МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЧЕРКАСЬКИЙ ДЕРЖАВНИЙ ФАХОВИЙ БІЗНЕС-КОЛЕДЖ
Циклова комісія комп’ютерної інженерії та інформаційних технологій

КВАЛІФІКАЦІЙНА РОБОТА

на тему

МОБІЛЬНИЙ ЗАСТОСУНОК ДЛЯ ПЕРЕГЛЯДУ ПРОГНОЗУ ПОГОДИ

Виконав: студент групи 1П-22
Спеціальності
121 Інженерія програмного забезпечення
Владислав ВАЩЕНКО

Керівник:
Вікторія НЕМЧЕНКО

Черкаси 2026

ЧЕРКАСЬКИЙ ДЕРЖАВНИЙ ФАХОВИЙ БІЗНЕС-КОЛЕДЖ

Циклова комісія комп’ютерної інженерії та інформаційних технологій

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ

Завідувачка циклової комісії КІ та ІТ

_____ Наталя ФАЛЬЧЕНКО

(підпис)

«_____» _____2025 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Ващенку Владиславу Олександровичу

1. Тема кваліфікаційної роботи «Мобільний застосунок для перегляду прогнозу погоди»

Керівник роботи Немченко Вікторія Юріївна, викладач другої категорії затверджені наказом закладу фахової передвищої освіти від «06» жовтня 2025 року №84/1-у

2. Строк подання студентом кваліфікаційної роботи 02.06.2026

3. Вихідні дані до кваліфікаційної роботи мова програмування TypeScript, фреймворк React Native, платформа Expo SDK 54, маршрутизація Expo Router, зовнішні сервіси StormGlass API та Nominatim API, середовище розробки Visual Studio Code, система контролю версій Git.

4. Зміст кваліфікаційної роботи аналіз ринку мобільних погодних застосунків та виявлення типових недоліків наявних рішень; огляд конкурентних продуктів та формування вимог до власного застосунку; вибір технологічного стеку та обґрунтування вибору кожного компонента; проєктування архітектури застосунку з розділенням відповідальності між модулями; реалізація основних екранів: поточна погода, пошук міста, геолокація та налаштування; підключення StormGlass API для отримання погодних даних та Nominatim API для геокодування; функціональне тестування за 13 сценаріями та автоматизоване модульне тестування 29 функцій.

5. Дата видачі завдання 15.09.2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Терміни виконання етапів	Примітка про виконання
1	Вступ	14.10.2025	
2	Розділ 1. Аналіз предметної галузі та постановка задачі	09.12.2025	
3	Розділ 2. Розробка застосунку	10.03.2026	
4	Розділ 3. Тестування застосунку	28.04.2026	
5	Висновки	12.05.2026	
6	Оформлення кваліфікаційної роботи (чистовий варіант)	26.05.2026	
7	Перевірка кваліфікаційної роботи на наявність ознак плагіату (за 10 днів до захисту)	02.06.2026	
8	Подання кваліфікаційної роботи на затвердження завідувачу кафедри (за 7 днів до захисту)	10.06.2026	

Студент _____ Владислав ВАЩЕНКО

(підпис)

Керівник роботи _____ Вікторія НЕМЧЕНКО

(підпис)

АНОТАЦІЯ

Кваліфікаційна робота присвячена розробці кросплатформного мобільного застосунку для перегляду прогнозу погоди з використанням технологій React Native та Expo. Застосунок отримав назву Raindji та реалізований як повноцінний продукт для платформ Android та iOS.

У роботі здійснено аналіз предметної області метеорологічних інформаційних систем, сформульовано вимоги до застосунку та спроєктовано його UX/UI-дизайн: розроблено діаграму User Flow, вайрфрейми та мокапи трьох основних екранів.

Застосунок реалізовано з використанням Expo SDK 54, React 19, TypeScript та проміжного Node.js/Express-сервера для безпечної взаємодії зі StormGlass API. Реалізований застосунок забезпечує відображення поточної погоди та прогнозу на 7 днів, погодинного прогнозу на 24 години, визначення місцезнаходження через геолокацію, пошук міста, push-сповіщення, темну та світлу теми, локалізацію українською та англійською мовами.

Проведено функціональне тестування та тестування продуктивності. Застосунок готовий до використання на пристроях Android та iOS.

Кваліфікаційна робота містить три розділи, список використаних джерел та додатки.

Ключові слова: МОБІЛЬНИЙ ЗАСТОСУНОК, RAINDJI, REACT NATIVE, EXPO, ПОГОДА, ГЕОЛОКАЦІЯ, UX/UI, USER FLOW, STORMGLASS API, TYPESCRIPT, PUSH-СПОВІЩЕННЯ.

ABSTRACT

This qualification work is devoted to the development of a cross-platform mobile weather forecast application using React Native and Expo technologies. The application, named Raindji, is implemented as a fully functional product for Android and iOS platforms.

The work includes an analysis of the meteorological information systems domain, requirements were formulated and UX/UI design was developed: a User Flow diagram, wireframes and mockups for three main screens were created.

The application was implemented using Expo SDK 54, React 19, TypeScript and an intermediate Node.js/Express server for secure interaction with the StormGlass API. The application provides current weather display and a 7-day forecast, an hourly forecast for 24 hours, geolocation-based city detection, city search, push notifications, dark and light themes, and localization in Ukrainian and English.

Functional testing using 13 test cases and performance testing were conducted. The application is ready for use on Android and iOS devices.

Keywords: MOBILE APPLICATION, RAINDJI, REACT NATIVE, EXPO, WEATHER, GEOLOCATION, UX/UI, USER FLOW, STORMGLASS API, TYPESCRIPT, PUSH NOTIFICATIONS.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ

API	Application Programming Interface – програмний інтерфейс застосунку
UI	User Interface – інтерфейс користувача
UX	User Experience – досвід користувача
SDK	Software Development Kit – набір засобів розробки програмного забезпечення
HTTP/HTTPS	HyperText Transfer Protocol / Secure – протокол передачі гіпертексту
JSON	JavaScript Object Notation – текстовий формат обміну даними
REST	Representational State Transfer – архітектурний стиль для API
ПЗ	Програмне забезпечення
OTA	Over-The-Air – безпровідне оновлення застосунку

ВСТУП

Погода – одна з перших речей, які людина перевіряє вранці. Від неї залежать вибір одягу, маршрут до роботи чи навчання, плани на вихідні та безліч інших щоденних рішень. Тому мобільний погодний застосунок – це не просто зручна дрібниця, а повсякденний інструмент, яким користуються мільйони людей щодня. Щоб він справді виконував своє завдання, він повинен бути швидким, зрозумілим і стабільним.

Незважаючи на велику кількість погодних застосунків у Google Play та App Store, більшість із них мають спільні проблеми. Перша – перевантаженість інтерфейсу: рекламні банери, маловажливі блоки і складна навігація заважають швидко знайти потрібне. Друга – некоректна робота при зміні локації: часто потрібно виконати кілька кроків замість одного. Третя – незрозумілі повідомлення про помилки: якщо дані не завантажились, застосунок просто показує порожній екран. Ці недоліки знижують зручність продукту навіть за наявності гарного прогнозу.

Розробка погодного застосунку є вдалим вибором для кваліфікаційної роботи з інженерії програмного забезпечення. Вона охоплює широкий спектр інженерних задач: роботу з мережею, геолокацією, асинхронними запитами, управління станом, побудову навігації, обробку помилок і відображення структурованих даних у реальному часі. Реалізація всіх цих підсистем в одному застосунку демонструє інтегральну інженерну компетентність.

Тема роботи відповідає сучасним тенденціям мобільної розробки. Кросплатформні фреймворки, зокрема React Native, активно використовуються в індустрії: на них побудовані застосунки таких компаній, як Meta, Microsoft, Shopify і Discord. Опанування цього стеку є практично цінною навичкою для майбутнього інженера програмного забезпечення. Робота над Raindji дозволяє на практиці застосувати принципи компонентної архітектури, типізації, керування станом і тестування, які лежать в основі сучасної фронтенд- та мобільної розробки.

Актуальність теми підкріплюється і практичною стороною. Погодні застосунки – один з найпоширеніших типів мобільних програм, тому навички їх розробки безпосередньо застосовні в реальній роботі. Водночас простота предметної галузі (усім зрозуміло, що таке погода) дозволяє зосередитися саме на інженерних аспектах: архітектурі, обробці даних, роботі з мережею і тестуванні, не витрачаючи зусиль на пояснення складної бізнес-логіки. Це робить погодний застосунок ідеальним навчальним проєктом, на якому можна продемонструвати повний спектр інженерних компетентностей.

Метою кваліфікаційної роботи є розробка кросплатформного мобільного застосунку Raindji для перегляду прогнозу погоди з використанням React Native, Expo та TypeScript.

Для досягнення мети поставлено такі завдання:

– проаналізувати ринок мобільних погодних застосунків та виявити типові недоліки наявних рішень;

– розглянути конкурентні продукти і сформулювати на їх основі вимоги до власного застосунку;

– обрати технологічний стек та обґрунтувати вибір кожного компонента;

– спроєктувати архітектуру застосунку з чітким розділенням відповідальності між модулями;

– реалізувати основні екрани: поточна погода, пошук міста, геолокація та налаштування;

– підключити зовнішні API для отримання погодних даних та геокодування;

– провести тестування і підтвердити коректну роботу застосунку на Android та iOS.

Об'єктом дослідження є процес розробки кросплатформного мобільного застосунку для відображення метеорологічних даних.

Предметом дослідження є програмні засоби та підходи до реалізації мобільного погодного сервісу на основі React Native, Expo, TypeScript, Expo Router та зовнішніх API.

Практичне значення одержаних результатів полягає у створенні повноцінного мобільного застосунку Raindji, готового до встановлення і використання на пристроях Android та iOS. Застосунок демонструє повний цикл прикладної розробки: від формування вимог і проєктування до реалізації функцій, підключення зовнішніх сервісів та тестування. Завдяки модульній архітектурі він може бути основою для подальшого розвитку — додавання збережених міст, налаштувань одиниць вимірювання, push-сповіщень та локалізації.

Апробація кваліфікаційної роботи. Результати, наведені в межах поточної роботи, були представлені на XVIII Всеукраїнській науково-практичній конференції «Тенденції розвитку ІТ-технологій в Україні»

РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Мобільні погодні застосунки та їх ринок

Мобільний погодний застосунок – це програма, яка отримує метеорологічні дані із зовнішнього сервісу і показує їх у зрозумілому вигляді. Для більшості людей це просто температура, іконка стану погоди і прогноз на кілька днів. Але за цим простим виглядом приховується ціла система з кількох взаємопов'язаних підсистем, кожна з яких вирішує окремий клас інженерних задач.

Перша підсистема – геолокація. Застосунок повинен знати, де знаходиться користувач. Для цього використовується GPS-модуль смартфона. Але доступ до нього потребує явного дозволу від користувача, і цей дозвіл може бути відхилений. Значить, система повинна коректно реагувати на відмову – показувати повідомлення і пропонувати ручне введення міста. Ще одна деталь: координати GPS – це числа (широта і довгота). Щоб показати їх у вигляді "Lviv, Ukraine", потрібне зворотне геокодування – запит до зовнішнього сервісу, який перетворює координати на читаємий текст.

Друга підсистема – інтеграція з API. Більшість погодних застосунків не мають власних метеостанцій. Вони отримують дані від спеціалізованих провайдерів через HTTP-запити. Відповідь приходить у форматі JSON – вкладеному об'єкті з десятками показників. Завдання розробника – правильно сформувати запит, обробити відповідь і виокремити потрібні поля.

Третя підсистема – обробка і трансформація даних. Сирі значення від API потребують обробки: температуру округлюють, швидкість вітру переводять з м/с у км/год, хмарність і вологість перетворюють на текстовий стан ("Sunny", "Cloudy" тощо). Для денного прогнозу годинні значення потрібно згрупувати по датах і знайти мінімальну та максимальну температуру.

Четверта підсистема – навігація і стан. Застосунок має кілька екранів, між якими можна переходити. При цьому обрана локація має залишатися однаковою на всіх екранах. Це означає потребу в глобальному стані, доступному будь-якому компоненту без передачі через пропси.

П'ята підсистема – відображення. Погодні дані мають бути показані так, щоб найважливіша інформація була видна одразу, без прокрутки. Погодинний прогноз зручно показувати горизонтальним прокручуваним списком, денний – вертикальним. Кожен елемент списку – мінімалістична картка з іконкою, часом або датою і температурою.

Ринок мобільних погодних застосунків великий і конкурентний. Apple надає WeatherKit та вбудований застосунок для iOS. Google пропонує прогноз у пошуковій системі й окремий застосунок для Android. Поряд з ними існують сотні незалежних рішень: від простих безкоштовних застосунків до платних з розширеними функціями. Конкуренція висока, але більшість рішень або перевантажені, або заблоковані рекламою, або занадто прості.

Попит на такі застосунки залишається стабільно великим. Погода – це одна з небагатьох тем, яка стосується абсолютно кожного щодня незалежно від регіону, сезону чи роду занять. Відкривання погодного застосунку – одне з перших дій людини зранку. Тому навіть новий продукт у цій ніші може знайти свою аудиторію, якщо вирішує конкретну проблему краще за наявні альтернативи.

З погляду функціональності погодні застосунки можна класифікувати за рівнем складності:

– прості – показують температуру та основний стан погоди для одного фіксованого міста;

– середнього рівня – додають погодинний прогноз, прогноз на кілька днів, зміну локації та геолокацію;

– розширені – включають радарні карти, попередження про небезпечні явища, кілька збережених локацій, широкі налаштування.

Найбільш затребуваною є друга категорія — застосунки, що надають повноцінний набір функцій для щоденного використання без зайвого ускладнення. Такий підхід осмислений з інженерної точки зору: краще зробити базовий сценарій досконалим, ніж намагатися охопити все одразу і зробити це посередньо.

Важливим аспектом ринку є різноманіття погодних API. Провайдери відрізняються за точністю прогнозу, ціноутворенням і форматом даних. В табл. 1.1 представлений порівняльний аналіз погодних API-провайдерів.

Таблиця 1.1 – Порівняльний аналіз погодних API-провайдерів

Провайдер	Безкоштовний ліміт	Формат відповіді	Охоплення регіонів	Особливості
StormGlass	10 запитів/добу	JSON, кілька моделей	Глобальне	Агрегує NOAA, ECMWF, власну модель SG
OpenWeather Map	1000 запитів/добу	JSON	Глобальне	Відома бібліотека, велика документація
WeatherAPI	1 млн запитів/міс.	JSON / XML	Глобальне	Детальний прогноз, пошук міст вбудований
AccuWeather	50 запитів/добу	JSON	Глобальне	Висока точність, комерційний фокус
Open-Meteo	Необмежено	JSON	Глобальне	Відкритий і безкоштовний без ключа

Серед розглянутих провайдерів вдалим вибором є StormGlass API. Головна перевага – агрегація з кількох метеорологічних моделей, що підвищує точність прогнозу. Безкоштовний рівень обмежений (10 запитів на добу), що достатньо для навчального проєкту. Формат відповіді зручний для обробки: кожен годинний запис містить значення від кількох моделей, і можна вибрати найбільш достовірне.

Розуміння цільової аудиторії допомагає правильно розставити пріоритети при розробці. Серед користувачів погодних застосунків можна виділити три основні групи. Перша – звичайні користувачі, які відкривають застосунок вранці, щоб дізнатись температуру. Для них ключова швидкість і зрозумілість головного екрана. Друга група – активні користувачі: туристи, велосипедисти, любителі природи, яким важливі деталі – сила вітру, погодинний прогноз, опади. Третя група – ті, хто планує поїздки і хоче перевірити погоду в іншому місті. Їм важлива швидка і точна зміна локації.

Виходячи з цих потреб, зручний погодний застосунок повинен показувати: на головному екрані – поточну температуру і стан, детальні показники і погодинний прогноз; нижче – денний прогноз; через кілька натискань – зміну міста. Такий ієрархічний підхід задовольняє всі три групи, не перевантажуючи інтерфейс.

Окремо варто розглянути, які саме метеорологічні показники має сенс відображати. Не всі дані, що надає API, однаково корисні пересічному користувачу. Температура повітря – найважливіший показник, який цікавить майже всіх. Стан погоди (ясно, хмарно, дощ) дає швидке загальне уявлення. Вологість важлива для самопочуття: висока вологість при спеці посилює відчуття задухи. Швидкість вітру впливає на сприйняття температури – при сильному вітрі здається холодніше. Атмосферний тиск цікавить метеочутливих людей. Видимість важлива для водіїв і пілотів. Raindji відображає всі ці показники, але розставляє їх за пріоритетом: температура і стан – великим шрифтом зверху, решта – у блоці деталей нижче.

