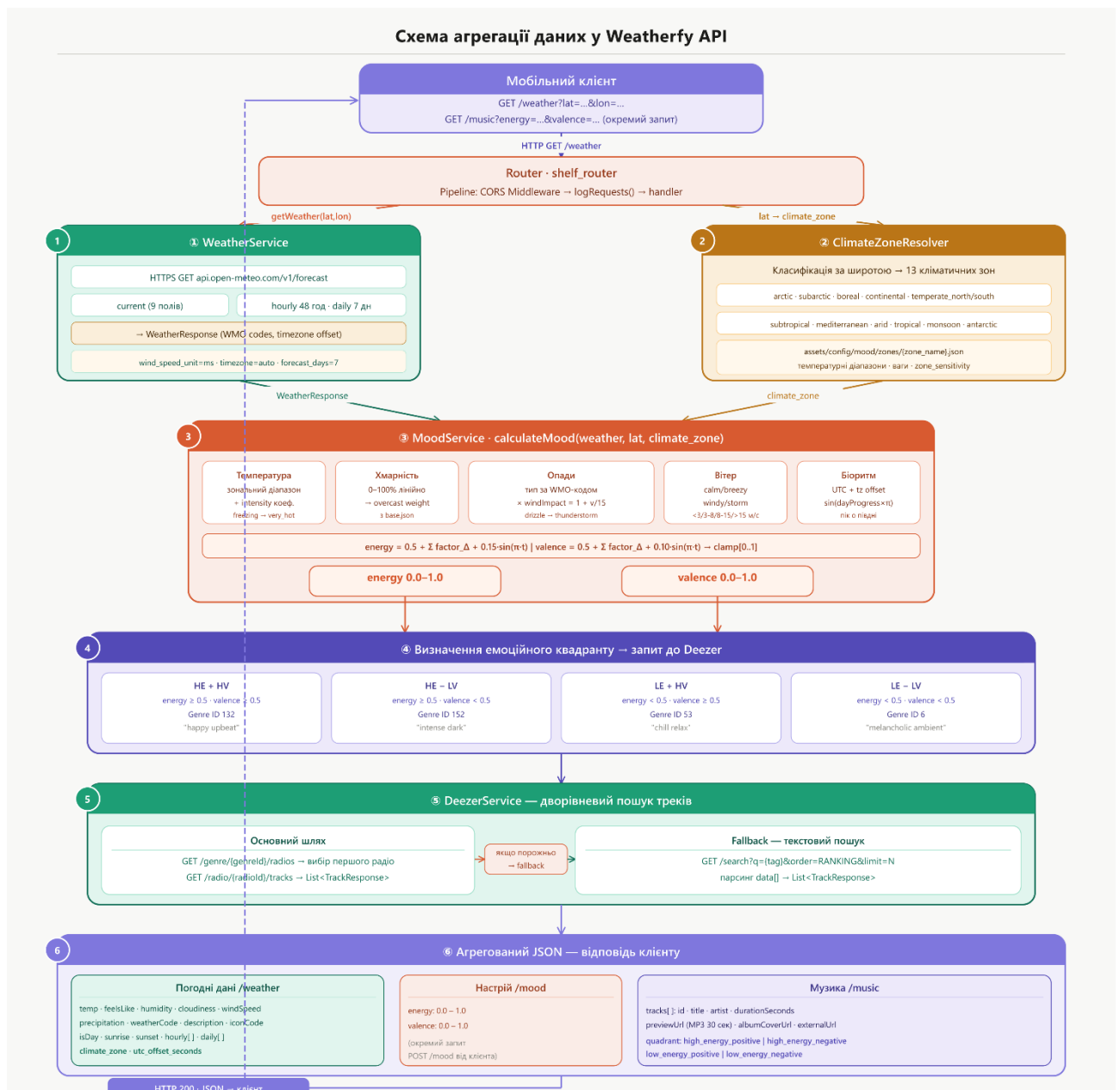


Рисунок Б.4. – Схема агрегації даних у Weatherfy API

Додаток В. Реалізація RecommendationEngine

```
lib > core > recommendation_engine > engine > recommendation_engine.dart > RecommendationEngine
1 // lib/core/recommendation_engine/engine/recommendation_engine.dart
2 import 'package:flutter/material.dart';
3 import 'package:weatherfy/core/recommendation_engine/models/recommendation.dart';
4 import 'package:weatherfy/core/recommendation_engine/models/recommendation_config.dart';
5 import 'package:weatherfy/core/recommendation_engine/models/recommendation_context.dart';
6 import 'package:weatherfy/core/recommendation_engine/models/energy_profile.dart';
7 import 'package:weatherfy/core/recommendation_engine/models/weather_pattern.dart';
8 import 'package:weatherfy/core/recommendation_engine/models/fallback_recommendation.dart';
9 import 'package:weatherfy/core/recommendation_engine/loaders/recommendation_loader.dart';
10
11 class RecommendationEngine {
12   final List<RecommendationConfig> _configs;
13   final List<EnergyProfile> _energyProfiles;
14   // FIX: список паттернів тепер сортується за priority DESC при ініціалізації,
15   // тому _findWeatherPattern завжди повертає найбільш пріоритетний збір
16   final List<WeatherPattern> _weatherPatterns;
17   final String _locale;
18
19   RecommendationContext? _lastContext;
20   List<Recommendation>? _lastResult;
21
22   RecommendationEngine(
23     this._configs,
24     this._energyProfiles,
25     this._weatherPatterns,
26     this._locale,
27   );
28
29   static Future<RecommendationEngine> initialize({String locale = 'ru'}) async {
30     final configs = await RecommendationLoader.loadRecommendations();
31     final energyProfiles = await RecommendationLoader.loadEnergyProfiles();
32     final rawPatterns = await RecommendationLoader.loadWeatherPatterns();
33
34     // FIX: сортуємо паттерни за priority DESC одразу при завантаженні
35     final sortedPatterns = List<WeatherPattern>.from(rawPatterns)
36       ..sort((a, b) => b.priority.compareTo(a.priority));
37
38     return RecommendationEngine(configs, energyProfiles, sortedPatterns, locale);
39   }
40
41   List<Recommendation> generate(RecommendationContext context) {
42     if (_lastContext != null && _contextsEqual(_lastContext!, context)) {
43       return _lastResult!;
44     }
45
46     final energyProfile = _findEnergyProfile(context.energy);
47     final weatherPattern = _findWeatherPattern(context);
48
49     final matched = _configs.where((config) => config.matches(context)).toList();
50
51     List<Recommendation> recommendations;
52
53     if (matched.isEmpty) {
54       recommendations = _generateSmartFallbacks(context, energyProfile, weatherPattern);
55     } else {
56       recommendations = matched.map((config) {
57         var rec = config.toRecommendation(context, _locale);
58         if (weatherPattern != null) {
59           rec = _applyWeatherBoost(rec, weatherPattern);
60         }
61         return rec;
62       }).toList();
63
64       recommendations.sort((a, b) => b.sortValue.compareTo(a.sortValue));
65       recommendations = _diversifyRecommendations(recommendations, maxCount: 5);
66
67       if (recommendations.length < 3) {
68         final fallbacks = _generateSmartFallbacks(
69           context,
70           energyProfile,
71           weatherPattern,
```

```

41 List<Recommendation> generate(RecommendationContext context) {
71     weatherPattern,
72     limit: 3 - recommendations.length,
73     );
74     recommendations.addAll(fallbacks);
75 }
76 }
77
78 _lastContext = context;
79 _lastResult = recommendations;
80
81 return recommendations;
82 }
83
84 EnergyProfile? _findEnergyProfile(double energy) {
85     return _energyProfiles.firstWhere(
86         (profile) => profile.matches(energy),
87         orElse: () => _energyProfiles.first,
88     );
89 }
90
91 // паттерн з найвищим priority завжди буде перевірено першим
92 WeatherPattern? _findWeatherPattern(RecommendationContext context) {
93     for (final pattern in _weatherPatterns) {
94         if (pattern.matches(context)) {
95             return pattern;
96         }
97     }
98     return null;
99 }
100
101 Recommendation _applyWeatherBoost(
