**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**

**Сумський державний університет**
Факультет електроніки та інформаційних технологій
Кафедра комп'ютерних наук

«До захисту допущено»
В.о. завідувача кафедри
Ігор ШЕЛЕХОВ
_____
(підпис)
червня 2023 р.

# КВАЛІФІКАЦІЙНА РОБОТА

**на здобуття освітнього ступеня бакалавр**

зі спеціальності 122 – Комп'ютерних наук,
освітньо-професійної програми «Інформатика»
на тему: «Інформаційна система візуалізації метеорологічних даних»
здобувача групи ІН-92 Дзернюка Вадима Юрійовича

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

_____Вадим ДЗЕРНЮК
(підпис)

Керівник,
асистент кафедри комп'ютерних наук,
кандидат фізико-математичних наук
Олександр ВЛАСЕНКО _____
(підпис)

**Суми – 2023**

**Сумський державний університет**
Факультет електроніки та інформаційних технологій
Кафедра комп'ютерних наук

«Затверджую»
В.о. завідувача кафедри
_____ Ігор ШЕЛЕХОВ
(підпис)

# ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

**на здобуття освітнього ступеня бакалавра**

зі спеціальності 122 - Комп'ютерних наук, освітньо-професійної програми «Інформатика»
здобувача групи ІН-92 Дзернюка Вадима Юрійовича

1. Тема роботи: «Інформаційна система візуалізації метеорологічних даних»
затверджую наказом по СумДУ від *«  » червня 2023 р. № 0475-VI*
2. Термін здачі здобувачем кваліфікаційної роботи *до 09 червня 2023 року*
3. Вхідні дані до кваліфікаційної роботи _____
4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити)
*1) Аналіз проблеми предметної області, постановка і формулювання завдань дослідження. 2) Огляд технологій, що використовуються для створення додатку. 3) Розробка інформаційної системи для візуалізації метеорологічних даних. 4) Аналіз результатів та тестування.*
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)_____
6. Консультанти до проекту (роботи), із значенням розділів проекту, що стосується їх

| Розділ | Консультант | Підпис, дата | |
|---|---|---|---|
| | | Завдання видав | Завдання прийняв |
| | | | |

7. Дата видачі завдання «____» _____ 20 ___ р.

Завдання прийняв до виконання _____ Керівник _____
(підпис) (підпис)

# КАЛЕНДАРНИЙ ПЛАН

| № п/п | Назва етапів кваліфікаційної роботи | Термін виконання | Примітка |
|---|---|---|---|
| *1* | *Аналіз проблеми предметної області, постановка і формулювання завдань дослідження* | | |
| *2* | *Огляд технологій, що використовуються для створення додатку* | | |
| *3* | *Розробка інформаційної системи для візуалізації метеорологічних даних* | | |
| *4* | *Аналіз отриманих результатів та тестування* | | |
| *5* | *Оформлення пояснювальної записки до кваліфікаційної роботи* | | |

Здобувач вищої освіти _____ Керівник _____
(підпис) (підпис)

# АНОТАЦІЯ

**Записка**: 37 стр., 16 рис., 18 використаних джерел.

**Обґрунтування актуальності теми роботи** – Тема кваліфікаційної роботи є актуальною, оскільки присвячена відображенню погодних умов шляхом розробки відповідних методів, моделей та інформаційної технології.

**Об’єкт дослідження** — інформаційна система візуалізації метеорологічних даних на основі React Native.

**Мета роботи** — розробка додатку для отримання даних про погоду за допомоги React Native.

**Методи дослідження** — літературний огляд, практична реалізація, тестування й аналіз результатів.

**Результати** — здійснено аналітичний огляд засобів і методів розробки мобільних додатків. На основі зробленого аналізу було обрано методи розробки. Визначені завдання на розробку проекту, описано та обґрунтовано вибір структури додатку. Побудована логічна, та на її основі фізична структури додатку. Також було проведено тестування й реліз додатку.