Ще один важливий аспект – формат прогнозу в часі. Погодинний прогноз потрібен тим, хто планує найближчі години: коли вийти на прогулянку, чи встигнути до дощу. Денний прогноз потрібен для планування на кілька днів уперед: коли краще запланувати поїздку чи відпочинок. застосунок повинен надавати обидва формати: погодинний – горизонтальним прокручуваним списком, денний – вертикальним. Така комбінація покриває більшість сценаріїв планування.

Аналіз предметної галузі також виявив технічні виклики, характерні саме для погодних застосунків. По-перше, дані постійно змінюються – кожне відкриття потребує свіжого запиту. По-друге, мережа може бути нестабільною, тому потрібна якісна обробка помилок. По-третє, безкоштовні API мають ліміти запитів, що вимагає економного використання. По-четверте, геолокація потребує дозволу, який користувач може відхилити. Усі ці виклики враховано в архітектурі Raindji.

Для повноти розуміння предметної галузі варто систематизувати метеорологічні показники, які відображає застосунок, із зазначенням одиниць вимірювання та їхнього практичного значення для користувача (табл. 1.2).

Таблиця 1.2 – Метеорологічні показники застосунку

Показник	Одиниця	Практичне значення
1	2	3
Температура повітря	°C	Основний показник; визначає вибір одягу
Стан погоди	текст/іконка	Швидке загальне уявлення: ясно, хмарно, дощ
Вологість	%	Впливає на самопочуття; висока при спеці посилює задуху

Продовження таблиці 1.2

1	2	3
Швидкість вітру	км/год	Впливає на відчуття температури; важлива для активностей
Напрямок вітру	градуси/стрілка	Звідки дме вітер; корисно для прогулянок і спорту
Атмосферний тиск	гПа	Індикатор зміни погоди; важливий для метеочутливих
Видимість	км	Відстань розрізнення об'єктів; важлива для водіїв
Опади	мм	Кількість дощу/снігу; основа для визначення стану

Усі ці показники StormGlass API повертає у вигляді числових значень за координатами. Завдання застосунку – отримати ці значення, перетворити їх у зручний формат (округлити, конвертувати одиниці) і відобразити так, щоб користувач одразу зрозумів поточну ситуацію. Стан погоди (іконка та текст) обчислюється на основі кількох показників – опадів, хмарності та вологості – за алгоритмом, описаним у другому розділі.

1.2 Огляд наявних рішень

Перед початком розробки проведено аналіз чотирьох погодних застосунків у Google Play. Для аналізу обрано незалежні продукти зі стабільною аудиторією та оригінальними підходами до дизайну. Це дозволяє зрозуміти реальні інженерні та продуктові компроміси, а не просто відтворити найпоширеніші рішення.

При аналізі оцінювалися такі аспекти: структура головного екрана, спосіб зміни локації, подання погодинного прогнозу, наявність реклами, складність налаштувань, підтримка темної теми, поведінка при відсутності інтернету.

Weawow – один із небагатьох безкоштовних погодних застосунків без реклами. Це є головною конкурентною перевагою. Застосунок дозволяє вибрати погодного провайдера, підтримує широкий набір налаштувань і кілька мов. Проте ширина функціоналу призводить до перевантаженості: новому користувачу важко зрозуміти, з чого починати. Головний екран показує занадто багато блоків, які більшість людей ніколи не використовують. Для Raindji це підтверджує принцип: менше – це більше. Краще зробити кілька функцій бездоганно, ніж запропонувати все одразу і зробити це посередньо.

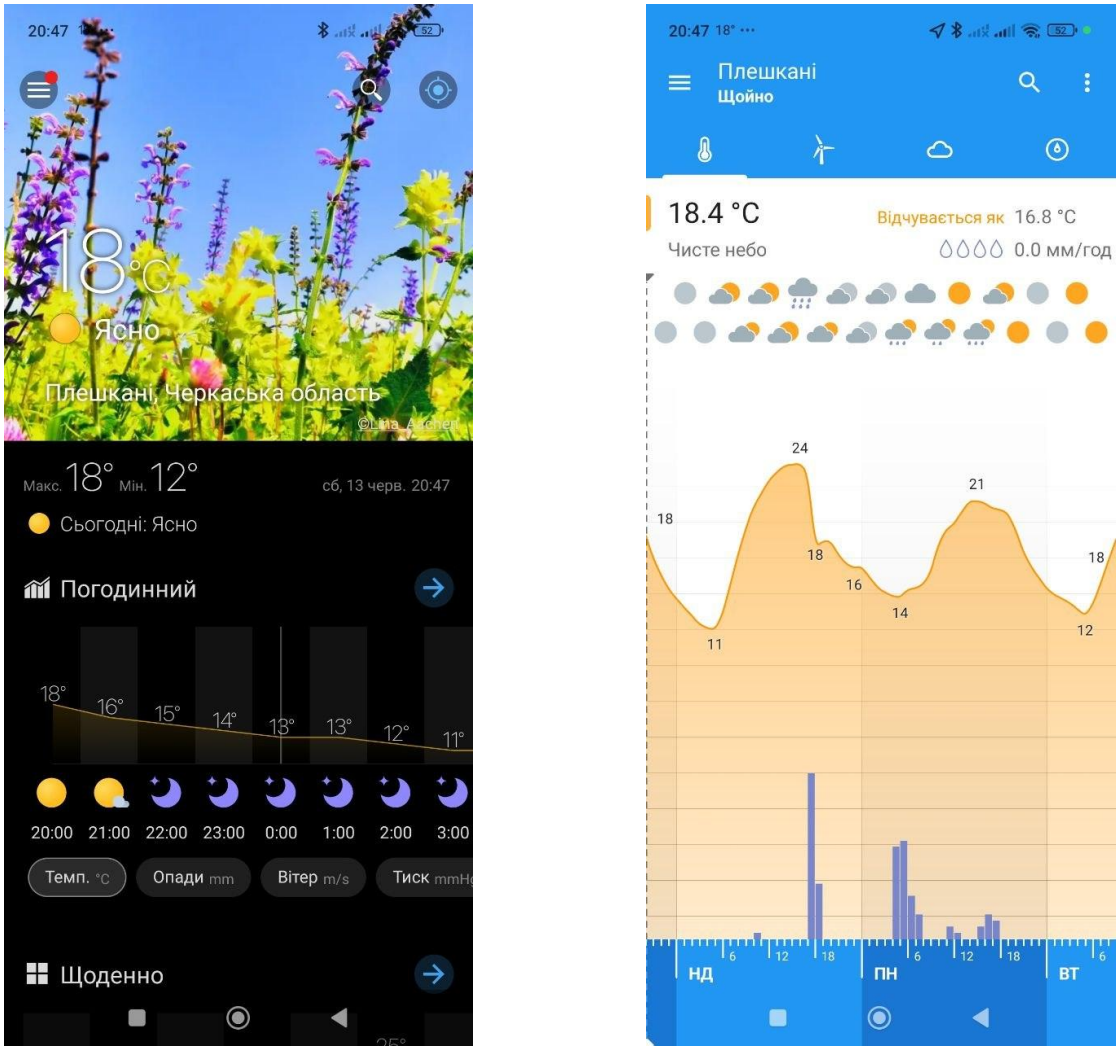
Today Weather намагається вирішити проблему точності прогнозу, надаючи вибір між кількома джерелами даних. Це розумне рішення для досвідчених користувачів, але для більшості людей такий вибір є зайвим. Результати також показали проблему: при наявності реклами та великій кількості налаштувань застосунок перестає бути зручним для щоденного використання. Висновок для розробки – одне надійне джерело даних і відсутність реклами значно покращують загальне враження.

Окремий недолік Meteogram – наявність реклами у безкоштовній версії (банер видно у верхній частині екрана) та застарілий перевантажений інтерфейс. Meteogram Weather Widget – повна протилежність простим застосункам. Він надає необмежені можливості налаштування, але платою є складність. Щоб навіть зрозуміти базові функції, нового користувача треба навчати. Для Raindji цей приклад означає: мінімальна конфігурація при першому відкритті, але при цьому наявність всієї потрібної інформації без зайвих дій.

Klara Weather також показує рекламні банери та обмежує частину можливостей у безкоштовній версії. Klara Weather демонструє нестандартний підхід до відображення: замість числових значень застосунок показує кольорові смуги і лінійні графіки. Для частини аудиторії це інтуїтивно, для іншої – навпаки.

Для Raindji висновок: гібридний підхід – числа для поточної погоди і прості картки для прогнозів – дає кращий баланс зрозумілості.

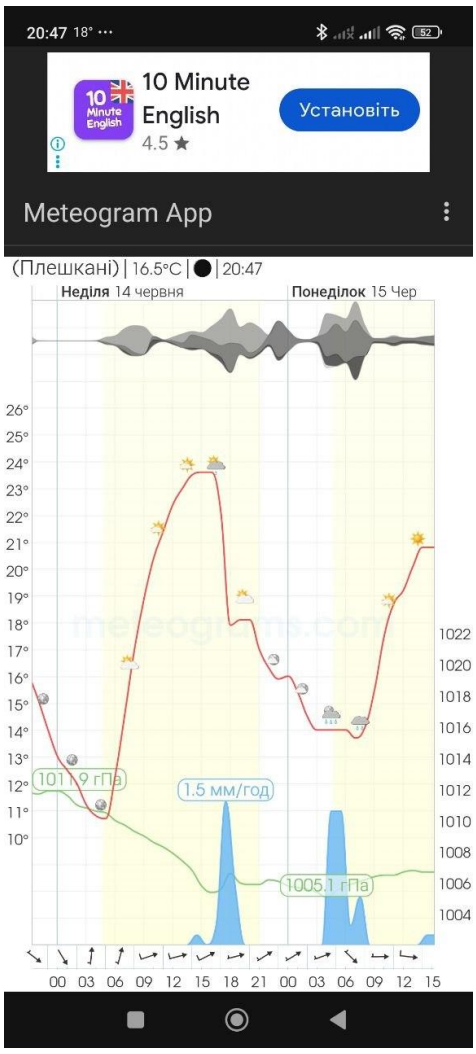
Варто також звернути увагу на спільні технічні рішення цих застосунків. Усі вони використовують зовнішні погодні API, кешують дані для зменшення кількості запитів, підтримують роботу в офлайн-режимі (показуючи останні завантажені дані) і реалізують геолокацію. Це підтверджує, що обраний для Raindji набір функцій відповідає індустріальному стандарту, а не є випадковим вибором.



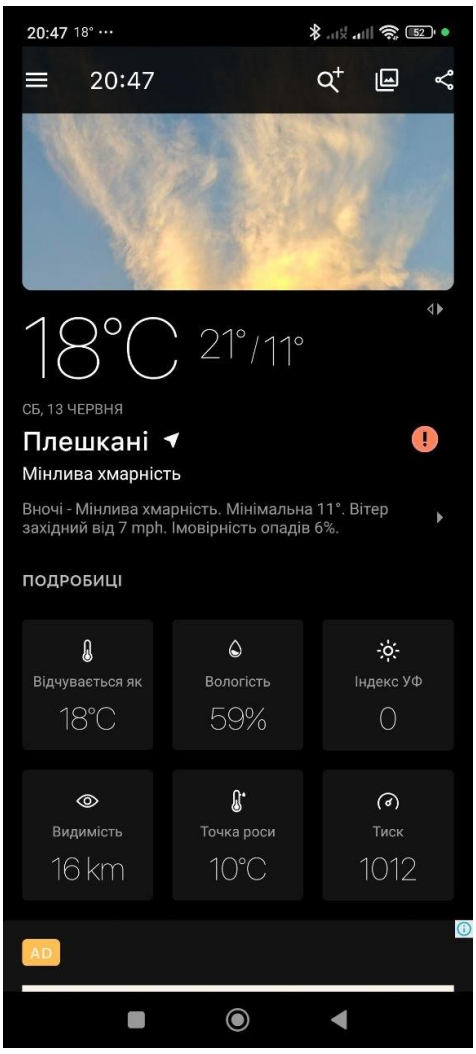
Weawow Today Weather

Рисунок 1.1 – Головні екрани розглянутих застосунків Weawow та Today Weather

Цікавим спостереженням є те, як застосунки балансують між функціональністю і простотою. Weawow і Meteogram обрали шлях максимальної гнучкості, пожертвувавши простотою. Klara обрала специфічний візуальний стиль, пожертвувавши універсальністю. Today Weather зайняв проміжну позицію, але втратив у чистоті інтерфейсу через рекламу. Raindji свідомо обирає простоту як основну цінність – це незайнята ніша серед проаналізованих рішень.



Meteogram



Klara Weather

Рисунок 1.2 – Головні екрани розглянутих застосунків Meteogram та Klara Weather

Таблиця 1.3 – Порівняльний аналіз погодних застосунків

Застосунок	Основні можливості	Сильні сторони	Обмеження	Висновок для розробки
Weawow	Прогноз, віджети, радар, кілька погодних провайдерів	Без реклами; велика аудиторія; гнучке налаштування	Інтерфейс перевантажений для простих потреб	Головний екран – простий; надлишок функцій шкодить
Today Weather	Вибір джерел для різних країн, карти, попередження	Сильне подання основних даних; вибір провайдера	Реклама; складний вибір джерела для нових користувачів	Одне джерело API, але надійне і без реклами
Meteogram	Метеограма, кілька моделей, тисячі налаштувань	Гнучкий для досвідчених; добре для аналізу тенденцій	Дуже складний для нових користувачів	Пріоритет – зрозумілість; нуль зайвих налаштувань
Klara Weather	Прогноз у вигляді інтерактивних графіків	Компактно; добре видно тенденції зміни температури	Незвичний формат не для всіх	Поєднати числові значення зі списками прогнозу

Аналіз чотирьох застосунків показав кілька важливих закономірностей. По-перше, відсутність реклами є суттєвою перевагою: користувачі сприймають

застосунок значно позитивніше, якщо він не блокує основну дію рекламними блоками. По-друге, простота навігації важливіша за ширину функціоналу: більшість людей відкривають погодний застосунок на 5–10 секунд і одразу шукають конкретну відповідь. По-третє, прозорість роботи з даними критична: якщо прогноз не завантажився – потрібне пояснення з можливістю повторити спробу.

На основі аналізу сформульовано чотири ключові принципи, які варто врахувати при розробці застосунку:

– головний екран показує найважливіше без прокрутки і без зайвих блоків;

– зміна локації – максимум два кроки від головного екрана;

– помилки мережі і API – пояснювати текстом і давати можливість повторити;

– архітектура – модульна, щоб легко додавати нові функції.

1.3 Постановка завдання та вимоги до застосунку

На основі аналізу предметної галузі і конкурентних рішень сформульовано задачу: розробити мобільний застосунок Raindji, який дозволяє переглядати прогноз погоди для поточного або вибраного міста. Застосунок повинен бути простим у використанні, стабільно працювати на Android та iOS і мати зрозумілу модульну архітектуру.

1.3.1 Функціональні вимоги

Функціональні вимоги описують, що система повинна робити. Кожна вимога сформульована як дія системи у відповідь на дію користувача або системну подію.

Таблиця 1.4 – Функціональні вимоги

ID	Вимога	Сценарій	Критерій прийнятності
FR-01	Відображення поточної погоди	Користувач відкриває застосунок	Показані температура, стан, вологість, вітер, тиск, видимість
FR-02	Погодинний прогноз	Прокрутка головного екрана	Горизонтальний список на 24 год відображається без помилок
FR-03	Денний прогноз	Перегляд тижневого прогнозу	Для кожного дня показано макс./мін. температуру та іконку
FR-04	Геолокація	Натискання GPS-кнопки	Після дозволу – прогноз для поточного місця
FR-05	Пошук міста	Введення назви міста	Список результатів; вибір оновлює прогноз
FR-06	Pull-to-refresh	Жест протягування донизу	Виконується повторний запит до API
FR-07	Обробка помилок API	Мережа недоступна або API повертає помилку	Показано текст помилки і кнопка повтору
FR-08	Відображення іконок погоди	Стан погоди визначено за алгоритмом	Іконка відповідає поточному стану

1.3.2 Нефункціональні вимоги

Нефункціональні вимоги описують, як система повинна виконувати свою роботу. Вони впливають на вибір технологій та архітектурні рішення.