102     Recommendation rec,
103     WeatherPattern pattern,
104 ) {
105     final shouldBoost = pattern.boostCategories.contains(rec.category.name);
106     final shouldAvoid = pattern.avoidCategories.contains(rec.category.name);
107
108     if (shouldAvoid) {
109         return Recommendation(
110             id: rec.id,
111             title: rec.title,
112             description: rec.description,
113             icon: rec.icon,
114             color: rec.color,
115             category: rec.category,
116             priority: rec.priority,
117             relevanceScore: rec.relevanceScore * 0.5,
118             debugReason: '${rec.debugReason} + weather_avoid',
119         );
120     }
121
122     if (shouldBoost) {
123         return Recommendation(
124             id: rec.id,
125             title: rec.title,
126             description: rec.description,
127             icon: rec.icon,
128             color: rec.color,
129             category: rec.category,
130             priority: rec.priority,
131             relevanceScore: rec.relevanceScore * pattern.boostMultiplier,
132             debugReason: '${rec.debugReason} + weather_boost($pattern.boostMultiplier)',
133         );
134     }
135
136     return rec;
137 }
138
139 List<Recommendation> _generateSmartFallbacks(

```

```

139 List<Recommendation> _generateSmartFallbacks(
140     RecommendationContext context,
141     EnergyProfile? energyProfile,
142     WeatherPattern? weatherPattern, {
143     int? limit,
144     }) {
145     final fallbacks = <Recommendation>[];
146     final usedCategories = <String>[];
147
148     if (energyProfile != null) {
149         for (final fbRec in energyProfile.fallbackRecommendations) {
150             if (limit != null && fallbacks.length >= limit) break;
151             if (usedCategories.contains(fbRec.category)) continue;
152             fallbacks.add(_createRecommendationFromFallback(fbRec, context));
153             usedCategories.add(fbRec.category);
154         }
155     }
156
157     if (weatherPattern != null) {
158         for (final fbRec in weatherPattern.fallbackRecommendations) {
159             if (limit != null && fallbacks.length >= limit) break;
160             if (usedCategories.contains(fbRec.category)) continue;
161             fallbacks.add(_createRecommendationFromFallback(fbRec, context));
162             usedCategories.add(fbRec.category);
163         }
164     }
165
166     if (fallbacks.isEmpty || (limit != null && fallbacks.length < limit)) {
167         fallbacks.addAll(_generateUniversalFallbacks(context, usedCategories, limit));
168     }
169
170     return fallbacks.take(limit ?? 5).toList();
171 }
172
173 Recommendation _createRecommendationFromFallback(
174     FallbackRecommendation fbRec,
175     RecommendationContext context,
176     ) {
177     return Recommendation(
178         id: 'fallback_${fbRec.category}_${DateTime.now().millisecondsSinceEpoch}',
179         title: fbRec.title[_locale] ?? fbRec.title['en'] ?? fbRec.title.values.first,
180         description: fbRec.description[_locale] ?? fbRec.description['en'] ?? fbRec.description.values.first,
181         icon: RecommendationConfig.parseIcon(fbRec.icon),
182         color: RecommendationConfig.parseColor(fbRec.color),
183         category: RecommendationConfig.parseCategory(fbRec.category),
184         priority: RecommendationPriority.medium,
185         relevanceScore: 0.6,
186         debugReason: 'fallback from profile',
187     );
188 }
189
190 List<Recommendation> _generateUniversalFallbacks(
191     RecommendationContext context,
192     Set<String> usedCategories,
193     int? limit,
194     ) {
195     final universalFallbacks = <Recommendation>[];
196
197     if (!usedCategories.contains('wellness')) {
198         if (context.temperature < 0) {
199             universalFallbacks.add(_createFallback(
200                 'Зігрійтесь',
201                 'Заваріть гарячий чай ☕ загорніться в плед',
202                 Icons.local_cafe,
203                 Colors.orange,
204                 RecommendationCategory.wellness,
205             ));
206             usedCategories.add('wellness');
207         } else if (context.temperature > 28) {
208             universalFallbacks.add(_createFallback(