ІНФОРМАЦІЙНА СИСТЕМА, МЕТЕОРОЛОГІЧНІ ДАНІ, REACT NATIVE, OPENWEATHER API, ZUSTAND

# ЗМІСТ

ВСТУП 6
1. ЛІТЕРАТУРНИЙ ОГЛЯД 8
1.1. Аналіз принципів побудови мобільних додатків 8
1.2. Інструменти для розробки мобільних додатків 8
1.3. Архітектура мобільних додатків 9
1.4. State management й здійснення запитів у мобільних додатках 16
1.5. Постановка задачі 18
2. МЕТОДИКА ВИРІШЕННЯ ЗАДАЧ 20
2.1. Фреймворк React Native 20
2.2. Feature-sliced архітектура мобільного додатка 21
2.3. Zustand як state management й react-query для виконання запитів 24
2.4. Огляд інструментів для розробки 26
3. ПРОГРАМНА РЕАЛІЗАЦІЯ 27
3.1. Створення структури проекту 27
3.2. Управління станом з Zustand 29
3.3. Отримання даних з OpenWeatherAPI за допомогою React Query 30
3.4. Огляд додатку й тестування 31
4. ВИСНОВОК 36
5. СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 38

# ВСТУП

**Актуальність.** Розробка цього додатку є актуальною і важливою задачею, що дозволяє забезпечити користувачам точну, швидку та зручну інформацію про погоду, допомагаючи їм планувати свої дії та приймати розумні рішення на основі актуальних погодних умов.

**Об’єкт дослідження.** Дослідження спрямоване на вивчення та аналіз різних аспектів цього додатку, включаючи архітектурний підхід (feature-sliced), використання бібліотеки стану (Zustand), а також реалізацію запитів до зовнішнього сервісу (за допомогою бібліотеки React Query) та відображення погодних даних з використанням OpenWeatherApp API.

**Предмет дослідження.** Предметом дослідження цієї роботи є розробка та імплементація архітектури feature-sliced, використання бібліотеки стану Zustand та бібліотеки запитів React Query у додатку "Weather App". Основна увага зосереджується на вивченні та застосуванні цих технологій з метою поліпшення структури, розширюваності, керування станом та здійснення мережевих запитів у додатку. Предмет дослідження включає в себе аналіз можливостей, особливостей та переваг кожної з цих технологій та їх застосування для реалізації функціональності погодного додатку.

**Гіпотеза.** Застосування архітектури feature-sliced, бібліотеки стану Zustand та бібліотеки запитів React Query у додатку "Weather App" забезпечує полегшення розробки, покращення ефективності та підвищення зручності використання для користувачів.

**Новизна.** Новизна цієї роботи полягає у таких аспектах:

1. Використання архітектури feature-sliced: В роботі акцентується увага на використанні даної архітектурної парадигми для розробки додатку "Weather App". Це новаторський підхід, який дозволяє структурувати код додатку на основі функціональних блоків і полегшує його розширення та підтримку.

2. Використання бібліотеки стану Zustand: У роботі використовується

Zustand для управління станом додатку. Ця бібліотека є новим підходом до керування станом, який пропонує простий та ефективний спосіб збереження та оновлення стану додатку.

3. Використання бібліотеки запитів React Query: Робота включає використання бібліотеки React Query для здійснення запитів до зовнішнього сервісу (OpenWeatherApp API) та управління отриманими даними. Це новаторський підхід до роботи з мережевими запитами, який забезпечує автоматичне кешування, оптимізацію запитів та покращення продуктивності додатку.

4. Інтеграція з OpenWeatherApp API: Робота включає використання OpenWeatherApp API для отримання актуальної погодної інформації. Ця новизна полягає у використанні зовнішнього сервісу для отримання даних про погоду, що забезпечує користувачам точні й актуальні дані.