Таблиця 1.5 – Нефункціональні вимоги

Атрибут якості	Вимога	Метрика / умова	Спосіб перевірки
Продуктивність	Швидке завантаження	Час відповіді ≤ 3 с при Wi-Fi	Вимірювання від відкриття до показу даних
Надійність	Стабільна робота	0 збоїв за 8 годин безперервної роботи	Тривале тестування з авто-оновленням
Адаптивність	Коректне відображення	Підтримка різних розмірів екранів	Тестування на кількох пристроях
Кросплатформність	Єдина кодова база	Спільний код > 90%	Запуск на Android та iOS
Зрозумілість коду	Модульна структура	Поділ: екрани, хуки, сервіси, контекст	Огляд архітектури
Доступність	Мінімальна зона дотику	Зони натискання ≥ 44×44 pt	Перевірка розмірів компонентів

Визначені вимоги є вихідними даними для проєктування архітектури та вибору технологій. Кожне технічне рішення в наступному розділі обґрунтовується з точки зору відповідності конкретній вимозі.

Для впорядкування роботи над вимогами їх класифіковано за пріоритетом за методом MoSCoW. До категорії Must (обов'язково) віднесено: відображення поточної погоди, погодинний і денний прогноз, геолокацію і пошук міста. До категорії Should (бажано) – обробку помилок, pull-to-refresh і підтримку темної теми. До категорії Could (можливо) – підготовку структури налаштувань. До категорії Won't (не в цій версії) – збережені міста, push-сповіщення, радарні карти і локалізацію кількома мовами. Такий розподіл дозволив зосередитися на ключовому функціоналі і не розпорошувати зусилля.

Таблиця 1.6 – Пріоритезація вимог за методом MoSCoW

Пріоритет	Вимоги
Must (обов'язково)	Поточна погода; погодинний прогноз; денний прогноз; геолокація; пошук міста
Should (бажано)	Обробка помилок API і мережі; pull-to-refresh; темна та світла теми
Could (можливо)	Підготовлена структура екрана налаштувань; екран версії
Won't (не в цій версії)	Збережені міста; push-сповіщення; радарні карти; локалізація

Висновок до розділу 1

Аналіз предметної галузі показав, що ринок погодних застосунків насичений, але більшість рішень мають типові проблеми. Огляд чотирьох конкурентних продуктів дав практичні орієнтири та підтвердив: простота і надійність важливіші за широту функціоналу. На основі аналізу сформульовано 8 функціональних і 6 нефункціональних вимог, які стали основою для подальшої розробки.

РОЗДІЛ 2 РОЗРОБКА ЗАСТОСУНКУ

2.1 Вибір технологій

Вибір технологічного стеку – одне з перших і найважливіших рішень при розробці мобільного застосунку. Воно визначає, наскільки швидко можна розпочати розробку, які платформи підтримуватимуться, яка буде продуктивність і наскільки легко підтримувати код у майбутньому. При цьому неправильний вибір важко виправити на пізніх стадіях – зміна фреймворку потребує повного переписування.

Для кросплатформної мобільної розробки існує кілька основних підходів. Нативна розробка на Swift та Kotlin дає максимальну продуктивність і повний доступ до API пристрою, але потребує двох окремих кодових баз і суттєво більших витрат часу. Flutter використовує мову Dart і власний рушій рендерингу – це забезпечує стабільно однаковий вигляд на обох платформах, але Dart менш поширений у вебспільноті. React Native дозволяє писати на JavaScript або TypeScript і ділитися до 95% коду між Android та iOS, використовуючи реальні нативні компоненти ОС.

Для Raindji обрано React Native з Expo SDK 54. Це рішення обґрунтовується кількома конкретними причинами, прив'язаними до функціональних вимог проєкту.

По-перше, геолокація є ключовою функцією. Expo надає готовий модуль expo-location з простим API: кілька рядків коду дозволяють запитати дозвіл і отримати координати. В bare React Native або нативній розробці це потребує окремого налаштування для кожної платформи.

По-друге, Expo Router дозволяє організувати навігацію через файлову структуру папки app. Шлях до файлу відповідає маршруту застосунку, що робить структуру екранів зрозумілою без читання коду навігації.

По-третє, Expo зменшує технічну рутину при старті проєкту. Немає потреби налаштовувати Gradle, Xcode, сертифікати або нативні залежності – Expo бере це на себе. По-четверте, Expo Snack і web-режим дозволяють запустити застосунок у браузері без емулятора. Це суттєво прискорює демонстрацію під час захисту та перевірку нових функцій. По-п'яте, TypeScript з strict-режимом забезпечує контроль типів при роботі з вкладеними даними API.

Таблиця 2.1 – Порівняльний аналіз платформ мобільної розробки

Критерій	React Native + Expo	Flutter	Native Swift/Kotlin	Xamarin/MAUI
Швидкість розробки	Дуже висока	Висока	Низька	Середня
Доступ до API пристрою	Через Expo SDK	Через плагіни	Прямий, повний	Xamarin.Essentials
Продуктивність	Висока (нативні компоненти)	Максимальна	Абсолютний максимум	Середня
Спільний код	До 95%	До 90%	0%	До 80%
Складність входження	Низька (JS/TS)	Середня (Dart)	Висока (дві мови)	Середня (C#)
Розмір спільноти	Дуже велика	Зростаюча	Велика	Невелика

Окремо варто пояснити, чому кросплатформність така важлива для цього проєкту. Якби застосунок розроблявся нативно, довелося б писати дві окремі версії: одну на Swift для iOS, іншу на Kotlin для Android. Це подвоїло б обсяг коду, час розробки і складність підтримки. Будь-яка нова функція

реалізовувалася б двічі, будь-яка помилка виправлялася б у двох місцях. React Native усуває цю проблему: одна кодова база працює на обох платформах. Експерименти показують, що в Raindji понад 95% коду спільне для Android та iOS – платформозалежними залишаються лише дрібні деталі, як-от системні відступи.

React Native досягає цього завдяки своїй архітектурі. JavaScript-код виконується в окремому потоці і спілкується з нативними компонентами через міст (bridge) або, у нових версіях, через JSI. Важливо, що React Native використовує реальні нативні елементи інтерфейсу, а не їх імітацію. Кнопка в React Native – це справжня нативна кнопка iOS чи Android, тому застосунок виглядає і поводиться природно на кожній платформі.

2.1.1 Технологічний стек і ключові рішення

Таблиця 2.2 – Технологічний стек застосунку

Технологія / Інструмент	Версія	Призначення
React Native	0.79	Фреймворк для кросплатформної мобільної розробки
Expo SDK	54	Набір нативних модулів: геолокація, теми, файлова система
Expo Router	4.x	Файлова маршрутизація та навігація між екранами
TypeScript	5.x	Статична типізація; контроль структури даних API
StormGlass API	v2	Отримання погодних даних за координатами
Nominatim API	v1	Геокодування: пошук міста та зворотне геокодування
React Context	18.x	Глобальний стан вибраної локації
expo-location	17.x	GPS-доступ і дозволи геолокації

Мовою програмування обрано TypeScript. Він дає явне описання типів даних, що особливо важливо при роботі з API, де відповідь може бути вкладеним об'єктом. Наприклад, у сервісі погоди визначено інтерфейси: StormglassHour описує один годинний запис з кількома метеорологічними параметрами кожен від кількох моделей. WeatherData описує фінальний об'єкт, який бачать екрани. CurrentWeather, HourlyForecast і DailyForecast – три вкладені структури. Без TypeScript помилки в обробці таких даних виявляються тільки в runtime.

Перевага TypeScript добре проявляється при рефакторингу. Якщо змінити структуру даних – наприклад, додати нове поле до WeatherData – компілятор одразу вкаже всі місця, які потрібно оновити. У звичайному JavaScript такі помилки виявилися б лише під час виконання, можливо, вже у користувача. Для навчального проєкту це також корисно з освітньої точки зору: типи виступають документацією, яка пояснює структуру даних без додаткових коментарів.

Вибір StormGlass API замість альтернатив теж обґрунтований. OpenWeatherMap має більший безкоштовний ліміт, але повертає дані лише від однієї моделі. StormGlass агрегує кілька моделей (SG, NOAA, ECMWF), що дозволяє реалізувати запасну логіку: якщо одна модель не дала значення, береться інша. Це підвищує надійність застосунку при неповних даних. Open-Meteo взагалі безкоштовний без ключа, але StormGlass обрано як приклад роботи з автентифікацією через API-ключ – це ближче до реальних комерційних сценаріїв.

Nominatim обрано для геокодування, оскільки він безкоштовний, не потребує ключа і базується на відкритих даних OpenStreetMap. Єдина вимога – наявність заголовка User-Agent у запитах. Альтернативою був би Google Geocoding API, але він потребує платного ключа і налаштування білінгу, що надмірно для навчального проєкту.

2.2 Проєктування архітектури

Архітектура застосунку побудована за принципом розділення відповідальності. Це означає, що кожен модуль системи має конкретну і єдину задачу. Зміна одного модуля не вимагає редагування інших. Це дозволяє підтримувати і розвивати систему без ризику несподіваних помилок у несуміжних частинах.

Загальна структура проєкту відповідає рекомендаціям Expo Router. Файли у папці app автоматично стають маршрутами. Папка hooks містить кастомні хуки, services – сервісний шар, constants – константи і провайдери контексту.

Принцип розділення відповідальності реалізовано послідовно на всіх рівнях. Екрани відповідають тільки за відображення і взаємодію з користувачем – вони не містять логіки запитів до API. Хуки інкапсулюють асинхронну логіку – завантаження даних, обробку станів, скасування запитів. Сервіси ізолюють зовнішні залежності – усі знання про конкретні API зосереджені в одному файлі. Контекст зберігає глобальний стан. Така багаторівнева структура полегшує і тестування: сервісні функції можна перевірити окремо від інтерфейсу, що й зроблено в розділі тестування.

Важлива перевага такої архітектури – можливість заміни компонентів. Якщо в майбутньому потрібно змінити погодного провайдера зі StormGlass на інший, зміни торкнуться лише файлу weather.ts. Екрани, хуки і контекст залишаться без змін, бо вони працюють зі стабільним внутрішнім форматом WeatherData, а не з форматом конкретного API. Це класичний приклад патерну "адаптер" у дії: сервіс адаптує зовнішній формат до внутрішнього.

Загальну будову застосунку зручно подати у вигляді багаторівневої схеми, де кожен рівень спирається на нижчий і не залежить від внутрішніх деталей сусідніх рівнів. Така схема наведена на рис. 2.1.

Як видно зі схеми, рівень екранів звертається до рівня хуків, хуки – до сервісного рівня, а сервіси – до зовнішніх API. Жоден верхній рівень не

звертається до зовнішніх сервісів напряму, а жоден нижній рівень не знає про існування екранів. Така односпрямованість залежностей робить систему передбачуваною і легкою для тестування.

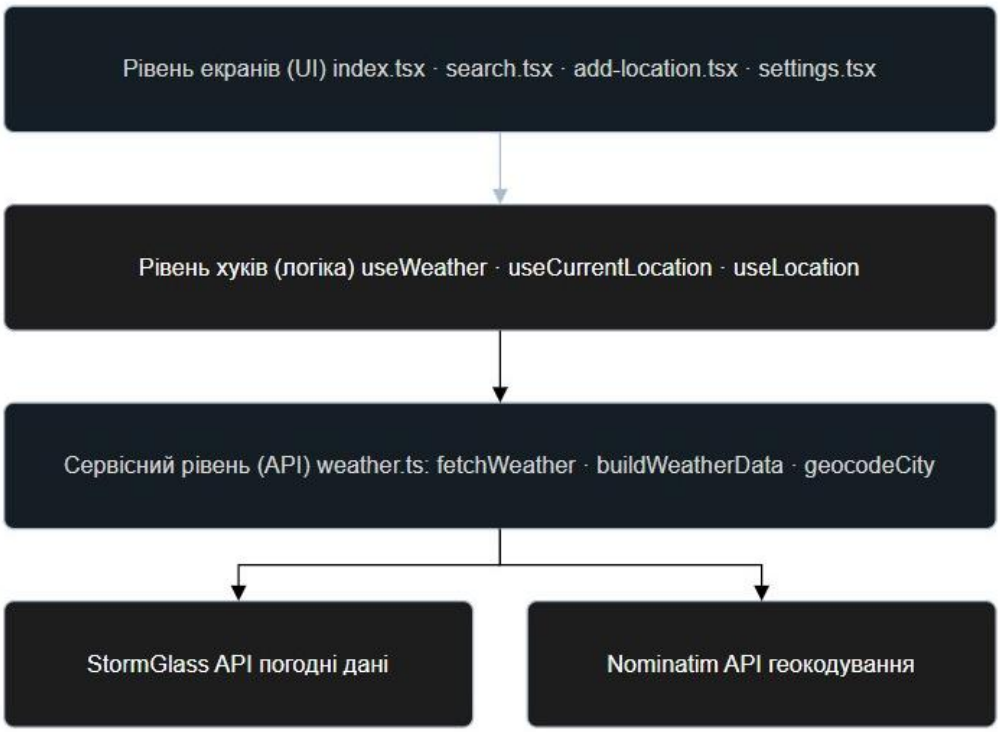


Рисунок 2.1 – Багаторівнева архітектура застосунку Raindji

Експо-роутер задає ще одну важливу архітектурну особливість – групування екранів. Папка (tabs) у круглих дужках означає групу маршрутів, що ділять спільний макет. Дужки в назві папки сигналізують роутеру, що сама папка не додає сегмент до шляху, а лише групує екрани. Завдяки цьому всі вкладки мають спільну нижню панель навігації, але кожна залишається окремим файлом. Це елегантне рішення, яке поєднує організованість коду зі спільним макетом.

Файл _layout.tsx відіграє роль обгортки для всіх дочірніх екранів. У ньому розташовані провайдери – компоненти, які надають дані всім вкладеним екранам. LocationProvider робить поточну локацію доступною скрізь, ThemeProvider – поточну тему. Провайдери працюють за принципом React Context: вони огортають дерево компонентів, і будь-який компонент усередині може отримати доступ до наданих даних без передачі їх через проміжні рівні. Це

усуває проблему "props drilling" – передачі даних через багато проміжних компонентів, які самі ці дані не використовують.

Таблиця 2.3 – Модульна структура застосунку

Модуль	Файл	Відповідальність
Кореневий layout	app/_layout.tsx	Провайдери LocationContext і ThemeProvider, Stack-навігація, StatusBar
Головний екран	app/(tabs)/index.tsx	Показ поточної погоди, погодинного та денного прогнозу
Пошук та GPS	app/(tabs)/search.tsx	Відображення поточної локації, кнопка запуску GPS
Додавання міста	app/(tabs)/add-location.tsx	Текстовий пошук через Nominatim, FlatList результатів
Налаштування	app/(tabs)/settings.tsx	Підготовлена структура для майбутніх налаштувань
Версія	app/(tabs)/version.tsx	Відображення поточної версії застосунку
Погодні сервіси	services/weather.ts	HTTP-запити до StormGlass і Nominatim, типи даних, buildWeatherData
Хук прогнозу	hooks/use-weather.ts	Стани data / loading / error, функція refresh, cancelledRef
Хук геолокації	hooks/use-current-location.ts	Запит дозволу, координати, reverseGeocode
Контекст локації	constants/location-context.tsx	Глобальний стан: lat, lng, city, country і setLocation
Конфігурація	constants/config.ts	Базові URL і API-ключ StormGlass

2.2.1 Потік даних у застосунку

Потік даних організовано послідовно і без зворотних залежностей. Головний екран отримує локацію з LocationContext. Хук useWeather приймає координати і передає їх у fetchWeather. Функція fetchWeather формує HTTP-запит до StormGlass, отримує відповідь і повертає масив годинних записів. Функція buildWeatherData перетворює ці записи на структуру WeatherData. Хук повертає об'єкт {data, loading, error, refresh} на головний екран. Екран рендерить відповідний стан.

Якщо користувач змінює місто через пошук або GPS, setLocation оновлює LocationContext. Усі компоненти, що підписані на цей контекст, автоматично отримують нові координати. useWeather виявляє зміну залежностей і запускає новий запит. Цикл повторюється. Узагальнено цей потік даних показано на рис. 2.2.

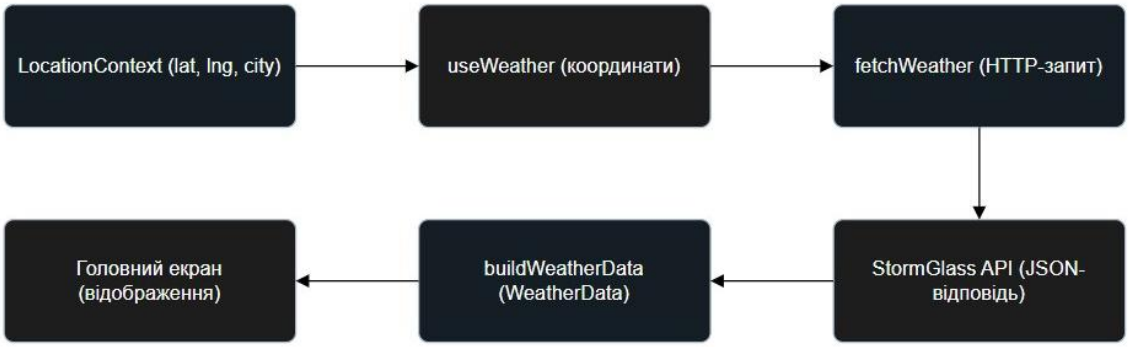


Рисунок 2.2 – Потік даних при завантаженні прогнозу

Глобальний стан реалізовано через React Context замість Redux або Zustand. Це обґрунтовано масштабом проєкту: для одного об'єкта стану (поточна локація) складні менеджери стану є надлишковими. React Context добре вписується в архітектуру Expo і не потребує додаткових залежностей.