```

```

208     universalFallbacks.add(_createFallback(
209         'Освіжіться',
210         'Пийте більше води та уникайте прямих сонячних променів',
211         Icons.ac_unit,
212         Colors.blue,
213         RecommendationCategory.wellness,
214     ));
215     usedCategories.add('wellness');
216 }
217 }
218
219 if (!usedCategories.contains('indoor') &&
220     (context.precipitationType == 'rain' || context.precipitationType == 'heavy_rain')) {
221     universalFallbacks.add(_createFallback(
222         'Затишний день вдома',
223         'Ідеальний час для домашніх справ або відпочинку',
224         Icons.home,
225         Colors.brown,
226         RecommendationCategory.indoor,
227     ));
228     usedCategories.add('indoor');
229 }
230
231 if (!usedCategories.contains('activity')) {
232     if (context.energy < 0.3) {
233         universalFallbacks.add(_createFallback(
234             'Відпочиньте',
235             'Дозвольте собі трохи розслабитись',
236             Icons.spa,
237             Colors.purple,
238             RecommendationCategory.wellness,
239         ));
240     } else if (context.energy > 0.7) {
241         universalFallbacks.add(_createFallback(
242             'Будьте активні',
243             'Використайте свою енергію для продуктивних справ',
244             Icons.fitness_center,
245             Colors.green,
246             RecommendationCategory.activity,
247         ));
248     }
249 }
250
251 if (universalFallbacks.isEmpty) {
252     universalFallbacks.add(_createFallback(
253         'Прислухайтеся до себе',
254         'Сьогодні ідеальний день для турботи про себе',
255         Icons.self_improvement,
256         Colors.teal,
257         RecommendationCategory.wellness,
258     ));
259 }
260
261 return universalFallbacks.take(limit ?? 3).toList();
262 }
263
264 Recommendation _createFallback(
265     String title,
266     String description,
267     IconData icon,
268     Color color,
269     RecommendationCategory category,
270 ) {
271     return Recommendation(
272         id: 'fallback_universal_${title.hashCode}_${DateTime.now().millisecondsSinceEpoch}',
273         title: title,
274         description: description,
275         icon: icon,
276         color: color,

```

```

264 Recommendation _createFallback(
275     color: color,
276     category: category,
277     priority: RecommendationPriority.medium,
278     relevanceScore: 0.5,
279     debugReason: 'universal fallback',
280 );
281 }
282 }
283
284 List<Recommendation> _diversifyRecommendations(
285     List<Recommendation> recommendations, {
286         required int maxCount,
287     }) {
288     final result = <Recommendation>[];
289     final usedCategories = <RecommendationCategory>{};
290
291     for (final rec in recommendations) {
292         if (!usedCategories.contains(rec.category)) {
293             result.add(rec);
294             usedCategories.add(rec.category);
295             if (result.length >= maxCount) return result;
296         }
297     }
298
299     for (final rec in recommendations) {
300         if (!result.contains(rec)) {
301             result.add(rec);
302             if (result.length >= maxCount) return result;
303         }
304     }
305
306     return result;
307 }
308
309 bool _contextsEqual(RecommendationContext a, RecommendationContext b) {
310     return a.energy == b.energy &&
311         a.valence == b.valence &&
312         a.temperature == b.temperature &&
313         a.precipitationType == b.precipitationType &&
314         a.precipitationAmount == b.precipitationAmount &&
315         a.windSpeed == b.windSpeed &&
316         a.cloudiness == b.cloudiness &&
317         a.isDaytime == b.isDaytime &&
318         a.season == b.season;
319 }
320
321 List<Map<String, dynamic>> debugAllMatches(RecommendationContext context) {
322     final energyProfile = _findEnergyProfile(context.energy);
323     final weatherPattern = _findWeatherPattern(context);
324
325     return [
326         {
327             'type': 'context',
328             'energy_profile': energyProfile?.id ?? 'none',
329             'weather_pattern': weatherPattern?.id ?? 'none',
330             // FIX: тепер показуємо priority паттерну для ясності при дебагу
331             'weather_pattern_priority': weatherPattern?.priority ?? 0,
332             'context': context.toString(),
333         },
334         ..._configs.map((config) {
335             final matches = config.matches(context);
336             return {
337                 'type': 'config',
338                 'id': config.id,
339                 'category': config.categoryKey,
340                 'matches': matches,
341                 'reason': matches ? config.conditions.debugReason(context) : 'no match',
342             };
343         })),
344     ];