**Структура.** Дана робота складається зі вступу, літературного огляду, постановки задачі, вибір методу розв’язання поставленої задачі, опису програмного забезпечення інформаційної системи, висновків, списку використаних джерел та додатків.

# 1. ЛІТЕРАТУРНИЙ ОГЛЯД

## 1.1. Аналіз принципів побудови мобільних додатків

На даний момент є кілька технологій, які успішно використовуються для створення мобільних додатків. Для різних платформ підходять різні мови, тому спочатку потрібно визначитися з цікавою для вас платформою, а далі – з мовою.

Розробка мобільних додатків для Android найчастіше виконується на Java – старій добрій об’єктно-орієнтованій мові, на якій написано більше 90% всіх додатків під Андроїд. За останні півроку більшої популярності набирає нова мова Kotlin. Поки близько 5% додатків в Google Play написані на мові Kotlin, але з кожним роком кількість цих додатків зростає [1].

Якщо говорити про iOS платформу, то тут також використовуються дві основних мови – Objective C, вона ж перша мова, яка була розроблена компанією Apple для створення програмного забезпечення під iOS. А друга мова – це більш просунутий і сучасніший Swift.

Якщо говорити про підтримку старих пропозицій, які були написані раніше, то тут однозначно вам потрібно знати Objective C, бо нові додатки все частіше пишуться саме на Swift.

## 1.2. Інструменти для розробки мобільних додатків

Як зазначено у вступі, я буду використовувати фреймворк React Native [2, 4]. Також буде використовуватися CSS для відображення стилів і JavaScript для всієї логіки мобільного додатку.

CSS – розшифровується як каскадні таблиці стилів, описує, як елементи HTML мають відображатися на екрані, папері чи в інших носіях, може керувати макетом кількох веб-сторінок одночасно.

JavaScript [5] – це легка мова програмування, яку веб-розробники зазвичай використовують для створення більш динамічної взаємодії під час розробки веб-сторінок, програм, серверів і навіть ігор.

Розробники зазвичай використовують JavaScript разом із HTML і CSS. Мова сценаріїв добре працює з CSS у форматуванні елементів HTML. Однак JavaScript все ще підтримує взаємодію з користувачем, чого CSS не може зробити сам по собі.

Також JavaScript має багато бібліотек й фреймворків, що дозволяє виконувати задачі різного призначення.

### 1.3. Архітектура мобільних додатків

Архітектура додатків на React Native [3] – це організаційна структура, яка визначає, як різні компоненти та модулі додатку будуть взаємодіяти між собою. Це відноситься до способу, як компоненти обробляють стан, взаємодіють між собою та як логіка додатку організована.

Зазвичай архітектура React Native включає наступні компоненти:

1. Компоненти React Native, які представляють відображення (View) та бізнес-логіку (Business Logic) додатку.
2. Стейт (State) та дії (Actions) – дозволяють компонентам взаємодіяти зі станом додатку та виконувати дії, що можуть змінювати стан.
3. Роутинг (Routing) – дозволяє організувати навігацію між різними сторінками додатку.
4. Контейнери (Containers) – це компоненти, які об'єднують різні компоненти та забезпечують їх взаємодію між собою.

Архітектура додатку на React Native дозволяє організовувати додаток у такий спосіб, щоб він був зрозумілим та легко зберігався, а також щоб його можна було розширити та розвивати в майбутньому.

Основні види архітектури на React Native:

- Класична архітектура [7] – це простий підхід до побудови додатків на React Native. У цій архітектурі всі компоненти знаходяться в одному файлі і компоненти взаємодіють між собою через властивості та зворотні виклики. Це підхід до розробки, коли весь функціонал додатку розділяється на дві частини: компоненти та контейнери. Компоненти відповідають за відображення інформації, контейнери – за управління станом додатку та зв'язком між компонентами. Це може бути простим та швидким способом побудови додатку, але може стати непрактичним при розробці складних додатків.