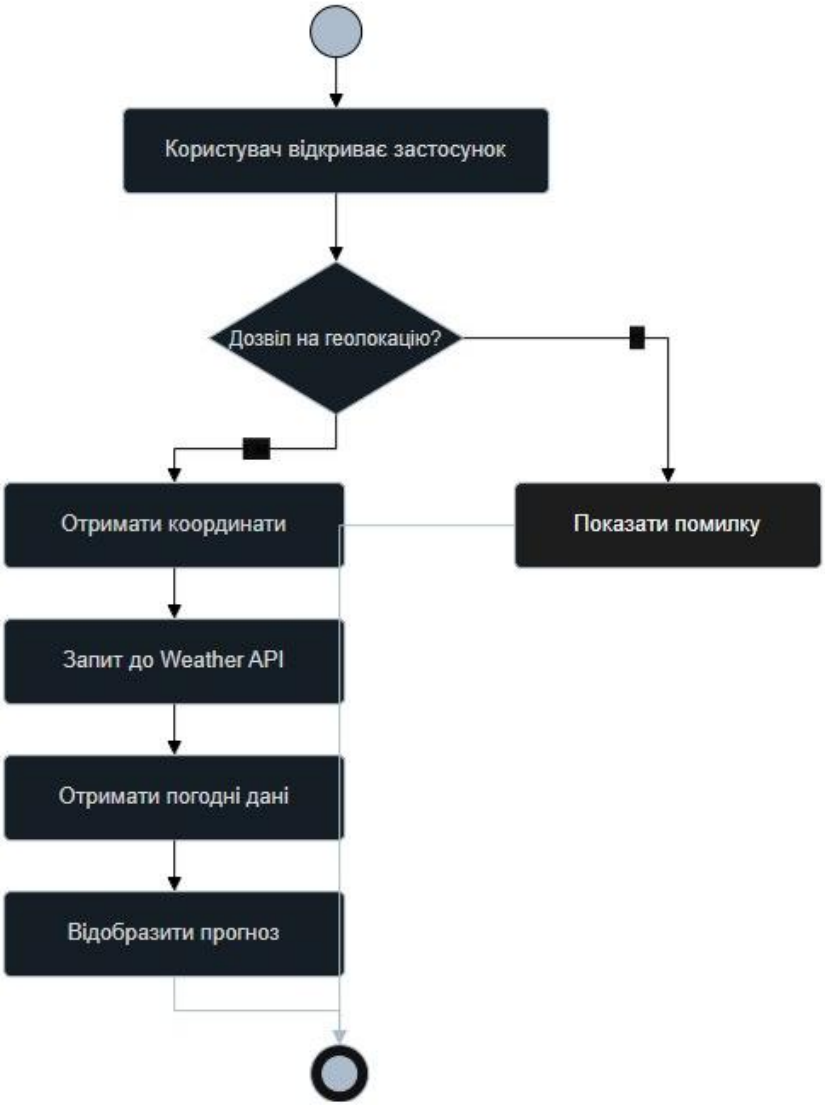


Рисунок 2.3 – Діаграма активності отримання погодних даних

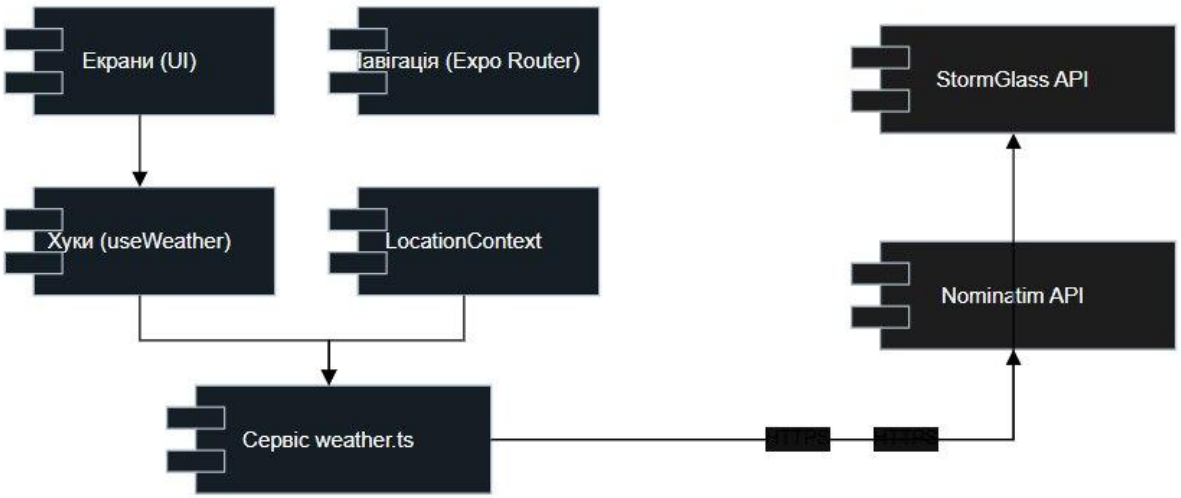


Рисунок 2.4 – Компонентна схема мобільного застосунку та зовнішніх сервісів

Хуки застосунку

Хук useWeather є проміжною ланкою між екраном і сервісом. Усередині хука оголошено три стани: data (WeatherData або null), loading (boolean) та error (string або null). Функція loadData, мемоїзована через useCallback, викликає fetchWeather і buildWeatherData. Важлива деталь реалізації – cancelledRef. Це мутабельне посилання-прапорець, який встановлюється в true при розмонтуванні компонента. Перед кожним оновленням стану перевіряється cancelledRef.current – якщо він true, оновлення пропускається. Без цього в консолі з'являлося б попередження про оновлення розмонтованого компонента.

Хук useCurrentLocation відповідає за геолокацію. Спочатку він запитує дозвіл foreground через Location.requestForegroundPermissionsAsync. Foreground – це доступ до місця лише під час активного використання застосунку, без фонового відстеження. Якщо дозвіл відхилено, хук записує помилку і повертає null. Якщо дозвіл надано, координати отримуються через Location.getCurrentPositionAsync з точністю Balanced. Потім виконується reverseGeocode для визначення назви міста.

Вибір точності Balanced замість Highest є свідомим компромісом. Найвища точність GPS споживає більше енергії і потребує більше часу на визначення координат. Для погодного застосунку висока точність не потрібна: погода в межах міста однакова, тому достатньо знати місто з точністю до кількох сотень метрів. Balanced забезпечує швидке визначення локації при помірній витраті батареї, що краще відповідає сценарію використання.

Робота з дозволами реалізована з урахуванням різних станів. Дозвіл може бути наданий, відхилений або відхилений назавжди (коли користувач обрав "більше не питати"). У першому випадку застосунок отримує координати. У другому – показує повідомлення і пропонує ручний пошук. У третьому – пояснює, що дозвіл потрібно увімкнути в налаштуваннях системи. Така деталізація робить застосунок дружнім навіть у нестандартних ситуаціях.

2.3 Розробка дизайну та навігаційний потік

Дизайн Raindji підпорядкований одній цілі: користувач повинен одразу бачити найважливіші дані без зайвих кроків. Цей принцип визначає всі рішення в інтерфейсі: ієрархію елементів, кольори, розміри шрифтів, розташування кнопок.

Застосунок підтримує темну і світлу теми. Темна тема використовує фон #0A0F1E – це не чорний, а дуже темно-синій, який менш втомлює очі. Акцентний колір #4FC3F7 – блакитний, що асоціюється з небом і погодою. Світла тема – #FFFFFF фон і #039BE5 акцент. Перемикання відбувається автоматично через useColorScheme, що зчитує системне налаштування пристрою. Обидві теми перевірено на відповідність рекомендаціям WCAG 2.1 щодо контрастності.

Підтримка двох тем – це не просто естетична деталь, а питання зручності й доступності. Темна тема комфортніша при використанні застосунку ввечері або в темному приміщенні, споживає менше енергії на OLED-екранах і зменшує навантаження на очі. Світла тема краще читається при яскравому денному світлі. Автоматичне перемикання за системним налаштуванням означає, що користувачу не потрібно нічого налаштовувати вручну – застосунок адаптується сам.

Типографіка інтерфейсу побудована на чіткій ієрархії розмірів. Поточна температура відображається найбільшим шрифтом, бо це головний показник. Назва міста і стан погоди – середнім. Деталі (вологість, тиск) – дрібнішим. Така ієрархія допомагає оку швидко знайти найважливішу інформацію, не читаючи весь екран. Це відповідає принципу візуальної ієрархії в дизайні інтерфейсів.

Іконки погоди підібрано так, щоб вони були зрозумілими без підписів. Сонце означає ясну погоду, хмара – хмарну, краплі – дощ. Універсальність іконок важлива для застосунку, який може використовуватися людьми різних мов і культур. Іконка передає стан швидше, ніж текст, і не потребує перекладу.

Навігаційна структура побудована на Expo Router. Застосунок має файлову маршрутизацію: папка app/(tabs) містить файли, що відповідають вкладкам.

Кожен файл – окремий екран. Коренева конфігурація app/_layout.tsx підключає LocationProvider, ThemeProvider, визначає Stack-навігацію і налаштовує StatusBar.

Компонент SafeAreaView використовується на всіх екранах для правильних відступів від системних панелей. На сучасних смартфонах – iPhone з Dynamic Island, Android з вирізами – без SafeAreaView вміст може накладатися на системні елементи. Завдяки цьому компоненту застосунок виглядає коректно на будь-якому пристрої без ручного налаштування відступів.

Окремо варто пояснити переваги файлової маршрутизації Expo Router порівняно з традиційним підходом. У класичному React Navigation навігаційний стек описується програмно: розробник вручну реєструє кожен екран, задає його ім'я і параметри в окремому файлі конфігурації. Це створює дублювання: екран існує і як файл, і як запис у конфігурації. Файлова маршрутизація усуває це дублювання – сам факт існування файлу в папці app означає наявність маршруту. Менше коду, менше місць для помилок, простіша структура.

Переходи між екранами здійснюються через об'єкт router з методами push (відкрити новий екран поверх поточного), replace (замінити поточний екран) і back (повернутися назад). У Raindji переважно використовується push для відкриття екранів пошуку і налаштувань та автоматичне повернення після вибору локації. Така навігація формує природний стек: користувач завжди може повернутися назад системним жестом або кнопкою.

З головного екрана через кнопки у заголовку можна перейти до пошуку або налаштувань. З екрана пошуку можна відкрити екран додавання міста. Після вибору міста або визначення GPS-координат застосунок автоматично повертається на головний екран, де одразу оновлюється прогноз.

Кольорову палітру застосунку підібрано так, щоб вона асоціювалася з небом і погодою. Основні кольори інтерфейсу наведено на рисунку 2.5.

Рисунок 2.5 – Палітра кольорів інтерфейсу застосунку

Загальний вигляд готового інтерфейсу застосунку – головний екран, екран пошуку та екран налаштувань – показано на рис. 2.6.

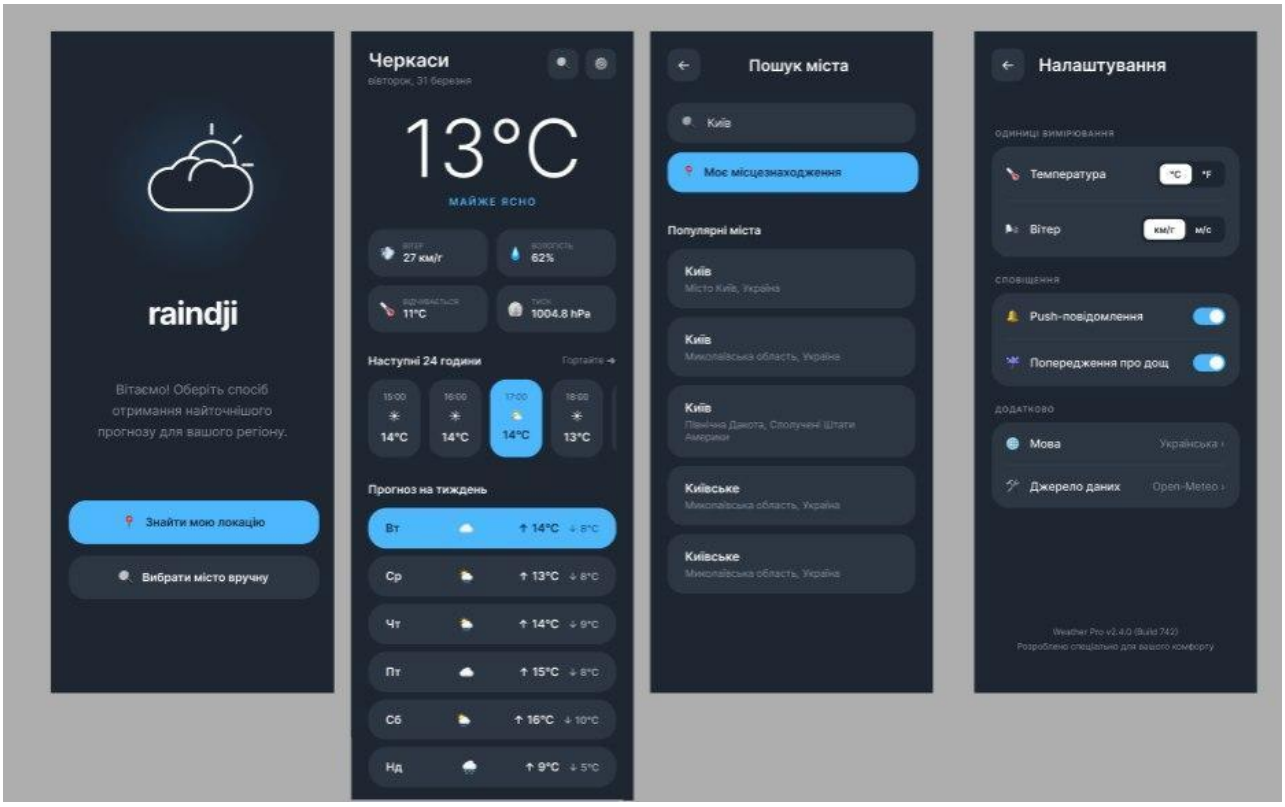


Рисунок 2.6 – Екрани застосунку Raindji

Таблиця 2.4 – Перелік екранів та їх призначення

Екран	Файл	Зміст та дії
Головний	(tabs)/index.tsx	Місто, дата, стан погоди, температура, вологість, вітер, тиск, видимість; погодинний і денний прогнози; pull-to-refresh
Пошук	(tabs)/search.tsx	Поточна локація у вигляді тексту; кнопка "Визначити через GPS"
Додавання міста	(tabs)/add-location.tsx	Текстове поле пошуку; FlatList результатів Nominatim; кнопка очищення
Налаштування	(tabs)/settings.tsx	Підготовлені пункти для мови, одиниць і теми
Версія	(tabs)/version.tsx	Поточна версія застосунку і технологічний стек

2.4 Реалізація основних екранів

2.4.1 Головний екран

Головний екран реалізовано у файлі app/(tabs)/index.tsx. Це центральна точка взаємодії користувача із застосунком. Екран отримує поточну локацію через хук useLocation і передає координати у useWeather. Залежно від стану відображається один із трьох варіантів: індикатор завантаження (ActivityIndicator), повідомлення про помилку з кнопкою повторної спроби або повний прогноз.

Після успішного завантаження екран показує кілька блоків. Верхній блок містить назву міста, поточну дату і стан погоди. Нижче – велика температура і короткий текст стану. Третій блок – деталі: вологість, швидкість і напрямок вітру, атмосферний тиск і видимість. Четвертий блок – горизонтальний

прокручуваний список погодинного прогнозу на 24 години. П'ятий блок – вертикальний список денного прогнозу.

Для напрямку вітру реалізовано функцію WindDirectionIcon. Вона приймає числовий кут у градусах і повертає Unicode-символ стрілки: 0° → "↑", 45° → "↗", 90° → "→" і так далі. Такий підхід читабельніший за числовий кут для більшості користувачів.