```

Додаток Г. Реалізація MoodService

```
lib > services > mood_service.dart > MoodService

1 import 'dart:convert';
2 import 'dart:io';
3 import 'dart:math' as math;
4 import 'package:path/path.dart' as path;
5 import 'package:weatherfy_api/models/responses.dart';
6
7 class MoodService {
8   final String _assetsPath;
9
10  MoodService({String? assetsPath})
11    : _assetsPath = assetsPath ?? _resolveAssetsPath() {
12    print('⚡ MoodService ініціалізовано. Шлях: $_assetsPath');
13  }
14
15  /// Визначає шлях до директорії з ресурсами (assets)
16  static String _resolveAssetsPath() {
17    try {
18      final scriptDir = path.dirname(Platform.script.toFilePath());
19      // Перевіряємо кілька варіантів розташування assets (Docker vs Local)
20      final candidates = [
21        path.join(Directory.current.path, 'assets', 'config', 'mood'),
22        path.join(scriptDir, '..', 'assets', 'config', 'mood'),
23        path.join(scriptDir, 'assets', 'config', 'mood'),
24      ];
25
26      for (final candidate in candidates) {
27        final normalized = path.normalize(candidate);
28        if (Directory(normalized).existsSync()) {
29          return normalized;
30        }
31      }
32      throw Exception("Директорію assets не знайдено за шляхами: $candidates");
33    } catch (e) {
34      print('❌ Помилка визначення шляху: $e');
35      // Повернення до поточної директорії у разі помилки
36      return path.join(Directory.current.path, 'assets', 'config', 'mood');
37    }
38  }
39
40  /// Головний метод розрахунку настрою (Energy та Valence)
41  Future<MoodResponse> calculateMood({
42    required WeatherResponse weather,
43    required double latitude,
44    required String climateZone,
45  }) async {
46    try {
47      // 1. Завантаження конфігурацій
48      final baseConfig = await _loadJson('base.json');
49
50      // Нормалізуємо назву зони: "Temperate North" -> "temperate_north.json"
51      final normalizedZone = climateZone.toLowerCase().trim().replaceAll(' ', '_');
52      final zoneConfig = await _loadJson('zones/$normalizedZone.json');
53
54      // 2. Початкові значення
55      double energy = _toDouble(baseConfig['initial']?['energy'] ?? 0.5);
56      double valence = _toDouble(baseConfig['initial']?['valence'] ?? 0.5);
57
58      final factors = baseConfig['factors'] as Map<String, dynamic>? ?? {};
59      final sensitivity = zoneConfig['zone_sensitivity'] as Map<String, dynamic>? ?? {};
60
61      // --- РОЗРАХУНОК ФАКТОРІВ ---
62
63      // 🌡 Температурна
64      final tempCategory = _getTemperatureCategory(weather.temp, zoneConfig);
65      final tempWeights = (factors['temperature'] as Map<String, dynamic>?)?[tempCategory];
66
67      if (tempWeights != null) {
68        final weightMap = tempWeights as Map<String, dynamic>;
69        final zoneFactor = _toDouble(sensitivity['temperature']?[tempCategory] ?? 1.0);
70
71        // Плавний розрахунок відхилення всередині діапазону (якщо є дані)
```

```

lib > services > mood_service.dart > MoodService
7 class MoodService {
41   Future<MoodResponse> calculateMood({
71     // Плавний розрахунок відхилення всередині діапазону (якщо є дані)
72     final ranges = zoneConfig['temperature_ranges'] as Map<String, dynamic>;
73     double intensity = 1.0;
74     if (ranges != null && ranges.containsKey(tempCategory)) {
75       final r = ranges[tempCategory] as List;
76       final mid = (_toDouble(r[0]) + _toDouble(r[1])) / 2;
77       final span = (_toDouble(r[1]) - _toDouble(r[0])) / 2;
78       if (span > 0) intensity = 1.0 - ((weather.temp - mid).abs() / span).clamp(0.0, 0.5);
79     }
80
81     energy += _toDouble(weightMap['energy']) * zoneFactor * intensity;
82     valence += _toDouble(weightMap['valence']) * zoneFactor * intensity;
83   }
84
85   // 2. Хмарність
86   final cloudFactor = (weather.cloudiness / 100.0).clamp(0.0, 1.0);
87   final cloudWeights = factors['cloudiness']?['overcast'] as Map<String, dynamic>;
88   if (cloudWeights != null) {
89     energy += cloudFactor * _toDouble(cloudWeights['energy']);
90     valence += cloudFactor * _toDouble(cloudWeights['valence']);
91   }
92
93   // 3. Опави
94   final precipType = _getPrecipitationType(weather.description, weather.weatherCode);
95   final precipWeights = (factors['precipitation'] as Map<String, dynamic>)?[precipType] as Map<String, dynamic>;
96
97   if (precipWeights != null) {
98     final zoneFactor = _toDouble(sensitivity['precipitation']?[precipType] ?? 1.0);
99     // Вітер посилює дискомфорт від опадів
100     final windImpact = 1.0 + (weather.windSpeed / 15.0).clamp(0.0, 0.5);
101
102     energy += _toDouble(precipWeights['energy']) * zoneFactor * windImpact;
103     valence += _toDouble(precipWeights['valence']) * zoneFactor * windImpact;
104   }
105
106   // 4. Вітер
107   final windWeightsMap = factors['wind'] as Map<String, dynamic>;
108   if (windWeightsMap != null) {
109     final ws = weather.windSpeed;
110     String windCat = ws < 3 ? 'calm' : ws < 8 ? 'breezy' : ws < 15 ? 'windy' : 'storm';
111     final w = windWeightsMap[windCat] as Map<String, dynamic>;
112     if (w != null) {
113       energy += _toDouble(w['energy']);
114       valence += _toDouble(w['valence']);
115     }
116   }
117
118   // 5. Час доби (біоритми)
119   final now = DateTime.now().toUtc();
120   // Враховуємо зміщення часового поясу конкретного міста
121   final localNow = now.add(Duration(seconds: weather.timezoneOffsetSeconds));
122   final dayProgress = _getDayProgress(localNow, weather.sunrise, weather.sunset);
123
124   // Пік енергії вдень (синусоїда)
125   energy += 0.15 * math.sin(dayProgress * math.pi);
126   valence += 0.10 * math.sin(dayProgress * math.pi);
127
128   // 6. Фінальна нормалізація результатів
129   energy = energy.clamp(0.0, 1.0);
130   valence = valence.clamp(0.0, 1.0);
131
132   print('✅ Настрій для $climateZone: E:${energy.toStringAsFixed(2)} V:${valence.toStringAsFixed(2)}');
133   return MoodResponse(energy: energy, valence: valence);
134
135 } catch (e, stack) {
136   print('❌ Критична помилка calculateMood: $e');
137   print(stack);
138   // Повертаємо нейтральний стан замість помилки 500, якщо щось пішло не так
139   return MoodResponse(energy: 0.5, valence: 0.5);