Переваги:

1. Простота – цей підхід є досить простим для розуміння та використання. Він не вимагає великої кількості додаткових бібліотек або розширень.
2. Легкість відлагодження – класична архітектура React Native дозволяє легко відлагоджувати додаток, оскільки логіка додатку розділена на дві частини: компоненти та контейнери.
3. Швидкість розробки – цей підхід дозволяє розробляти додаток досить швидко, оскільки він досить простий для розуміння та використання.

Недоліки:

1. Обмеженість – класична архітектура React Native не забезпечує достатньої розширюваності для великих та складних додатків.
2. Проблеми з керуванням станом – цей підхід може призвести до проблем з керуванням станом додатку, особливо якщо додаток має багато станів, що потрібно контролювати.
3. Проблеми зі збереженням коду – великі додатки, розроблені за допомогою класичної архітектури React Native, можуть бути складні для збереження та розуміння коду.

У підсумку, класична архітектура в React Native є простою та легко зрозумілою для більшості розробників, але може мати проблеми зі складністю

та масштабованістю при збільшенні розміру проекту. Також вона може вимагати дублювання коду при розробці для різних платформ.

- Модульна архітектура – підхід до розробки додатків на React Native, який базується на розділенні додатку на невеликі модулі, які можна використовувати в інших додатках або компонентах. У цій архітектурі модулі зазвичай зберігаються у власних файлах та можуть містити різні компоненти та логіку.

Переваги:

1. Розширюваність – кожен модуль може бути доданий або вилучений без впливу на решту додатку.
2. Перевикористання – модульна архітектура дозволяє легко перевикористовувати компоненти та модулі в інших проектах або частинах додатку.
3. Легкість розробки – цей підхід дозволяє розробляти додаток швидко та ефективно, оскільки робота над окремими модулями дозволяє розподілити завдання між розробниками та зосередитися на конкретних частинах додатку.

Недоліки:

1. Потрібність управління залежностями – при використанні багатьох модулів, управління залежностями може стати складним та потребувати додаткових зусиль для їх збирання та розгортання.
2. Складність зв'язку – модульна архітектура може стати складною у випадку, якщо зв'язок між модулями виявиться складним або залежним від інших частин додатку.
3. Недостатність стандартизації – при використанні модульної архітектури, розробники можуть застосовувати різні підходи до розробки та назв компонентів, що може призвести до недостатньої стандартизації та складнощів при розробці.

Загалом, модульна архітектура React Native є досить популярним підходом до розробки, оскільки дозволяє покращити розширюваність та

перевикористання коду, а також полегшити розробку окремих частин додатку. Однак, він може мати свої обмеження та потребувати додаткових зусиль для управління залежностями та зв'язками між модулями.

- Атомна (Atomic) архітектура [8] є однією з популярних архітектур в React Native, яка зосереджена на розбитті компонентів на менші та більш прості елементи. Атомарність означає розділення компонентів на найменші можливі частини – атоми, які можна було б перевикористовувати в більш складних компонентах.

Основна ідея Atomic архітектури полягає в тому, щоб зробити компоненти максимально простими та повторно використовуваними.

Компоненти розбиваються на 5 рівнів:

1. Атоми (Atoms) – це найбільш прості елементи, такі як кнопки, інпути, заголовки тощо. Вони є найменшими частинами, з яких складаються складніші компоненти.

2. Молекули (Molecules) – це більш складні елементи, які складаються з кількох атомів та можуть виконувати певні функції, наприклад, форми для введення даних або список пунктів меню.

3. Організми (Organisms) – це ще більш складні елементи, які складаються з кількох молекул та можуть виконувати більш складні функції, наприклад, головне меню або блоки контенту.

4. Сторінки (Pages) – це вже готові сторінки, які включають в себе кілька організмів та можуть містити додаткову логіку, таку як маршрутизація та взаємодія з API.