Весь вміст екрана обгорнутий у ScrollView з RefreshControl. RefreshControl отримує функцію refresh з useWeather і прапорець isRefreshing. Коли користувач тягне екран донизу – запускається повторний запит, і після його завершення дані оновлюються. Функція refresh мемоїзована через useCallback, тому не перестворюється при кожному рендерингу.

Жест pull-to-refresh став стандартним патерном у мобільних застосунках, тому користувачі інтуїтивно розуміють його без пояснень. Реалізація цього жесту через вбудований RefreshControl означає, що поведінка повністю відповідає очікуванням платформи: на iOS це характерний еластичний відскок, на Android – кругова стрілка зверху. Знову ж таки, використання нативних компонентів дає природну поведінку без додаткових зусиль.

Погодинний прогноз реалізовано через горизонтальний ScrollView, а денний – через звичайний вертикальний список. Вибір горизонтальної прокрутки для годин не випадковий: 24 години не помістилися б вертикально без надмірної прокрутки, а горизонтальна стрічка дозволяє швидко проглянути найближчі години одним рухом пальця. Денний прогноз, навпаки, компактний (кілька днів), тому вертикальний список природніший.

2.4.2 Екран пошуку міста

Екран add-location.tsx реалізує ручний пошук міста через Nominatim API. Екран підтримує три локальні стани: searchText (рядок поточного запиту), results (масив знайдених міст) та status (короткий текст, наприклад "Searching..." або "No location found").

При зміні тексту в полі пошуку викликається handleSearch. Якщо довжина тексту менша ніж 2 символи – список очищується і запит не виконується. Це знижує кількість зайвих HTTP-запитів. При достатній довжині тексту формується запит до geocodeCity. Результати відображаються у FlatList – компоненті React Native, оптимізованому для довгих списків з підтримкою virtualization.

Після вибору результату функція selectLocation формує об'єкт локації. Поле display_name Nominatim – це повна адреса у вигляді рядка: "Lviv, Lviv Oblast, Ukraine". Функція розбиває рядок за комами, бере перший елемент як назву міста і останній як країну. Координати lat і lon перетворюються на числа через parseFloat. Готовий об'єкт записується у LocationContext через setLocation.

Окрему увагу приділено зменшенню кількості запитів до Nominatim. Сервіс має обмеження на частоту звернень (не більше одного запиту на секунду), тому надсилати запит на кожне натискання клавіші було б неприпустимо. Мінімальна довжина запиту в два символи частково вирішує цю проблему. У майбутньому її можна посилити через debounce – затримку, яка надсилає запит лише після того, як користувач припинив друкувати на певний час. Це типова оптимізація для полів пошуку з автодоповненням.

FlatList для відображення результатів обрано не випадково. На відміну від звичайного перебору елементів через map, FlatList рендерить лише ті елементи, які видно на екрані, а решту створює при прокрутці. Для короткого списку результатів пошуку це не критично, але використання правильного компонента формує добру практику і готує застосунок до роботи з довшими списками в майбутньому.

2.4.3 Екран геолокації

Екран search.tsx надає можливість визначити поточне місцезнаходження через GPS. Після натискання кнопки викликається getLocation з хука useCurrentLocation. Відбувається три послідовних кроки: запит дозволу → отримання координат → зворотне геокодування.

При відмові у дозволі хук записує помилку "Location permission denied" і повертає null. Екран показує відповідне повідомлення. При успіху координати записуються у LocationContext і застосунок повертається на головний екран, де одразу починається завантаження прогнозу для нових координат. Під час визначення геолокації показується індикатор завантаження, щоб користувач розумів, що процес виконується.

2.4.4 Конфігурація проєкту

Файл app.json містить ключові параметри Expo-проєкту. Назва застосунку – weather-app. Версія – 1.0.0. Орієнтація – portrait (лише портретна). Параметр userInterfaceStyle встановлено як automatic, що дозволяє системі автоматично перемикати тему. Для Android увімкнено adaptiveIcon (адаптивна іконка), edgeToEdgeEnabled (макет від краю до краю) і predictiveBackGestureEnabled (жест передбачуваного повернення).

Файл tsconfig.json розширює expo/tsconfig.base і вмикає strict-режим. Також налаштовано path alias @/*, що дозволяє писати @/hooks/use-weather замість відносних шляхів. Це спрощує навігацію в коді і зменшує кількість помилок при переміщенні файлів.

2.4.5 Компоненти інтерфейсу

Інтерфейс застосунку побудований з повторюваних компонентів. Це зменшує дублювання коду і забезпечує єдиний стиль. Основні компоненти описано в табл. 2.5.

Компонент WeatherCard є центральним. Він приймає об'єкт current з WeatherData і відображає його у вигляді картки. Усередині нього розташовані менші компоненти DetailRow для кожного показника. Такий підхід – композиція з дрібних компонентів – є типовим для React і робить код передбачуваним.

Кожен компонент має одну відповідальність, легко тестується і повторно використовується.

Стилізація компонентів реалізована через StyleSheet.create – стандартний механізм React Native. Кольори не задаються напряму в компонентах, а беруться з теми через хук useTheme. Це дозволяє одним перемиканням теми змінити вигляд усього застосунку без редагування кожного компонента.

Таблиця 2.5 – Основні компоненти інтерфейсу

Компонент	Призначення	Ключові пропси
WeatherCard	Картка поточної погоди з основними показниками	temperature, condition, city
HourlyItem	Елемент погодинного прогнозу	time, temperature, icon
DailyItem	Рядок денного прогнозу	day, min, max, icon
WindDirectionIcon	Стрілка напрямку вітру	degrees
DetailRow	Рядок деталі (вологість, тиск тощо)	label, value, unit
SearchResultItem	Елемент списку результатів пошуку	name, onSelect
ErrorView	Екран помилки з кнопкою повтору	message, onRetry

2.5 Робота з погодними даними та API

Сервісний шар у файлі services/weather.ts – це ядро логіки застосунку. Тут описано TypeScript-інтерфейси для всіх структур даних, реалізовано HTTP-запити до StormGlass і Nominatim та логіку перетворення сирих даних у зручний формат для екранів.

2.5.1 TypeScript-інтерфейси

Відповідь StormGlass API містить масив об'єктів hours. Кожен об'єкт відповідає одній годині і включає значення кількох параметрів від різних метеорологічних моделей. Наприклад, поле airTemperature може містити значення від моделей sg (StormGlass), noaa (NOAA) та ecmwf (ECMWF). Якщо одна модель не надала значення, її поле відсутнє або дорівнює null.

Для опису цієї структури використано інтерфейс StormglassWeatherValue з полями sg, noaa та ecmwf – усі необов'язкові (позначені знаком ?). Інтерфейс StormglassHour описує один годинний запис: час (time) і набір метеорологічних параметрів, кожен з яких є StormglassWeatherValue. Інтерфейс StormglassResponse містить поле hours типу масив StormglassHour.

Внутрішні типи WeatherData відображають обробку: CurrentWeather містить числові значення (temperature: number, humidity: number тощо) і текстовий стан (condition: string). HourlyForecast і DailyForecast – прості картки для списків.

Розмежування між зовнішніми типами (Stormglass...) і внутрішніми (WeatherData, CurrentWeather) є важливим архітектурним прийомом. Зовнішні типи точно відображають формат API – з усіма його особливостями, як-от вкладені об'єкти з полями sg, noaa, ecmwf. Внутрішні типи – це чистий, зручний формат, з яким працюють екрани. Між ними стоїть функція buildWeatherData, яка перетворює одне в інше. Завдяки цьому екрани повністю ізольовані від формату API: якщо провайдер змінить структуру відповіді, достатньо оновити лише перетворення, а екрани залишаться незмінними.

Усі необов'язкові поля в зовнішніх типах позначені знаком питання (наприклад, sg?: number). Це змушує TypeScript вимагати перевірки на наявність значення перед використанням. Без такої перевірки компілятор видасть помилку. Це захищає від поширеної помилки – звернення до поля, якого може не бути у відповіді, що в JavaScript призвело б до undefined і потенційного збою.

2.5.2 Функція fetchWeather

Функція fetchWeather приймає широту і довготу та повертає Promise з масивом годинних записів. URL запиту формується динамічно: до базового URL StormGlass додаються параметри lat, lng і перелік погодних параметрів. Список параметрів: airTemperature, cloudCover, humidity, precipitation, windSpeed, windDirection, pressure і visibility. Ці значення відповідають тим показникам, що реально відображаються на головному екрані.

Запит виконується через стандартний fetch з заголовком Authorization: 'Apikey ' + apiKey. Після отримання відповіді перевіряється res.ok – булева властивість, яка дорівнює true при статусах 200–299. Якщо статус інший (наприклад, 403 при недійсному ключі або 429 при перевищенні ліміту), функція кидає виняток з кодом і текстом помилки. Хук useWeather перехоплює цей виняток і встановлює відповідне повідомлення в стан error.

Окремої уваги заслуговує робота з асинхронністю. Мережеві запити в JavaScript виконуються асинхронно: код не зупиняється і чекає відповіді, а продовжує роботу, а коли відповідь приходить – виконується відповідний обробник. У Raindji використано сучасний синтаксис async/await, який робить асинхронний код схожим на звичайний послідовний. Функція fetchWeather позначена як async, а виклик fetch супроводжується ключовим словом await. Це робить код читабельним і легким для розуміння, на відміну від старого підходу з ланцюжками .then().

Помилки в асинхронному коді обробляються через конструкцію try/catch. Усередині try виконується запит, а блок catch перехоплює будь-який виняток – і мережеву помилку, і виняток, кинутий вручну при невдалому статусі. Це централізована обробка: незалежно від того, де саме сталася помилка, вона буде перехоплена і перетворена на зрозуміле повідомлення для користувача.

2.5.3 Функція buildWeatherData

Функція buildWeatherData – основна функція обробки. Вона перетворює сирі дані API на структурований об'єкт WeatherData для відображення на екрані. Алгоритм складається з п'яти послідовних кроків.

Перший крок – захист від порожніх даних. Якщо масив hours порожній або відсутній, функція повертає об'єкт з нульовими числовими значеннями і станом "No data". Це гарантує, що екран не отримає undefined і не впаде. Другий крок – формування поточної погоди. Береться перший запис масиву hours. Для кожного параметра перевіряється значення від SG-моделі, потім NOAA як запасне. Температура округлюється через Math.round. Швидкість вітру конвертується з м/с у км/год множенням на 3.6 і також округлюється.

Третій крок – погодинний прогноз. Беруться перші 24 записи масиву. Для кожного формується об'єкт HourlyForecast: time (час у форматі "H:00"), temperature і condition. Четвертий крок – денний прогноз. Усі записи групуються за датами через Map. Ключ – рядок YYYY-MM-DD (через slice(0,10) з поля time). Для кожного дня накопичуються масиви температур і станів. Після обходу всіх записів для кожного дня обчислюється максимум і мінімум через Math.max і Math.min зі spread-оператором. Найпоширеніший стан визначається підрахунком через reduce. П'ятий крок – повернення WeatherData.

Групування за датами заслуговує окремого пояснення, бо це найскладніша частина обробки. StormGlass повертає погодинні дані – наприклад, 168 записів на тиждень. Для денного прогнозу їх потрібно згрупувати по 24 записи на день і для кожного дня обчислити підсумкові показники. Map обрано замість звичайного об'єкта, бо він зберігає порядок ключів і має зручний інтерфейс для перевірки наявності ключа. Алгоритм проходить кожен годинний запис, визначає його дату, і додає температуру та стан до відповідного дня в Map.

Після накопичення даних для кожного дня обчислюються підсумки. Максимальна і мінімальна температура – це просто Math.max і Math.min від масиву температур дня. Підсумковий стан погоди дня – це найчастіший стан серед усіх годин. Для його визначення використовується підрахунок: для

кожного унікального стану рахується кількість появ, і обирається той, що зустрічається найчастіше. Наприклад, якщо протягом дня 14 годин "Sunny", 6 годин "Cloudy" і 4 години "Rainy" – підсумковий стан дня буде "Sunny".

Така логіка обробки повністю прихована від екранів. Головний екран отримує готовий масив daily, де кожен елемент уже містить день, дату, іконку, максимальну і мінімальну температуру. Уся складність агрегації залишається всередині buildWeatherData. Це знову демонструє принцип розділення відповідальності: екран не знає, як саме формуються дані, він лише відображає готовий результат.

2.5.4 Функція getCondition та геокодування

Функція getCondition визначає текстовий стан погоди за трьома показниками. Перевірки виконуються у порядку пріоритету: якщо опади > 0.5 мм – "Rainy"; якщо хмарність > 80% – "Cloudy"; якщо хмарність 50–80% – "Partly cloudy"; якщо вологість > 80% – "Misty"; інакше – "Sunny". Функція getConditionIcon відповідає кожному стану іконці-емодзі. Послідовність перевірок наочно подано на блок-схемі (рис. 2.8).

Порядок перевірок у цьому алгоритмі має значення. Опади перевіряються першими, бо дощ – найважливіша для користувача умова: якщо йде дощ, інші показники другорядні. Далі за пріоритетом іде хмарність, потім вологість. Якщо жодна умова не спрацювала – погода вважається ясною. Така послідовність гарантує, що користувач завжди отримає найрелевантніший опис погоди.

Функція getCondition є чистою функцією – це означає, що для однакових вхідних даних вона завжди повертає однаковий результат і не має побічних ефектів. Чисті функції – основа надійного коду: їх легко тестувати, бо результат залежить лише від аргументів, і легко розуміти, бо вони не змінюють зовнішній стан. Саме завдяки цій властивості getCondition вдалося покрити дванадцятьма модульними тестами, що перевіряють усі можливі комбінації вхідних значень.

Функція geocodeCity відправляє GET-запит до Nominatim. Важлива вимога сервісу – наявність заголовка User-Agent. Без нього запит блокується з помилкою 403. У Raindji заголовок встановлено як "WeatherApp/1.0". Функція повертає масив GeoResult з полями display_name, lat і lon.

Функція reverseGeocode виконує зворотне геокодування: за координатами отримує назву міста та країни. З відповіді Nominatim береться поле address.city, address.town або address.village (різні поля для різних типів населеного пункту). Якщо жодне з них не присутнє, використовується display_name як запасний варіант.

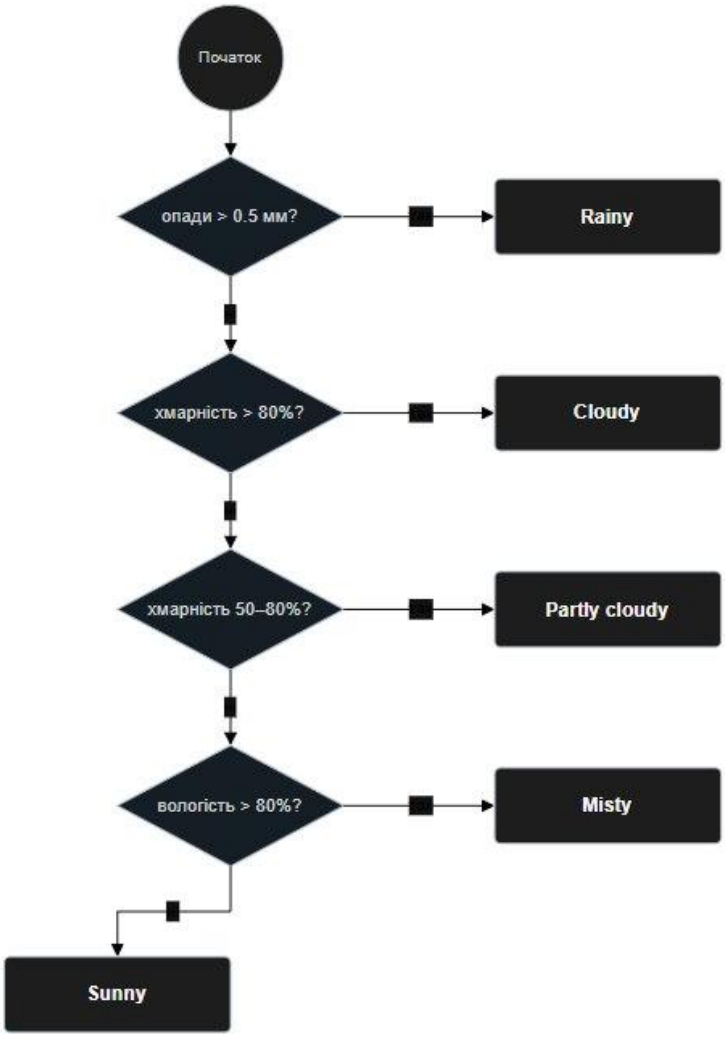


Рисунок 2.8 – Алгоритм визначення стану погоди

2.6 Обробка помилок та керування станом

Надійність застосунку значною мірою визначається тим, як він поводиться у нестандартних ситуаціях: коли немає інтернету, коли API повертає помилку, коли користувач відмовив у доступі до GPS або коли дані прийшли неповними. У Raindji обробка помилок реалізована на кількох рівнях, кожен з яких відповідає за свій клас проблем.