```

```

lib > services > mood_service.dart > MoodService
7   class MoodService {
41   Future<MoodResponse> calculateMood({
136     print('✖ Критична помилка calculateMood: $e');
137     print(stack);
138     // Повертаємо нейтральний стан замість помилки 500, якщо щось пішло не так
139     return MoodResponse(energy: 0.5, valence: 0.5);
140   }
141 }
142
143 // --- ДОПОМІЖНІ МЕТОДИ ---
144
145 /// Завантажує та декодує JSON файл за відносним шляхом
146 FutureMap<String, dynamic> _loadJson(String relativePath) async {
147   final fullPath = path.normalize(path.join(_assetsPath, relativePath));
148   final file = File(fullPath);
149
150   if (!await file.exists()) {
151     print('⚠ Файл відсутній: $fullPath. Використовується порожній конфіг.');
```

Додаток Д. Реалізація DiskCacheManager

```
lib > core > utils > disk_cache_manager.dart > ...
1  import 'dart:convert';
2  import 'package:shared_preferences/shared_preferences.dart';
3  import 'package:weatherfy/domain/entities/weather.dart';
4
5  class DiskCacheManager {
6    static const String _cachePrefix = 'weather_cache_';
7
8    //  ЄДИНИЙ TTL = 10 хвилин для всієї системи
9    static const Duration cacheDuration = Duration(minutes: 10);
10
11    String _key(String cityId) => '_cachePrefix$cityId';
12
13    /// Отримання погоди + інформація про актуальність.
14    /// ВАЖЛИВО: НІКОЛИ не видаляє дані автоматично!
15    Future<CachedWeatherResult?> getCachedWeather(String cityId) async {
16      final prefs = await SharedPreferences.getInstance();
17      final raw = prefs.getString(_key(cityId));
18      if (raw == null) return null;
19
20      try {
21        final map = json.decode(raw) as Map<String, dynamic>;
22        final timestamp = DateTime.fromMillisecondsSinceEpoch(map['timestamp'] as int);
23        final weather = Weather.fromJson(map['weather'] as Map<String, dynamic>);
24
25        final age = DateTime.now().difference(timestamp);
26        final isFresh = age <= cacheDuration;
27
28        return CachedWeatherResult(
29          weather: weather,
30          isFresh: isFresh,
31          savedAt: timestamp,
32          age: age,
33        );
34      } catch (e) {
35        print('⚠️ Помилка читання кешу для $cityId: $e');
36        // У разі пошкодження даних — очищуємо запис
37        await removeWeather(cityId);
38        return null;
39      }
40    }
41
42    /// Збереження погоди.
43    /// ВАЖЛИВО: Перезаписує старий кеш тільки при успішному збереженні нового.
44    Future<void> saveWeather(String cityId, Weather weather) async {
45      final prefs = await SharedPreferences.getInstance();
46
47      try {
48        await prefs.setString(
49          _key(cityId),
50          json.encode({
51            'timestamp': DateTime.now().millisecondsSinceEpoch,
52            'weather': weather.toJson(),
53          }),
54        );
55        print('📁 Кеш збережено для $cityId');
56      } catch (e) {
57        print('❌ Помилка збереження кешу для $cityId: $e');
58        // Якщо не вдалося зберегти, старий кеш залишається недоторканим
59      }
60    }
61
62    /// Видалення кешу конкретного міста (тільки за явним запитом)
63    Future<void> removeWeather(String cityId) async {
64      final prefs = await SharedPreferences.getInstance();
65      await prefs.remove(_key(cityId));
66      print('🗑️ Кеш видалено для $cityId');
67    }
68
69    /// Інформація по одному запису кешу (для debug)
70    Future<CacheInfo?> getCacheInfo(String cityId) async {
71      final prefs = await SharedPreferences.getInstance();
72    }
```

```

70 Future<CacheInfo> getCacheInfo(String cityId) async {
71   final prefs = await SharedPreferences.getInstance();
72   final raw = prefs.getString(_key(cityId));
73   if (raw == null) return null;
74
75   try {
76     final map = json.decode(raw) as Map<String, dynamic>;
77     final savedAt = DateTime.fromMillisecondsSinceEpoch(map['timestamp'] as int);
78     final age = DateTime.now().difference(savedAt);
79     final remaining = cacheDuration - age;
80
81     return CacheInfo(
82       key: cityId,
83       savedAt: savedAt,
84       age: age,
85       remainingTime: remaining > Duration.zero ? remaining : Duration.zero,
86       isExpired: age > cacheDuration,
87     );
88   } catch (e) {
89     return null;
90   }
91 }
92
93 /// Вся інформація по кешу (для debug-екрана)
94 Future<List<CacheInfo>> getAllCacheInfo() async {
95   final prefs = await SharedPreferences.getInstance();
96
97   final keys = prefs.getKeys()
98     .where((k) => k.startsWith(_cachePrefix))
99     .map((k) => k.replaceFirst(_cachePrefix, ''))
100    .toList();
101
102   final List<CacheInfo> infos = [];
103
104   for (final key in keys) {
105     final info = await getCacheInfo(key);
106     if (info != null) infos.add(info);
107   }
108
109   return infos;
110 }
111
112 /// Повне очищення кешу (для Налаштувань)
113 Future<void> clearAllCache() async {
114   final prefs = await SharedPreferences.getInstance();
115   final keys = prefs.getKeys().where((k) => k.startsWith(_cachePrefix));
116
117   for (final key in keys) {
118     await prefs.remove(key);
119   }
120
121   print('❏ Весь кеш очищено');
122 }
123
124 /// Перевіряє, чи свіжий кеш (без завантаження всіх даних)
125 Future<bool> isWeatherFresh(String cityId) async {
126   final prefs = await SharedPreferences.getInstance();
127   final raw = prefs.getString(_key(cityId));
128   if (raw == null) return false;
129
130   try {
131     final map = json.decode(raw) as Map<String, dynamic>;
132     final timestamp = DateTime.fromMillisecondsSinceEpoch(map['timestamp'] as int);
133     final age = DateTime.now().difference(timestamp);
134     return age <= cacheDuration;
135   } catch (e) {
136     return false;
137   }
138 }
139 }
140