5. Шаблони (Templates) – це загальні стилі, які використовуються на всіх рівнях та є основою дизайну.

Переваги:

1. Зменшення залежностей: розділення компонентів на менші елементи дозволяє зменшити залежності між ними та зробити код більш чистим та простим у розумінні.

2. Повторне використання: атомарний підхід дозволяє використовувати компоненти на різних рівнях, що сприяє повторному використанню та зменшенню кількості коду.

3. Зручне тестування: оскільки компоненти є меншими та простішими, то тестування стає більш зручним та ефективним.

4. Масштабованість: з ростом проекту можна додавати нові компоненти та використовувати їх на різних рівнях, що дозволяє збільшувати масштаб проекту без значного збільшення складності коду.

Недоліки:

1. Додатковий рівень складності: декомпозиція компонентів на менші елементи може спричинити додаткову складність в управлінні проектом та розумінні структури коду.

2. Можливість перевантаження компонентів: якщо не правильно декомпозувати компоненти, то може виникнути перевантаження компонентів, що призводить до складності в управлінні проектом та збільшенню часу на розробку.

3. Не підходить для всіх проектів: Atomic архітектура не підходить для всіх типів проектів та може бути перевантажена для менших проектів з невеликою кількістю компонентів.

Уцілому, Atomic архітектура є корисною для великих та середніх проектів, де необхідно зменшити залежності між компонентами та зробити код більш повторно використовуваним. Однак, її застосування варто обдумати в залежності від конкретних потреб проекту та його розміру.

- Feature-Sliced архітектура [9] – це підхід до розробки додатків на React Native, що передбачає організацію коду за функціональністю, або "фічами" (features). Кожна "фіча" включає в себе усі компоненти, стилі, логіку та інші ресурси, пов'язані з цією функціональністю.

Переваги:

1. Чіткі межі між функціональностями: кожна "фіча" відповідає за конкретну функціональність додатку, що дозволяє зробити межі між різними функціональностями чіткими та зрозумілими.

2. Швидка розробка: даний підхід дозволяє розробляти функціональність додатку швидше та ефективніше завдяки чіткій організації коду.

3. Легке тестування: кожна "фіча" може бути легко протестована окремо, що дозволяє зменшити час на тестування та підвищити якість коду.

4. Легка підтримка: оскільки кожна "фіча" має свою власну структуру та логіку, то підтримка коду стає більш простою та зрозумілою.

Недоліки:

1. Складність розгортання: якщо в додатку багато фіч, то збільшується складність налаштування проекту, оскільки кожен розділ має свій власний набір залежностей, конфігурацій та інфраструктури.

2. Збільшена кількість файлів: кожен розділ має власну структуру каталогів і файлів, що може призвести до збільшення загальної кількості файлів у проекті.

3. Суттєва залежність між фічами: коли фіча містить в собі більше однієї залежності, може виникнути проблема з управлінням цими залежностями і розширенням фіч.

4. Високий рівень абстракції: оскільки архітектура Feature-Sliced має високий рівень абстракції, нові розробники можуть знайти складнощі в розумінні проекту та швидкому запуску розробки.

Хоча Feature-Sliced архітектура може бути корисною для середніх і великих проектів, вона може бути перебільшеною для менших проектів. Проекти з Feature-Sliced архітектурою можуть бути складнішими для розробки, але вони забезпечують більшу гнучкість і розширюваність при розвитку.

- Монорепна архітектура [10], як і з імені зрозуміло, передбачає розміщення всього проекту в одному репозиторії. Це означає, що весь код,

стилі, конфігурації та інші ресурси проекту зберігаються в одному місці. Така архітектура дозволяє швидко відшукати потрібний файл, має більш просту структуру проекту і зменшує кількість залежностей між різними модулями проекту.