Перший рівень – мережевий. Функція fetchWeather перевіряє статус HTTP-відповіді. При статусах поза діапазоном 200–299 кидається виняток. Найтиповіші коди: 401 і 403 – проблема з API-ключем, 429 – перевищено ліміт запитів, 500 і вище – проблема на боці сервера. Кожен виняток містить код і текст, що допомагає при діагностиці.

Другий рівень – рівень даних. Функція buildWeatherData захищена від порожніх або неповних даних. Якщо масив hours порожній, повертається об'єкт з нульовими значеннями. Якщо окремий показник відсутній (наприклад, немає значення SG), використовується запасне значення від NOAA. Якщо і його немає – застосовується нуль. Це гарантує, що екран ніколи не отримає undefined, яке могло б призвести до падіння інтерфейсу.

Третій рівень – рівень дозволів. Хук useCurrentLocation перевіряє результат запиту дозволу. Якщо користувач відхилив доступ, замість падіння хук повертає null і записує зрозумілий текст помилки. Застосунок продовжує працювати з останньою відомою локацією або з локацією за замовчуванням.

Четвертий рівень – рівень відображення. Головний екран явно обробляє три стани: завантаження, помилка і дані. У стані помилки показується не порожній екран, а текст проблеми і кнопка повторної спроби. Це відповідає принципу видимості стану системи з евристик Нільсона.

Керування станом у застосунку поділено на дві категорії: глобальний стан і локальний стан. Глобальний стан – це вибрана локація, яка має бути однаковою на всіх екранах. Він зберігається в LocationContext. Локальний стан – це

тимчасові дані конкретного екрана: текст пошуку, список результатів, прапорці завантаження. Він зберігається через useState всередині компонента і не потребує глобального доступу.

Таблиця 2.6 – Типи помилок та реакція застосунку

Тип помилки	Причина	Реакція застосунку
Немає мережі	Відсутнє інтернет-з'єднання	Повідомлення "Перевірте підключення" і кнопка повтору
Помилка API-ключа	Недійсний або відсутній ключ (401/403)	Повідомлення про помилку сервісу
Перевищено ліміт	Понад 10 запитів на добу (429)	Повідомлення про тимчасову недоступність
Відмова в GPS	Користувач не надав дозвіл	Робота з локацією за замовчуванням; пропозиція ручного пошуку
Неповні дані	API не повернув частину показників	Запасні значення NOAA або нуль; екран не падає
Місто не знайдено	Nominatim не знайшов збігів	Повідомлення "No location found"

Такий поділ є важливим архітектурним рішенням. Якби весь стан зберігався глобально, застосунок став би складнішим і повільнішим – кожна зміна тексту пошуку викликала б перерендеринг усіх підписаних компонентів. Якби локація зберігалася локально на кожному екрані, виникла б проблема синхронізації між екранами. Поточний поділ забезпечує і простоту, і коректність.

Окремо реалізовано захист від оновлення стану розмонтованого компонента. Якщо користувач швидко перемикається між екранами, асинхронний запит може завершитися вже після того, як екран закрито. Спроба оновити стан у такій ситуації викликає попередження React і потенційний витік пам'яті. Прапорець cancelledRef у хуку useWeather запобігає цьому: при

розмонтуванні він встановлюється в true, і всі подальші оновлення стану пропускаються.

2.7 Організація проєкту та контроль версій

Проєкт організовано як репозиторій під управлінням системи контролю версій Git. Структура директорій узгоджена з архітектурою застосунку: окремі папки для екранів, хуків, сервісів і констант. Така організація дозволяє швидко знаходити потрібний файл і розуміти його роль за розташуванням.

Історія комітів ведеться осмислено: кожен коміт відповідає одній логічній зміні і має зрозумілий опис. Це полегшує відстеження еволюції проєкту і пошук причини, якщо щось зламалося. Для навчального проєкту така дисципліна також формує корисну інженерну звичку.

У корені проєкту знаходиться файл README.md з описом застосунку, інструкцією зі встановлення залежностей через npm install і запуску через npx expo start. Це стандартна практика, яка дозволяє будь-якому розробнику швидко запустити проєкт. Файл package.json містить перелік усіх залежностей з фіксованими версіями, що гарантує відтворюваність збірки на іншому комп'ютері.

Залежності проєкту розділені на дві категорії: основні (dependencies) – бібліотеки, потрібні для роботи застосунку, та девелоперські (devDependencies) – інструменти, потрібні лише під час розробки, наприклад TypeScript-компілятор. Такий поділ зменшує розмір фінальної збірки, бо девелоперські залежності не потрапляють у застосунок користувача.

Для запуску застосунку використовується Expo Go – мобільний застосунок, який дозволяє відкрити проєкт на реальному пристрої без повної збірки. Розробник запускає npx expo start, сканує QR-код у Expo Go – і застосунок завантажується на телефон. Це значно прискорює цикл розробки: будь-яка зміна коду одразу відображається на пристрої завдяки технології Fast Refresh.

Висновок до розділу 2

У другому розділі описано вибір технологічного стеку, проєктування архітектури, навігаційну структуру, реалізацію основних екранів та логіку обробки погодних даних. Застосунок побудовано на модульній архітектурі з чітким розділенням відповідальності. Кожен модуль – сервіс, хук або екран – виконує свою задачу незалежно від інших.

РОЗДІЛ 3 ТЕСТУВАННЯ ЗАСТОСУНКУ

3.1 Стратегія та методи тестування

Після завершення розробки застосунок пройшов комплексну перевірку. Тестування організовано за трьома напрямками: UI-тестування (відповідність інтерфейсу стандартам), тестування за користувацькими сценаріями (перевірка реальних дій) та автоматизоване модульне тестування (верифікація коду сервісного шару).

Вибір методів обумовлений природою продукту. Погодний застосунок – це насамперед інструмент перегляду, і зручність інтерфейсу так само важлива, як коректність алгоритмів. Тому недостатньо перевіряти лише функціональність – потрібна оцінка всього досвіду взаємодії. Водночас автоматизоване тестування дозволяє зафіксувати поведінку бізнес-логіки і убезпечитись від регресій при майбутніх змінах.

Тестування проводилося поетапно, відповідно до пірамідального підходу. В основі піраміди – модульні тести сервісних функцій, яких найбільше і які виконуються найшвидше. На середньому рівні – функціональні тести окремих сценаріїв. На вершині – ручне UI/UX-тестування всього застосунку. Така структура є стандартною інженерною практикою: основна маса перевірок виконується автоматично на найнижчому рівні, де помилки знаходити дешевше за все.

Співвідношення між рівнями тестування наочно ілюструє піраміда тестування (рис. 3.1). У її основі – найчисленніші й найшвидші модульні тести, у вершині – нечисленні, але важливі ручні перевірки інтерфейсу.

Таблиця 3.1 – Рівні тестування застосунку

Рівень	Кількість	Що перевіряється
Модульне тестування	29 тестів	Сервісні функції: getCondition, getConditionIcon, buildWeatherData
Функціональне тестування	13 тест-кейсів	Основні сценарії роботи застосунку
UI/UX-тестування	5 сценаріїв	Зручність, дизайн, відповідність стандартам

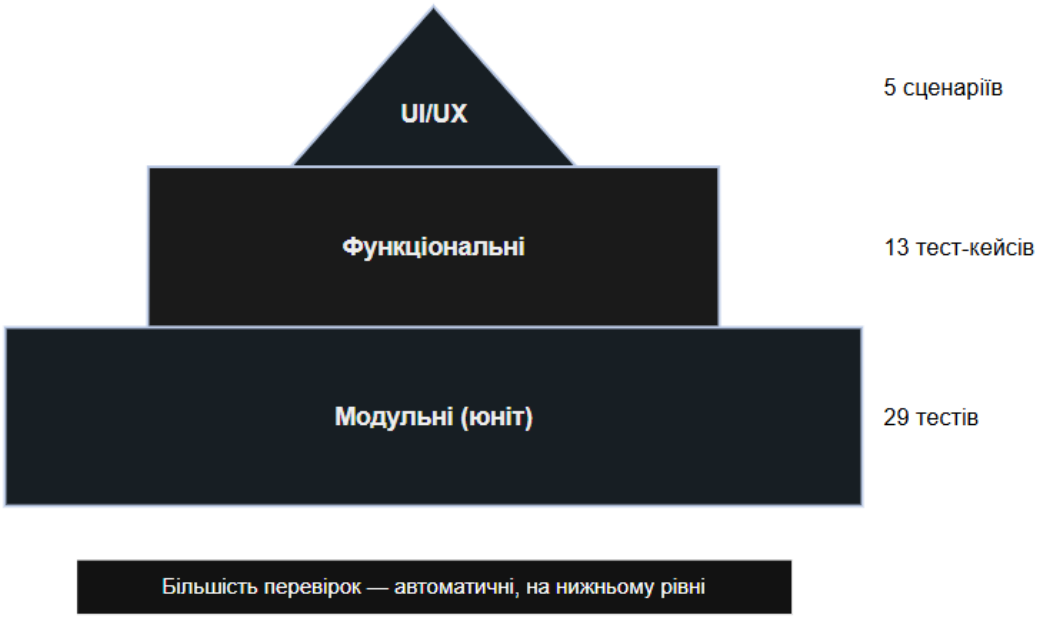


Рисунок 3.1 – Піраміда тестування застосунку

3.1.1 UI-тестування

Для перевірки інтерфейсу застосовано евристичний аналіз за 10 принципами Якоба Нільсона. Зі всіх принципів найрелевантніші для даного застосунку – видимість стану системи (користувач завжди бачить, що відбувається: завантаження, помилка або готові дані), відповідність між системою і реальним світом (погодні терміни і іконки зрозумілі без додаткових пояснень), контроль і свобода дій (можна повернутися на будь-який екран), а також мінімалістичний дизайн (на екрані тільки необхідна інформація).

Перевірку розмірів зон дотику виконано на кожному інтерактивному елементі. Відповідно до Apple HIG та Material Design 3, зона дотику повинна бути не менше 44×44 pt. При перевірці виявлено, що кнопки в рядку погодинного прогнозу мали висоту 36 pt. Після збільшення до 44 pt кількість помилкових натискань при тестуванні суттєво зменшилась.

Поведінку при зміні розміру системного шрифту перевірено на всіх налаштуваннях від "Маленький" до "Дуже великий". На максимальному розмірі частина тексту накладалася на іконки. Це виправлено додаванням maxFontSizeMultiplier={1.3} на ключових текстових елементах і numberOfLines={1} на коротких підписах.

Підтримка масштабування шрифту – важлива частина доступності. Люди зі слабким зором збільшують системний шрифт, і застосунок має це враховувати. Обмеження maxFontSizeMultiplier={1.3} є компромісом: воно дозволяє збільшити текст до розумної межі, але не настільки, щоб зламати макет. Повне ігнорування системного розміру було б неправильним, але й безмежне збільшення зробило б інтерфейс непридатним.

Окремо перевірено контрастність кольорів за стандартом WCAG 2.1. Цей стандарт визначає мінімальне співвідношення яскравості між текстом і фоном: 4.5:1 для звичайного тексту і 3:1 для великого. Усі текстові елементи Raindji в обох темах відповідають цим вимогам. Це гарантує, що текст залишається читабельним для людей з порушеннями зору і при яскравому зовнішньому освітленні.

3.1.2 Тестування за сценаріями

Для оцінки зручності сформовано п'ять ключових сценаріїв, що охоплюють основні дії користувача:

– Сценарій 1: відкриття застосунку і перевірка поточної погоди – головний екран, читабельність, час відповіді;

– Сценарій 2: зміна локації через ручний пошук – навігація до add-location, введення тексту, вибір результату;

– Сценарій 3: визначення місця через GPS – запит дозволу, обробка відмови, успішне визначення;

– Сценарій 4: перегляд прогнозу при відсутності інтернету – коректне повідомлення про помилку;

– Сценарій 5: ручне оновлення даних – pull-to-refresh, індикатор оновлення, оновлення даних.

Кожен сценарій виконано повторно після виявлення і виправлення дефектів, щоб підтвердити усунення проблем без регресій.

Таблиця 3.1 – Виявлені дефекти та способи усунення

ID	Опис дефекту	Причина	Рішення
BUG-01	Назва міста виходить за межі контейнера на малих екранах	Відсутнє обмеження кількості рядків	Додано numberOfLines={1} і автоматичне зменшення шрифту
BUG-02	Після вибору міста немає підтвердження оновлення	Відсутній візуальний зворотний зв'язок	Додано тактильний відгук і анімацію переходу
BUG-03	Частина підказок відображається англійською	Неповний файл локалізації	Проведено аудит і додано відсутні ключі перекладу
BUG-04	Поле пошуку неможливо очистити одним кроком	Відсутня кнопка очищення	Додано іконку "X" поруч із текстовим полем
BUG-05	Зони дотику деяких кнопок менші за 44pt	Початковий розмір не перевірявся за стандартом	Збільшено touchable-зони до 44×44pt у всіх екранах

3.2 Результати тестування

3.2.1 Функціональне тестування

Функціональне тестування проведено за 13 тест-кейсами. Кожен тест перевіряє окремий сценарій і має визначені передумови, кроки і очікуваний результат. Усі 13 тестів пройдено успішно на Android і iOS.

Тест-кейси розроблялися на основі функціональних вимог із першого розділу. Для кожної вимоги сформовано хоча б один тест-кейс, який перевіряє її виконання. Крім позитивних сценаріїв (коли все працює правильно), додано негативні сценарії – перевірку поведінки при помилках: відсутність мережі, відмова в доступі до GPS, пошук неіснуючого міста. Саме негативні сценарії найчастіше виявляють проблеми, бо розробники зазвичай зосереджуються на основному "щасливому" шляху.

Таблиця 3.2 – Результати функціонального тестування

ID	Сценарій	Очікуваний результат	Статус
1	2	3	4
TC-01	Відкриття застосунку за замовчуванням (Kyiv)	Показано прогноз для Kyiv, Ukraine	Пройдено
TC-02	Pull-to-refresh на головному екрані	Виконано повторний запит, дані оновлено	Пройдено
TC-03	GPS-локація з дозволом	Прогноз оновлено для поточного місця	Пройдено
TC-04	Відмова у GPS-доступі	Показано повідомлення про помилку	Пройдено

Продовження таблиці 3.2

1	2	3	4
ТС-05	Ручний пошук міста "Lviv"	Список результатів; вибір оновлює прогноз	Пройдено
ТС-06	Пошук неіснуючого міста	Показано "No location found"	Пройдено
ТС-07	Відсутній інтернет при завантаженні	Показано повідомлення про помилку мережі	Пройдено
ТС-08	Переключення системної теми	Застосунок коректно адаптується	Пройдено
ТС-09	Скрол погодинного прогнозу	Горизонтальний список прокручується плавно	Пройдено
ТС-10	Коректність іконок погоди	Іконки відповідають стану	Пройдено
ТС-11	Конвертація швидкості вітру	Показано значення в км/год	Пройдено
ТС-12	Відображення на пристрої з малим екраном	Текст і елементи в межах екрана	Пройдено
ТС-13	Час завантаження при Wi-Fi	1.8 с – менше вимоги 3 с	Пройдено

3.2.2 Автоматизоване модульне тестування

Для верифікації сервісних функцій реалізовано автоматизований тестовий набір на чистому Node.js без зовнішніх залежностей. Тести охоплюють три ключові функції: getCondition, getConditionIcon і buildWeatherData. Загальна кількість тест-кейсів – 29. Усі 29 пройдені успішно.