```

```

125     Future<bool> isWeatherFresh(String cityId) async {
132         final timestamp = DateTime.fromMillisecondsSinceEpoch(map['timestamp'] as int);
133         final age = DateTime.now().difference(timestamp);
134         return age <= cacheDuration;
135     } catch (e) {
136         return false;
137     }
138 }
139 }
140
141 class CacheInfo {
142     final String key;
143     final DateTime savedAt;
144     final Duration age;
145     final Duration remainingTime;
146     final bool isExpired;
147
148     CacheInfo({
149         required this.key,
150         required this.savedAt,
151         required this.age,
152         required this.remainingTime,
153         required this.isExpired,
154     });
155
156     String get ageFormatted {
157         final minutes = age.inMinutes;
158         final seconds = age.inSeconds % 60;
159         return '${minutes}x${seconds}';
160     }
161
162     String get remainingTimeFormatted {
163         final minutes = remainingTime.inMinutes;
164         final seconds = remainingTime.inSeconds % 60;
165         return '${minutes}x${seconds}';
166     }
167 }
168
169 class CachedWeatherResult {
170     final Weather weather;
171     final bool isFresh;
172     final DateTime savedAt;
173     final Duration age;
174
175     const CachedWeatherResult({
176         required this.weather,
177         required this.isFresh,
178         required this.savedAt,
179         required this.age,
180     });
181
182     /// Повертає true, якщо кеш застарів (> 10 хвилин)
183     bool get isExpired => !isFresh;
184
185     /// Повертає час, що залишився до закінчення терміну дії
186     Duration get remainingTime {
187         final remaining = DiskCacheManager.cacheDuration - age;
188         return remaining > Duration.zero ? remaining : Duration.zero;
189     }
190 }