Переваги:

1. Простота: Монорепна архітектура проста в розумінні, вона дозволяє розробникам швидко зорієнтуватися у структурі проекту.
2. Менші залежності: Оскільки всі частини проекту зберігаються в одному репозиторії, зменшується кількість залежностей між різними модулями проекту.
3. Простота управління: Одним з головних переваг монорепної архітектури є простота управління ресурсами проекту.
4. Швидкий розвиток: Монорепна архітектура дозволяє розробникам швидко розвивати проект.

Недоліки:

1. Збільшення обсягу коду: Оскільки весь проект зберігається в одному репозиторії, збільшується обсяг коду, що робить проект важчим для обслуговування та відладки.
2. Стійкість: Якщо монореп зламається, всі модулі, які залежать від нього, також зламаються.
3. Конфлікти: Конфлікти в git можуть бути проблемою, якщо багато розробників працює над одним проектом.

Уцілому, монорепна архітектура може бути ефективним вибором для проектів з багатьма додатками або проектів, що розробляються декількома розробниками. Однак, для менших проектів або проектів з невеликою кількістю функцій ця архітектура може бути надто складною і неоптимальною.

### 1.4. State management й здійснення запитів у мобільних додатках

State management та виконання запитів – це два різних поняття в React Native, але вони часто використовуються разом, оскільки одне з них зазвичай залежить від іншого.

Ось деякі з найпопулярніших видів state management у React Native:

1. Redux [11] – це бібліотека для керування станом додатка, що дозволяє зберігати та оновлювати стан додатка в одному місці. Redux дозволяє зберігати стан у вигляді дерева, яке складається зі зв'язаних між собою редюсерів. Використання Redux дозволяє зробити стан додатка більш прогнозованим та зручним для керування.

2. MobX [12] – це ще одна бібліотека для керування станом додатка, яка дозволяє автоматично оновлювати стан додатка, коли відбуваються зміни. За допомогою MobX можна визначати "спостерігачів" за деякими змінними та автоматично оновлювати відповідні компоненти.

3. Context API [13] – це механізм для передачі даних від одного компонента до іншого, не використовуючи проміжні компоненти. Context API дозволяє зберігати стан додатка в глобальному контексті та передавати його у вкладені компоненти. Використання Context API дозволяє зменшити кількість проміжних компонентів у додатку.

4. Zustand [14] – це бібліотека для керування станом додатку в React Native, яка дозволяє легко і швидко організовувати стан вашого додатку. Вона пропонує зручний і простий API, який дозволяє легко створювати, читати і змінювати стан вашого додатку.

Щодо здійснення запитів, є кілька підходів для керування станом запитів в додатку. Один з найбільш популярних підходів – це використання бібліотеки axios [15] або fetch для виконання запитів на сервер, і збереження стану запиту в сторонній бібліотеці для керування станом, наприклад, Redux. В цьому випадку, запити відправляються з компонентів, і стан запитів зберігається в глобальному стані додатку. Коли відповідь приходить з сервера, вона також зберігається в глобальному стані, і компоненти, які підписалися на цей стан, автоматично оновлюються.

Іншим підходом є використання локального стану компонентів для збереження стану запитів. У цьому випадку, компоненти відправляють запити на сервер і зберігають стан запиту локально, використовуючи хуки, такі як useState або useReducer. Цей підхід зазвичай застосовується в додатках, де запити не потрібно ділити з іншими компонентами.

Інший варіант – це використання спеціалізованих бібліотек для керування станом запитів, таких як react-query [16]. Ця бібліотека дозволяє керувати станом запитів та кешуванням даних, забезпечуючи зручний API для виконання запитів з компонентів та оновлення стану компонентів при отриманні відповіді від сервера.

Основна ідея React Query полягає в тому, що вона дозволяє виконувати запити на сервер лише в тому випадку, якщо дані дійсно змінилися або їх ще немає в кеші. Це зменшує кількість зайвих запитів до сервера, зменшує трафік і покращує швидкість відгуку додатка.