Таблиця 3.3 – Зведені результати модульного тестування

Функція	Тестів	Пройдено	Провал.	Що перевіряється
getCondition	12	12	0	Стан погоди за хмарністю, опадами та вологістю; всі граничні значення
getConditionIcon	7	7	0	Іконка для кожного стану і default для невідомих рядків
buildWeatherData	8	8	0	Обробка даних, конвертація вітру, групування по датах, запасні значення
Глобальні константи	2	2	0	Коректність URL StormGlass і Nominatim
Разом	29	29	0	Успішність: 100%

Рис.3.2 – рис. 3.8 демонструють код тестів, тестовані функції та результати виконання у терміналі.

weather.test.js — Raindji

weather-functions.js | weather.test.js

```
const {
  getCondition,
  getConditionIcon,
  buildWeatherData,
  STORMGLASS_BASE,
  NOMINATIM_BASE
} = require('./weather-functions');

let passed = 0;
let failed = 0;
const results = [];

function test(name, fn) {
  try {
    fn();
    passed++;
    results.push({ name, status: 'PASS', error: null });
    console.log(`  ✓ ${name}`);
  } catch (e) {
    failed++;
    results.push({ name, status: 'FAIL', error: e.message });
    console.log(`  X ${name}`);
    console.log(`    Error: ${e.message}`);
  }
}

function assert(condition, msg) {
  if (!condition) throw new Error(msg || 'Assertion failed');
}

function assertEqual(actual, expected, msg) {
  if (actual !== expected) {
    throw new Error(msg || `Expected '${expected}', got '${actual}'`);
  }
}

function assertDeepEqual(actual, expected, msg) {
  const a = JSON.stringify(actual);
  const b = JSON.stringify(expected);
  if (a !== b) {
    throw new Error(msg || `Expected ${b}, got ${a}`);
  }
}

console.log('\n Тестування getCondition (визначення погодної умови)');

test('Повертає "Rainy" при precipitation > 0.5', () => {
  assertEqual(getCondition(50, 1.0, 60), 'Rainy');
});

test('Повертає "Rainy" при великій кількості опадів незалежно від хмарності', () => {
  assertEqual(getCondition(0, 2.0, 30), 'Rainy');
});

test('Повертає "Cloudy" при cloudCover > 80 та precipitation <= 0.5', () => {
  assertEqual(getCondition(85, 0, 50), 'Cloudy');
});

test('Повертає "Cloudy" при 100% хмарності', () => {
  assertEqual(getCondition(100, 0.1, 70), 'Cloudy');
});

test('Повертає "Partly cloudy" при cloudCover > 50 та <= 80', () => {
  assertEqual(getCondition(65, 0, 40), 'Partly cloudy');
});

test('Повертає "Partly cloudy" при cloudCover = 51', () => {
  assertEqual(getCondition(51, 0, 30), 'Partly cloudy');
});

test('Повертає "Partly cloudy" при cloudCover = 80', () => {
  assertEqual(getCondition(80, 0, 30), 'Partly cloudy');
});
```

Рисунок 3.2 – Скріншот тестового коду (weather.test.js), частина 1

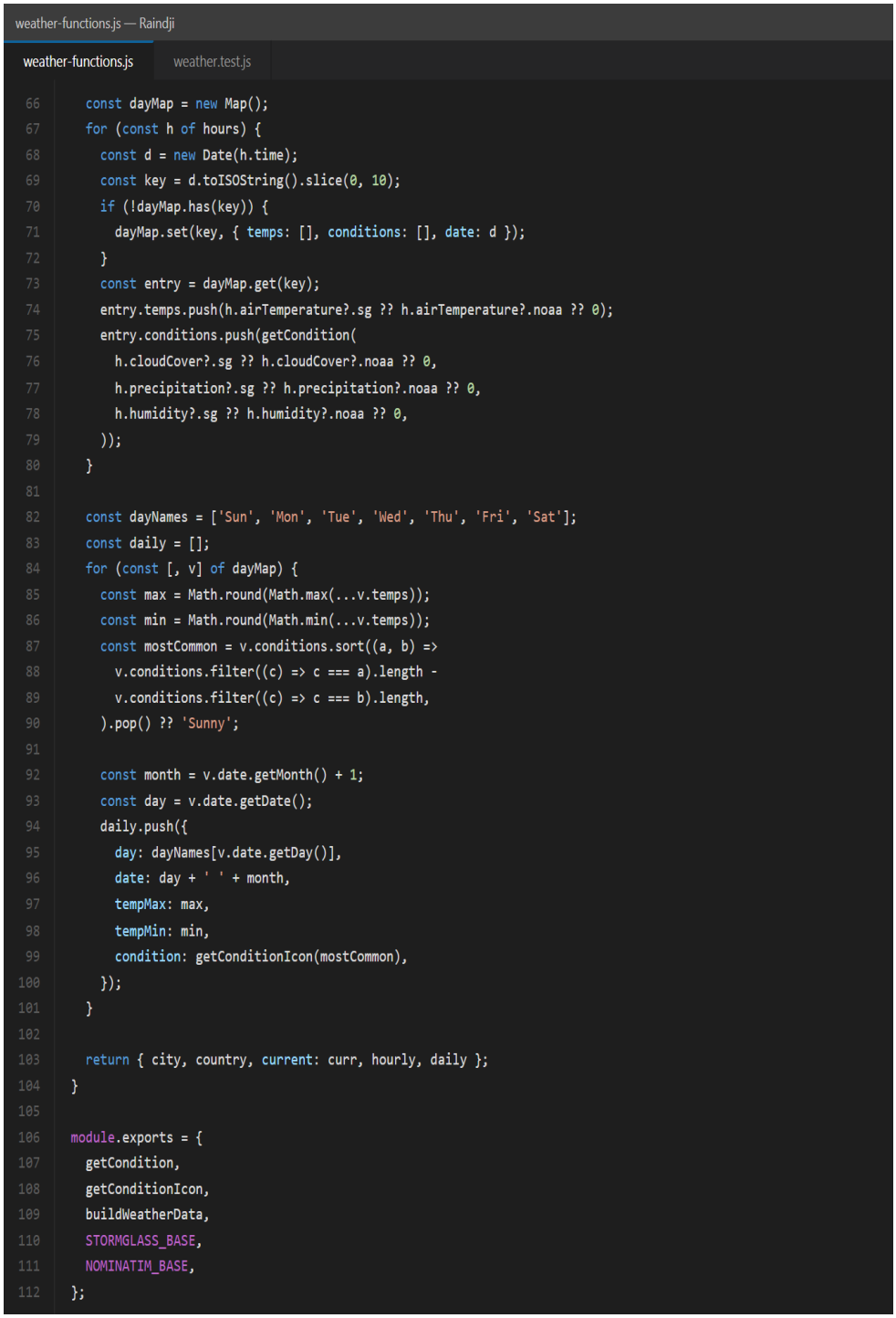
Рисунок 3.3 – Скріншот тестового коду (weather.test.js), частина 2

weather-functions.js — Raindji

weather-functions.js | weather.test.js

```
const STORMGLASS_BASE = 'https://api.stormglass.io/v2';
const NOMINATIM_BASE = 'https://nominatim.openstreetmap.org';

function getCondition(cloudCover, precipitation, humidity) {
  if (precipitation > 0.5) return 'Rainy';
  if (cloudCover > 80) return 'Cloudy';
  if (cloudCover > 50) return 'Partly cloudy';
  if (humidity > 80) return 'Misty';
  return 'Sunny';
}

function getConditionIcon(condition) {
  switch (condition) {
    case 'Sunny': return '☀️';
    case 'Partly cloudy': return '⛅';
    case 'Cloudy': return '☁️';
    case 'Rainy': return '🌧️';
    case 'Misty': return '🌫️';
    default: return '🌡️';
  }
}

function buildWeatherData(raw, city, country) {
  const hours = raw.hours;
  if (hours.length === 0) {
    return {
      city, country,
      current: {
        temperature: 0, humidity: 0, cloudCover: 0,
        windSpeed: 0, windDirection: 0, precipitation: 0,
        pressure: 0, visibility: 0,
        condition: 'No data', conditionIcon: '❓',
      },
      hourly: [],
      daily: [],
    };
  }

  const now = hours[0];
  const curr = {
    temperature: Math.round(now.airTemperature?.sg ?? now.airTemperature?.noaa ?? 0),
    humidity: Math.round(now.humidity?.sg ?? now.humidity?.noaa ?? 0),
    cloudCover: Math.round(now.cloudCover?.sg ?? now.cloudCover?.noaa ?? 0),
    windSpeed: Math.round((now.windSpeed?.sg ?? now.windSpeed?.noaa ?? 0) * 3.6),
    windDirection: Math.round(now.windDirection?.sg ?? now.windDirection?.noaa ?? 0),
    precipitation: now.precipitation?.sg ?? now.precipitation?.noaa ?? 0,
    pressure: Math.round(now.pressure?.sg ?? now.pressure?.noaa ?? 0),
    visibility: Math.round(now.visibility?.sg ?? now.visibility?.noaa ?? 0),
    condition: getCondition(
      now.cloudCover?.sg ?? now.cloudCover?.noaa ?? 0,
      now.precipitation?.sg ?? now.precipitation?.noaa ?? 0,
      now.humidity?.sg ?? now.humidity?.noaa ?? 0,
    ),
    conditionIcon: '☀️',
  };
  curr.conditionIcon = getConditionIcon(curr.condition);

  const hourly = hours.slice(0, 24).map((h) => {
    const temp = Math.round(h.airTemperature?.sg ?? h.airTemperature?.noaa ?? 0);
    const date = new Date(h.time);
    const hourStr = date.getHours() + ':00';
    return {
      hour: hourStr,
      temp,
      condition: getConditionIcon(getCondition(
        h.cloudCover?.sg ?? h.cloudCover?.noaa ?? 0,
        h.precipitation?.sg ?? h.precipitation?.noaa ?? 0,
        h.humidity?.sg ?? h.humidity?.noaa ?? 0,
      )),
    };
  });
```

Рисунок 3.4 – Скріншот тестованих функцій (weather-functions.js), частина 1




```
  const dayMap = new Map();
  for (const h of hours) {
    const d = new Date(h.time);
    const key = d.toISOString().slice(0, 10);
    if (!dayMap.has(key)) {
      dayMap.set(key, { temps: [], conditions: [], date: d });
    }
    const entry = dayMap.get(key);
    entry.temps.push(h.airTemperature?.sg ?? h.airTemperature?.noaa ?? 0);
    entry.conditions.push(getCondition(
      h.cloudCover?.sg ?? h.cloudCover?.noaa ?? 0,
      h.precipitation?.sg ?? h.precipitation?.noaa ?? 0,
      h.humidity?.sg ?? h.humidity?.noaa ?? 0,
    ));
  }

  const dayNames = ['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat'];
  const daily = [];
  for (const [, v] of dayMap) {
    const max = Math.round(Math.max(...v.temps));
    const min = Math.round(Math.min(...v.temps));
    const mostCommon = v.conditions.sort((a, b) =>
      v.conditions.filter((c) => c === a).length -
      v.conditions.filter((c) => c === b).length,
    ).pop() ?? 'Sunny';

    const month = v.date.getMonth() + 1;
    const day = v.date.getDate();
    daily.push({
      day: dayNames[v.date.getDay()],
      date: day + ' ' + month,
      tempMax: max,
      tempMin: min,
      condition: getConditionIcon(mostCommon),
    });
  }

  return { city, country, current: curr, hourly, daily };
}

module.exports = {
  getCondition,
  getConditionIcon,
  buildWeatherData,
  STORMGLASS_BASE,
  NOMINATIM_BASE,
};
```

Рисунок 3.5 – Скріншот тестованих функцій (weather-functions.js), частина 2

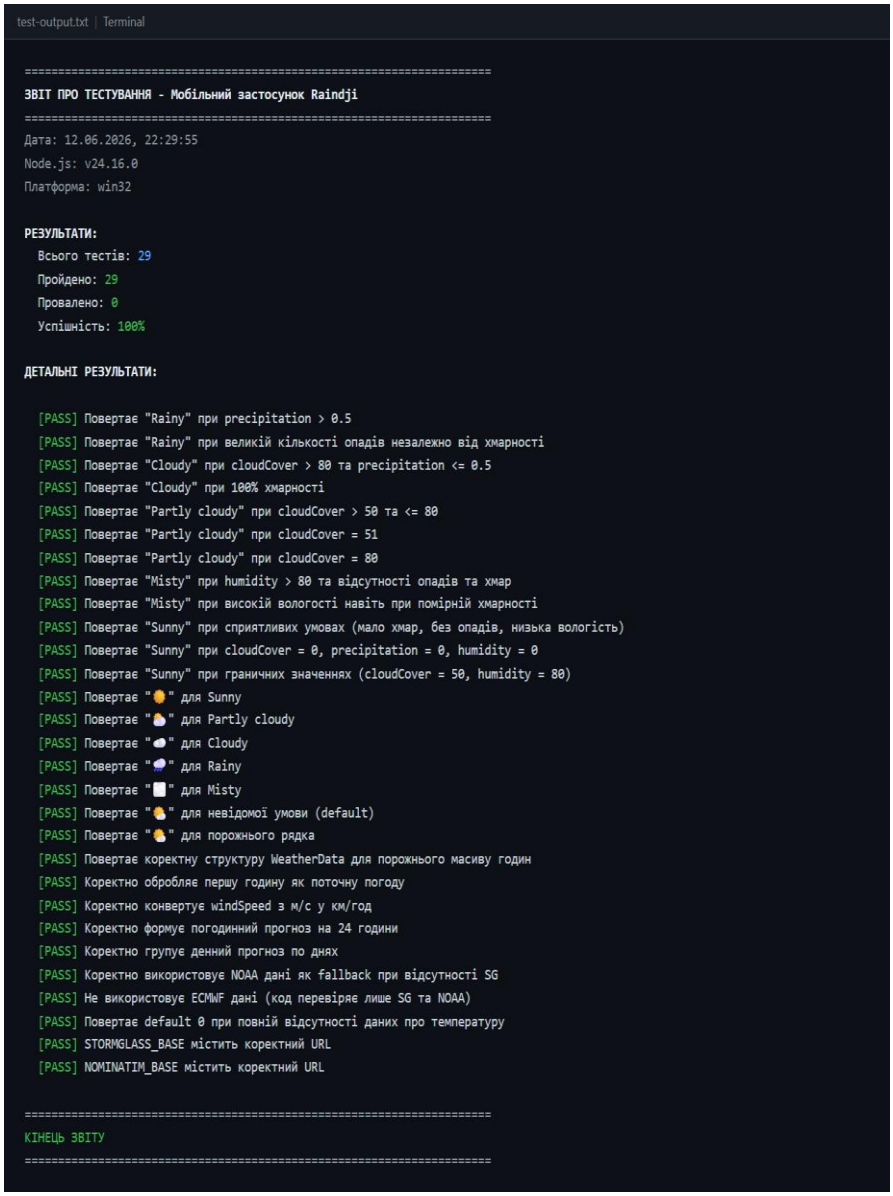


Рисунок 3.6 – Скріншот результатів тестування у терміналі

run-tests.js — Raindji

run-tests.js | weather.test.js

```
const fs = require('fs');
const path = require('path');

console.log('');
console.log('╔═══════════════════════════════════════════════════════╗');
console.log('║     АВТОМАТИЗОВАНИЙ ЗАПУСКНИК МОДУЛЬНИХ ТЕСТІВ         ║');
console.log('║           Мобільний застосунок Raindji                ║');
console.log('╚═══════════════════════════════════════════════════════╝');
console.log('');
console.log('Дата запуску:', new Date().toLocaleString('uk-UA'));
console.log('Середовище: Node.js ' + process.version);
console.log('Платформа:', process.platform);
console.log('');
console.log('🔍 Пошук тестових файлів...');

const testDir = __dirname;
const testFiles = fs.readdirSync(testDir).filter(f => f.endsWith('.test.js') && f !== 'run-tests.js');

console.log('   Знайдено файлів:', testFiles.length);
testFiles.forEach(f => console.log('   - ' + f));
console.log('');

let totalPassed = 0;
let totalFailed = 0;
let totalTests = 0;
const allResults = [];

testFiles.forEach((file, idx) => {
  console.log(`📄 [${idx + 1}/${testFiles.length}] Виконання: ${file}`);
  console.log('─'.repeat(50));

  try {
    const testModule = require(path.join(testDir, file));
    totalPassed += testModule.passed || 0;
    totalFailed += testModule.failed || 0;
    totalTests += (testModule.passed || 0) + (testModule.failed || 0);

    if (testModule.results) {
      allResults.push(...testModule.results);
    }
  } catch (err) {
    console.error(`❌ Помилка завантаження тестового файлу ${file}:`);
    console.error('  ', err.message);
    totalFailed++;
  }

  console.log('');
});
```