```

Додаток Е. Реалізація WeatherNotifier / weather_provider.dart

```
lib > presentation > weather_screen > providers > weather_provider.dart > ...
1  import 'dart:async';
2  import 'dart:io';
3
4  import 'package:flutter_riverpod/flutter_riverpod.dart';
5  import 'package:weatherfy/domain/entities/weather.dart';
6  import 'package:weatherfy/presentation/repository_providers.dart';
7
8  // Тип для параметрів (Record)
9  typedef WeatherArgs = ({String cityId, double lat, double lon});
10
11  /// ✅ Потік "тікає" щохвилини для перевірки терміну дії кешу
12  /// Але реальне оновлення відбувається лише тоді, коли кеш застарів (> 10 хвилин)
13  final weatherSyncPulseProvider = StreamProvider<DateTime>((ref) {
14    return Stream.periodic(const Duration(minutes: 1), (_) => DateTime.now());
15  }); // StreamProvider
16
17  final weatherNotifierProvider =
18  AsyncNotifierProvider.autoDispose.family<WeatherNotifier, Weather, WeatherArgs>(
19    (ref, args) => WeatherNotifier(),
20  );
21
22  class WeatherNotifier extends AutoDisposeFamilyAsyncNotifier<Weather, WeatherArgs> {
23    int _errorCount = 0;
24    bool _isOfflineMessageLogged = false;
25    bool _isErrorMessageLogged = false;
26    Timer? _retryTimer; // Швидкий retry при офлайн
27
28    @override
29    FutureOr<Weather> build(WeatherArgs arg) {
30      // Утримуємо стан у пам'яті
31      ref.keepAlive();
32
33      // Скасовуємо таймер при dispose
34      ref.onDispose(() {
35        _retryTimer?.cancel();
36      });
37
38      // Слухаємо глобальний пульс (тікає щохвилини)
39      ref.listen(weatherSyncPulseProvider, (previous, next) {
40        _onSyncTick();
41      });
42
43      return _fetchWeather(forceRefresh: false);
44    }
45
46    /// Реакція на хвилинний тік пульсу
47    void _onSyncTick() async {
48      if (state.isLoading || !state.hasValue) return;
49
50      // ✅ Отримуємо інформацію про кеш із надійного джерела
51      final cacheInfo = await ref.read(diskCacheManagerProvider).getCacheInfo(arg.cityId);
52
53      // Якщо кешу немає або він свіжий – нічого не робимо
54      if (cacheInfo == null || !cacheInfo.isExpired) return;
55
56      print('🔴 [${arg.cityId}] Кеш застарів – оновлюємо');
57
58      // Експоненціальний відкат (backoff) при помилках
59      if (_errorCount > 0) {
60        final retryMinutes = _errorCount <= 3 ? _errorCount : 15;
61        if (cacheInfo.age.inMinutes < retryMinutes) {
62          print('🔴 [${arg.cityId}] Чекаємо на retry через ${retryMinutes - cacheInfo.age.inMinutes}хв');
63          return;
64        }
65      }
66
67      _performUpdate();
68    }
69
70    /// Запускаємо швидкий retry-таймер при офлайн (кожні 15 секунд)
71    void _startRetryTimer() {
```

```

22 class WeatherNotifier extends AutoDisposeFamilyAsyncNotifier<Weather, WeatherArgs> {
70 // Запускаємо швидкий retry-таймер при офлайні (кожні 15 секунд)
71 void _startRetryTimer() {
72   _retryTimer?.cancel();
73
74   print('🔁 [${arg.cityId}] Запускаємо retry-таймер (кожні 15 сек)');
75   _retryTimer = Timer.periodic(const Duration(seconds: 15), (_) {
76     if (!state.isLoading) {
77       print('🔁 [${arg.cityId}] Спроба retry...');
78       _performUpdate();
79     }
80   }); // Timer.periodic
81 }
82
83 Future<void> _performUpdate() async {
84   print('🔄 [${arg.cityId}] Починаємо оновлення даних...');
85
86   state = const AsyncLoading();
87
88   try {
89     final weather = await _fetchWeather(forceRefresh: false);
90     state = AsyncValue.data(weather);
91     print('✅ [${arg.cityId}] Оновлення завершено');
92   } catch (error, stack) {
93     // Якщо є старі дані – показуємо їх
94     if (state.hasValue) {
95       print('⚠️ [${arg.cityId}] Помилка оновлення, використовуємо кеш');
96       state = AsyncValue.data(state.value!);
97     } else {
98       state = AsyncValue.error(error, stack);
99     }
100   }
101 }
102
103 Future<Weather> _fetchWeather({required bool forceRefresh}) async {
104   final repository = ref.read(weatherRepositoryProvider);
105
106   try {
107     final weather = await repository.getWeatherByCoords(
108       cityId: arg.cityId,
109       lat: arg.lat,
110       lon: arg.lon,
111       forceRefresh: forceRefresh,
112     );
113
114     _logSuccess();
115
116     // ✅ Зупиняємо retry-таймер при успіху
117     _retryTimer?.cancel();
118     _retryTimer = null;
119
120     _errorCount = 0;
121     _isOfflineMessageLogged = false;
122     _isErrorMessageLogged = false;
123
124     return weather;
125   } catch (error, stack) {
126     _errorCount++;
127
128     // ✅ Запускаємо швидкий retry при офлайні
129     if (error is SocketException || error.toString().contains('Connectivity')) {
130       if (_retryTimer == null || !_retryTimer!.isActive) {
131         _startRetryTimer();
132       }
133     }
134
135     // Якщо дані вже є у стейті, повертаємо їх (silent error mode)
136     if (state.hasValue) {
137       _logErrorSilently(error);
138       return state.value!;
139     }
140   }

```

```

117     _retryTimer?.cancel();
118     _retryTimer = null;
119
120     _errorCount = 0;
121     _isOfflineMessageLogged = false;
122     _isErrorMessageLogged = false;
123
124     return weather;
125 } catch (error, stack) {
126     _errorCount++;
127
128     // ✅ Запускаємо швидкий retry при офлайн
129     if (error is SocketException || error.toString().contains('Connectivity')) {
130         if (_retryTimer == null || !_retryTimer!.isActive) {
131             _startRetryTimer();
132         }
133     }
134
135     // Якщо дані вже є у стейті, повертаємо їх (silent error mode)
136     if (state.hasValue) {
137         _logErrorSilently(error);
138         return state.value!;
139     }
140
141     // Якщо даних немає (перший запуск), прокидаємо помилку в UI
142     Error.throwWithStackTrace(error, stack);
143 }
144 }
145
146 /// Публічний метод для ручного оновлення (кнопка Refresh)
147 Future<void> refresh() async {
148     print('🔄 [${arg.cityId}] Ручне оновлення');
149
150     _isOfflineMessageLogged = false;
151     _isErrorMessageLogged = false;
152     _retryTimer?.cancel();
153
154     state = const AsyncLoading();
155     state = await AsyncValue.guard(() => _fetchWeather(forceRefresh: true));
156 }
157
158 /// --- Допоміжні методи логуювання ---
159
160 void _logSuccess() {
161     if (_isOfflineMessageLogged || _isErrorMessageLogged) {
162         print('✅ [${arg.cityId}] Зв'язок відновлено. Дані актуальні.');
```