React Query також має можливість автоматичного оновлення даних з сервера з певною періодичністю, щоб завжди мати свіжі дані. Крім того, вона підтримує попереднє завантаження даних, коли користувач переходить з однієї сторінки на іншу, щоб запобігти затримкам в завантаженні даних.

Окрім управління станом даних, React Query має також можливість кешування запитів, що дозволяє ефективно працювати з локальними даними. Бібліотека також дозволяє просто виконувати запроси з використанням HTTPS-методів, таких як GET, POST, PUT, DELETE, PATCH.

React Query має кілька переваг порівняно з іншими бібліотеками стану та кешування даних. Основні переваги включають:

1. Простота використання та налаштування.
2. Підтримка вбудованих оптимізацій для кешування даних та управління станом.
3. Підтримка автоматичного оновлення даних з сервера.
4. Широкий спектр налаштувань та функцій для кастомізації.

Основним недоліком React Query є його складність, особливо для початківців. Оскільки бібліотека досить молода, документація може бути неповною або неясною у деяких місцях. Крім того, під час розробки складних додатків може виникнути проблема з ефективним кешуванням та підтримкою оптимістичного оновлення даних. Також, використання React Query може призвести до певного розмиття меж між бекендом та фронтендом, оскільки бібліотека включає в себе деякі функції, що зазвичай пов'язані з бекендом, такі як кешування даних із зовнішніх джерел.

Крім того, React Query не підтримує синхронізацію даних між вкладеними компонентами, що може призвести до труднощів під час розробки складних додатків з багатьма взаємозалежними компонентами.

Загалом, react-query є потужним інструментом для керування станом додатків та виконання запитів до сервера, забезпечуючи зручний інтерфейс, ефективне кешування даних та підтримку важливих функцій, таких як оптимістичні оновлення та мутації.

## 1.5. Постановка задачі

За допомогою вищезазначених інструментів потрібно створити мобільний додаток «Weather App», що буде допомагати людям дізнатися про погоду в своєму чи іншому місті. Список завдань, що потрібно буде виконати, задля успішної реалізації проекту:

1. Вибір технологій та скоупа для розробки
2. Відшукати API для отримання даних погоди за параметрами
3. Створення дизайну застосунка
4. Створення архітектури застосунка
5. Створення клієнтської сторони за дизайном
6. Додавання логіки з API, state management

У разі успішного виконання, можна буде провести відкрите тестування й після всіх репортів відкрити додаток для всіх.

# 2. МЕТОДИКА ВИРІШЕННЯ ЗАДАЧ

## 2.1. Фреймворк React Native

React Native поєднує найкращі частини нативної розробки з React, найкращою у своєму класі бібліотекою JavaScript для створення інтерфейсів користувача.

Примітиви React відображаються у власному інтерфейсі користувача платформи, тобто ваша програма використовує ті самі API власної платформи, що й інші програми.

Компоненти React обгортають існуючий нативний код і взаємодіють з нативними API через декларативну парадигму інтерфейсу React і JavaScript. Це дає змогу створювати нативні додатки цілим новим командам розробників, а наявні нативні команди можуть працювати набагато швидше.

Подібно до React для Інтернету, програми React Native написані з використанням суміші розмітки JavaScript і XML, відомої як JSX. Потім під капотом «bridge» React Native викликає рідні API рендерингу в Objective-C (для iOS) або Java (для Android). Таким чином, ваша програма відтворюватиметься за допомогою компонентів реального мобільного інтерфейсу користувача, а не веб-переглядів, і виглядатиме та працюватиме як будь-яка інша мобільна програма. React Native також надає інтерфейси JavaScript для API платформи, тому ваші програми React Native можуть отримувати доступ до таких функцій платформи, як камера телефону або місцезнаходження користувача.

Наразі React Native підтримує як iOS, так і Android і має потенціал для розширення на майбутніх платформах.