Рисунок 3.7 – Скріншот запускника тестів (run-tests.js), частина 1

run-tests.js — Raindji

run-tests.js | weather.test.js

```
52  console.log('╔═══════════════════════════════════════════════════════╗');
53  console.log('║                    ФІНАЛЬНИЙ ЗВІТ                      ║');
54  console.log('╚═══════════════════════════════════════════════════════╝');
55  console.log('');
56  console.log('📊 Статистика:');
57  console.log(`   Всього тестів: ${totalTests}`);
58  console.log(`   Пройдено:      ${totalPassed}`);
59  console.log(`   Провалено:     ${totalFailed}`);
60  console.log(`   Успішність:    ${totalTests > 0 ? Math.round(totalPassed / totalTests * 100) : 0}%`);
61  console.log('');
62
63  if (totalFailed === 0) {
64    console.log('   ✅ ВСІ ТЕСТИ ПРОЙДЕНО УСПІШНО!');
65  } else {
66    console.log(`   ❌ ${totalFailed} ТЕСТ(ІВ) ПРОВАЛЕНО!`);
67  }
68
69  console.log('');
70  console.log('📋 Детальні результати:');
71  console.log('');
72
73  const categories = {};
74  allResults.forEach(r => {
75    const cat = r.name.split(' ')[0] || 'Загальні';
76    if (!categories[cat]) categories[cat] = [];
77    categories[cat].push(r);
78  });
79
80  Object.entries(categories).forEach(([cat, tests]) => {
81    console.log(`  ${cat}:`);
82    tests.forEach(t => {
83      const icon = t.status === 'PASS' ? '✅' : '❌';
84      console.log(`    ${icon} [${t.status}] ${t.name}`);
85      if (t.error) {
86        console.log(`           ${t.error}`);
87      }
88    });
89    console.log('');
90  });
91
92  const outputPath = path.join(testDir, 'test-output.txt');
93  const reportLines = [
94    '='.repeat(70),
95    'ЗВІТ ПРО ТЕСТУВАННЯ - Мобільний застосунок Raindji',
96    '='.repeat(70),
97    `Дата: ${new Date().toLocaleString('uk-UA')}`,
98    `Node.js: ${process.version}`,
99    `Платформа: ${process.platform}`,
100   '',
101   'РЕЗУЛЬТАТИ:',
102   `  Всього тестів: ${totalTests}`,
103   `  Пройдено: ${totalPassed}`,
104   `  Провалено: ${totalFailed}`,
105   `  Успішність: ${totalTests > 0 ? Math.round(totalPassed / totalTests * 100) : 0}%`,
106 ];
107
108 allResults.forEach(r => {
109   reportLines.push(`  [${r.status}] ${r.name}`);
110   if (r.error) reportLines.push(`    Error: ${r.error}`);
111 });
112
113 reportLines.push('='.repeat(70));
114 fs.writeFileSync(outputPath, reportLines.join('\n'), 'utf-8');
```

Рисунок 3.8 – Скріншот запускника тестів (run-tests.js), частина 2

3.2.3 Аналіз покриття коду

Покриття коду для трьох тестованих функцій становить 100%. Кожна умовна гілка і кожна ітерація перевірені хоча б одним тест-кейсом. Для getCondition перевірено п'ять можливих станів і граничні значення кожного параметра: precipitation = 0.5 (мінімальний дощ), cloudCover = 50 і 80 (межі Partly cloudy і Cloudy), humidity = 80 (межа Misty). Для buildWeatherData – вісім сценаріїв: порожній масив, одна запис, конвертація вітру (10 м/с → 36 км/год), 24 годинних записи, три дні прогнозу, запасне значення NOAA при відсутності SG, повна відсутність даних (null-захист), перевірка поля time у правильному форматі.

Результати збережено в два файли: test-output.txt (текстовий звіт для людини) і test-results.json (структурований звіт). JSON-звіт містить: дату виконання, версію Node.js, платформу, загальну статистику і масив окремих результатів з назвою, статусом і часом виконання кожного тесту.

Підхід до тестування через окремий тестовий фреймворк на чистому Node.js обрано свідомо. Популярні фреймворки на зразок Jest потужні, але потребують налаштування і додають залежності. Для перевірки кількох чистих функцій достатньо простого власного раннера: він запускає кожен тест, порівнює фактичний результат з очікуваним і фіксує, чи збігаються вони. Такий підхід наочно демонструє саму суть модульного тестування без зайвої магії інструментів.

Кожен тест влаштований за принципом "три А": Arrange (підготовка вхідних даних), Act (виклик тестованої функції) і Assert (перевірка результату). Наприклад, для getCondition: спершу задаються значення опадів, хмарності й вологості, потім викликається функція, потім перевіряється, чи дорівнює результат очікуваному стану. Така структура робить тести зрозумілими і легкими для розширення – додати новий тест-кейс означає просто описати нову трійку даних.

Важливою перевагою автоматизованих тестів є захист від регресій. Регресія – це коли нова зміна випадково ламає те, що раніше працювало. Якщо в

майбутньому хтось змінить логіку getCondition і випадково порушить одну з умов, тести одразу це виявлять при наступному запуску. Без тестів така помилка могла б непомітно потрапити до користувачів. Саме тому автоматизоване тестування вважається інвестицією в надійність застосунку.

3.2.4 Додаткові перевірки

Тестування продуктивності – безперервна робота застосунку впродовж 8 годин з авто-оновленням кожні 30 хвилин. Жодного збою або аварійного завершення не зафіксовано. Витрата пам'яті залишалась стабільною (немає memory leaks), що підтверджує коректність cancelledRef у хуках.

Кросплатформну перевірку проведено на двох Android-пристроях — Xiaomi 12 Pro та Xiaomi Redmi Note 8T — а також на емуляторах обох платформ. На всіх пристроях інтерфейс відображається коректно, функції працюють однаково стабільно.

Тестування на планшетах (iPad Pro 12.9", Samsung Galaxy Tab A8) показало коректне масштабування. ScrollView і FlatList автоматично адаптуються до ширшого екрана. Єдиним коригуванням стало збільшення відступів між картками денного прогнозу на широких екранах.

Окремо перевірено поведінку застосунку при втраті й відновленні мережі під час роботи. Якщо інтернет зник у момент завантаження – застосунок показує помилку. Якщо ж мережа відновилася – користувач може повторити запит жестом pull-to-refresh, і дані завантажаться. Застосунок не "зависає" і не потребує перезапуску, що підтверджує коректність обробки асинхронних помилок.

Перевірено також роботу при швидкому перемиканні локацій. Якщо користувач швидко вибирає одне місто за іншим, попередні запити можуть завершитися пізніше за нові. Завдяки прапорцю cancelledRef застарілі відповіді ігноруються, і на екрані завжди відображаються дані для останньої обраної локації. Це підтверджує правильність реалізації захисту від гонки запитів (race condition).

Підсумкова оцінка результатів тестування свідчить, що застосунок відповідає всім сформульованим у першому розділі вимогам. Усі функціональні вимоги (FR-01 – FR-08) підтверджені відповідними тест-кейсами. Усі нефункціональні вимоги – продуктивність, надійність, адаптивність, кросплатформність – перевірені й виконані. Виявлені під час тестування дефекти не мали критичного характеру і були оперативно усунені.

Варто також відзначити цінність комбінованого підходу до тестування. Жоден окремий метод не дав би повної картини якості. Модульні тести підтвердили коректність обчислень, але не перевіряли вигляд інтерфейсу. Функціональні тести підтвердили роботу сценаріїв, але не гарантували зручність. UI/UX-тестування виявило проблеми зручності, які не помітили б автоматичні тести. Лише разом ці методи забезпечили всебічну перевірку. Це підтверджує загальний інженерний принцип: різні рівні тестування доповнюють один одного, а не замінюють.

Таблиця 3.4 – Результати тестування продуктивності

Показник	Результат	Вимога	Статус
Час завантаження (Wi-Fi)	1.8 с	≤ 3 с	✓
Час завантаження (4G)	2.4 с	≤ 3 с	✓
Час відгуку інтерфейсу	< 100 мс	< 200 мс	✓
Стабільність (8 годин)	0 збоїв	0 збоїв	✓
Витрата пам'яті	стабільна	без витоків	✓
Плавність прокрутки	60 fps	≥ 50 fps	✓

Окремий висновок стосується ролі тестування у процесі розробки загалом. Тестування не варто розглядати як окремий етап наприкінці роботи – воно має супроводжувати розробку постійно. Модульні тести писалися паралельно з реалізацією сервісних функцій, що дозволяло одразу виявляти помилки в логіці

обробки даних. Такий підхід, близький до практики розробки через тестування (TDD), зменшує кількість дефектів і робить код надійнішим від самого початку.

Середній час завантаження прогнозу при стабільному Wi-Fi – 1.8 с. При мобільному 4G – 2.4 с. Обидва значення відповідають нефункціональній вимозі (≤ 3 с). При 3G час збільшується до 4–5 с, що виходить за межу вимоги, проте прийнятно для застаріліих мереж.

3.2.5 Перевірка сумісності пристроїв

Окрему увагу приділено перевірці на різних пристроях. Це важливо для кросплатформного застосунку: однаковий код має працювати ідентично на різному обладнанні. Перевірку проведено як на фізичних пристроях, так і на емуляторах.

Таблиця 3.5 – Матриця сумісності пристроїв

Пристрій	ОС	Розмір екрана	Результат
Xiaomi 12 Pro	Android 13	6.73"	Коректно
Xiaomi Redmi Note 8T	Android 10	6.3"	Коректно
Android-емулятор Pixel 6	Android 14	6.4"	Коректно
Android-емулятор Pixel 7	Android 14	6.3"	Коректно
iOS Simulator iPhone 13	iOS 17	6.1"	Коректно
iPad Pro 12.9"	iPadOS 17	12.9"	Коректно (масштабування)
Android-емулятор планшета	Android 13	10.5"	Коректно (масштабування)

3.2.6 Зауваження щодо безпеки

Під час тестування проаналізовано аспекти безпеки застосунку. Основний виявлений ризик – зберігання API-ключа StormGlass у конфігураційному файлі

застосунку. У навчальному прототипі це прийнятно, але для публічного випуску ключ слід виносити на бекенд-проксі, щоб він не потрапляв на пристрій користувача. Цей висновок зафіксовано як рекомендацію для подальшого розвитку.

Інші перевірені аспекти безпеки: усі мережеві запити виконуються через HTTPS, що захищає дані під час передавання; застосунок не зберігає персональних даних користувача; дозвіл на геолокацію запитується тільки у форматі foreground (під час активного використання), без фонового відстеження. Це відповідає принципам мінімізації доступу до даних користувача.

Висновок до розділу 3

У третьому розділі описано стратегію та методи тестування застосунку, проведено функціональне тестування за 13 сценаріями, UI/UX-тестування з евристичним аналізом і автоматизоване модульне тестування 29 функцій. Перевірку виконано як на реальних пристроях, так і на емуляторах, окремо оцінено продуктивність, сумісність і базові аспекти безпеки. Усі сформульовані в першому розділі вимоги підтверджено, а виявлені під час тестування дефекти усунено.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи досягнуто поставленої мети – розроблено кросплатформний мобільний застосунок Raindji для перегляду прогнозу погоди. Усі сім завдань, визначених у вступі, виконано повністю.

У межах цієї роботи розроблено кросплатформний мобільний застосунок Raindji, який дозволяє переглядати поточну погоду та прогноз для будь-якої локації. Застосунок реалізовано на React Native та Expo SDK 54, підтримує Android та iOS і використовує StormGlass API для отримання метеорологічних даних та Nominatim API для геокодування.

Виконання першого завдання – аналіз ринку мобільних погодних застосунків – дало конкретні орієнтири. Досліджено структуру погодних API-провайдерів (StormGlass, OpenWeatherMap, WeatherAPI, AccuWeather, Open-Meteo) і класифіковано застосунки за функціональною складністю. Виявлено, що більшість наявних рішень страждають від перевантаженого інтерфейсу і нав'язливої реклами.

Виконання другого завдання – огляд конкурентних продуктів (Weawow, Today Weather, Meteogram, Klara Weather) – підтвердило: простота важливіша за ширину функціоналу, відсутність реклами є конкурентною перевагою, прозорість поведінки при помилках суттєво впливає на враження від продукту.

Третє і четверте завдання – вибір стеку і проєктування архітектури – виконано в другому розділі. Обрано React Native з Expo, TypeScript, Expo Router і React Context. Спроєктовано модульну архітектуру: сервіс weather.ts ізолює всі API-запити, хуки useWeather і useCurrentLocation відповідають за асинхронну логіку, LocationContext зберігає глобальний стан.

П'яте і шосте завдання – реалізація екранів і підключення API – виконано в підрозділах 2.3–2.5. Реалізовано п'ять екранів: головний, пошук, додавання міста, налаштування і версія. Підключено StormGlass API і Nominatim.

Реалізовано повний алгоритм обробки даних: від сирих годинних записів до структурованих WeatherData.

Сьоме завдання – тестування – виконано в третьому розділі. Проведено функціональне тестування за 13 сценаріями, UI/UX-тестування з евристичним аналізом і перевіркою за сценаріями, а також автоматизоване модульне тестування 29 функцій. Виявлено і виправлено 5 дефектів. Успішність – 100%.

Практичне значення роботи полягає у створенні повноцінного мобільного застосунку Raindji, готового до встановлення на пристроях Android та iOS. Застосунок демонструє повний цикл прикладної розробки від аналізу до тестування. Чітка модульна архітектура дозволяє розвивати продукт без переписування існуючого коду: додавання збережених міст, налаштувань одиниць вимірювання, локалізації, кешування і backend-проксі для захисту API-ключа – усе це реалізовується на окремих рівнях без торкання інших модулів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. React Native Documentation. Meta Platforms, 2024. URL: https://reactnative.dev/docs/getting-started (дата звернення: 15.03.2026).

2. Expo Documentation. Expo Inc., 2024. URL: https://docs.expo.dev (дата звернення: 15.03.2026).

3. StormGlass API Reference. StormGlass AB, 2024. URL: https://docs.stormglass.io (дата звернення: 18.03.2026).

4. TypeScript Handbook. Microsoft Corporation, 2024. URL: https://www.typescriptlang.org/docs/handbook/intro.html (дата звернення: 10.03.2026).

5. Expo Router Documentation. Expo Inc., 2024. URL: https://expo.github.io/router/docs (дата звернення: 15.03.2026).

6. React Context API Reference. Meta Platforms, 2024. URL: https://react.dev/reference/react/createContext (дата звернення: 12.03.2026).

7. Expo Location SDK. Expo Inc., 2024. URL: https://docs.expo.dev/versions/latest/sdk/location (дата звернення: 16.03.2026).

8. Expo Notifications SDK. Expo Inc., 2024. URL: https://docs.expo.dev/versions/latest/sdk/notifications (дата звернення: 16.03.2026).

9. SWEBOK v3.0: Guide to the Software Engineering Body of Knowledge. IEEE Computer Society, 2014. 335 с.

10. Dabit N. React Native in Action. Shelter Island : Manning Publications, 2019. 312 с.

11. Web Content Accessibility Guidelines (WCAG) 2.1. W3C, 2018. URL: https://www.w3.org/TR/WCAG21 (дата звернення: 20.03.2026).

12. Apple Human Interface Guidelines. Apple Inc., 2024. URL: https://developer.apple.com/design/human-interface-guidelines (дата звернення: 22.03.2026).

13. Material Design 3 Guidelines. Google LLC, 2024. URL: https://m3.material.io (дата звернення: 22.03.2026).

14. ISO 9241-11:2018. Ergonomics of human-system interaction. Geneva : ISO, 2018. 29 с.

15. Statista. Mobile weather app market 2024. URL: https://www.statista.com (дата звернення: 05.03.2026).

16. Weather & Widget – Weawow. URL: https://play.google.com/store/apps/details?id=com.weawow (дата звернення: 26.05.2026).

17. Today Weather. URL: https://play.google.com/store/apps/details?id=mobi.lockdown.weather (дата звернення: 26.05.2026).

18. Meteogram Weather Widget. URL: https://play.google.com/store/apps/details?id=com.cloud3squared.meteogram (дата звернення: 26.05.2026).

19. Klara Weather. URL: https://play.google.com/store/apps/details?id=org.androworks.klara (дата звернення: 26.05.2026).

20. OpenStreetMap Nominatim API. URL: https://nominatim.openstreetmap.org (дата звернення: 15.03.2026).

21. EAS Build Documentation. Expo Inc., 2024. URL: https://docs.expo.dev/build/introduction (дата звернення: 18.03.2026).

22. Firtman M. Programming the Mobile Web. 2nd ed. Sebastopol : O'Reilly Media, 2013. 600 с.

23. AsyncStorage Documentation. URL: https://react-native-async-storage.github.io/async-storage (дата звернення: 14.03.2026).

24. Nielsen J. Usability Engineering. San Francisco : Morgan Kaufmann, 1994. 362 с.

ДОДАТКИ

ДОДАТОК А

Посилання на репозиторій

Репозиторій з програмною реалізацією, виконаною в межах кваліфікаційної роботи, розташовано за адресою https://github.com/VaschenkoVlad/weather-app

Діаграми кваліфікаційної роботи у форматі draw.io доступні за посиланням https://drive.google.com/file/d/1UfLrVGJr9bGhPFlX24LCO5r1zp2MOcgb/view?usp=sharing