Додаток Є. Макети екранів застосунку Weatherfy

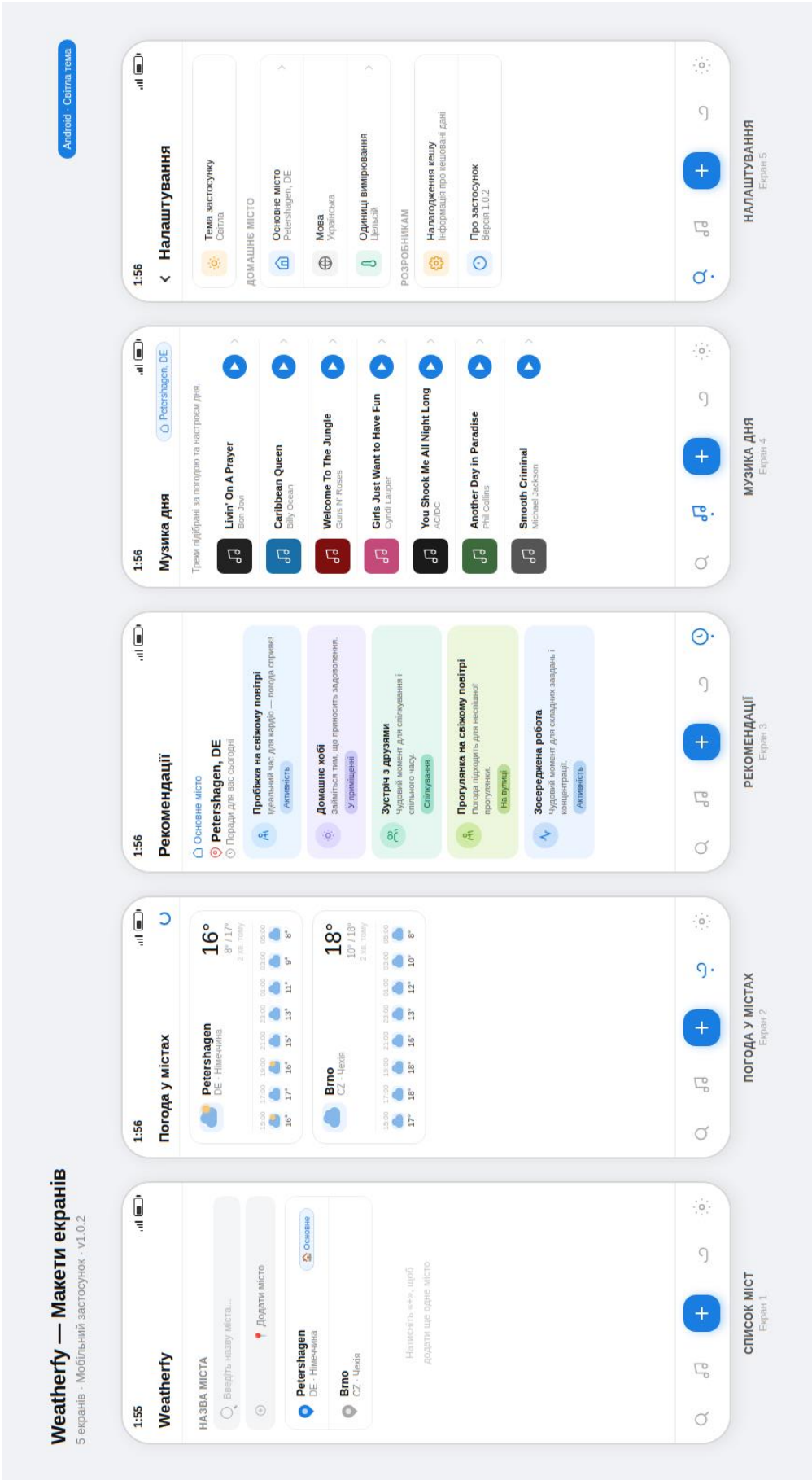


Рисунок Є.1. – Макети екранів застосунку Weatherfy

CERTIFICATE OF PARTICIPATION AND PUBLICATION



Vadym Demianou



participated in the XI Correspondence International
Scientific and Practical Conference

**Globalization of Scientific Knowledge:
International Cooperation and Integration Of Sciences**

held on May 15th, 2026 by

NGO European Scientific Platform (Vinnytsia, Ukraine)
LLC International Centre Corporate Management (Vienna, Austria)

and published scientific paper

**INTEGRATION OF MOOD ENGINE AND RECOMMENDATION ENGINE IN THE HYBRID
CLIENT-SERVER ARCHITECTURE WEATHERFY**

in Periodical scientific journal «**GRAIL OF SCIENCE**»

№ 68; ISSN 2710-3056; Media identifier R30-02704;
DOI 10.36074/grail-of-science.15.05.2026



0.8 ECTS credits (24 hours)

Recommended by the Academic Council of the «Institute
of Scientific and Technical Integration and Cooperation».
Protocol № 17 from May 14th, 2026.

Head of the
NGO «European Scientific Platform»
Chairman of the Organizing committee
GOLDENBLAT MIRIAM



Head of Community Outreach at the
LLC «International Centre
Corporate Management»
RACHAEL APARO