Для розробників, які звикли працювати в Інтернеті з React, це означає, що ви можете писати мобільні програми з продуктивністю та зовнішнім виглядом нативної програми, використовуючи знайомі інструменти. React

Native також покращує звичайну мобільну розробку у двох інших сферах: досвід розробника та потенціал кросплатформної розробки.

## 2.2. Feature-sliced архітектура мобільного додатка

Feature-Sliced Design (FSD) – це архітектурна методологія для проектування frontend-додатків. Простіше кажучи, це зведення правил та угод щодо організації коду. Головна мета методології — зробити проект зрозумілим та структурованим, особливо за умов регулярної зміни вимог бізнесу.

У методології FSD використовуються три рівні абстракції:

1. Шари – рівень, що визначає швидкість відповідальності шару, а також рівень небезпеки змін. Чим вище розташований шар, тим вище рівень його відповідальності та знань про інші шари. Чим нижче розташований шар, тим він абстрактніший і більше використовується у верхніх шарах, а значить більше небезпеки вносити до нього зміни.

2. Слайси – відображають конкретну функціональність бізнес-логіки. Методологія майже впливає цей рівень, багато залежить від конкретного проекту. За фактом це папки, які групують файли, що реалізують той чи інший модуль.

3. Сегменти – розподіляються за призначенням модуля в коді та реалізації. За методологією кожен сегмент відповідає за свою частину технічної реалізації модуля:

- api/ – робота з API. Автори методології радять класти API-логіку в shared, щоб вона не розпорошувалася за проектом.
- config/ – конфігурація модуля.
- lib/ – різні утилітарні функції та допоміжні бібліотеки.
- model/ – бізнес-логіка: store, actions, effects, reducers тощо.
- ui/ – відповідає за відображення.

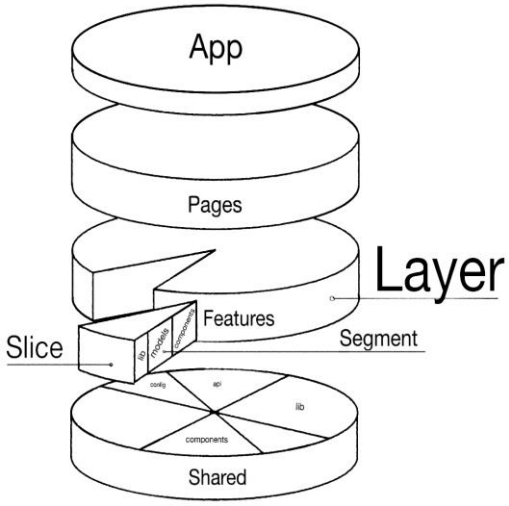


Рисунок 2.1 – Графічне представлення FSD

Розглянемо кожен шар докладно – від найнижчого до самого верхнього.

1. Shared – найабстрактніший шар програми, який містить модулі, що перевикористовуються, не пов'язані з бізнес-логікою. Цей шар добре підходить для початку застосування FSD.

Найголовніше – не плутати абстрактні UI-компоненти з компонентами, що реалізують конкретні бізнес-сутності чи фічі. Наприклад, Select – це Shared/UI-компонент, а CityWeatherSelect – вже фіча. При створенні того чи іншого модуля потрібно оцінити, чи використовуватимуться бізнес-сутності у компоненті. Якщо ні – модуль необхідно назвати максимально абстрактно та помістити в Shared. В іншому випадку, бажано вказати в назві бізнес-сутність або фічу, яку модуль реалізує, та помістити в шар Entities або Features. Від грамотного неймінгу у цій методології залежить дуже багато.

2. Entities – це компоненти, пов'язані з поданням бізнес-сутностей, «цеглинки», за допомогою яких відбувається побудова бізнес-логіки. Цей шар варто впроваджувати бажано разом із шаром Features, про який розповім нижче.

3. Features – частини функціональності програми. Мабуть, найскладніший для розуміння та визначення шар у методології, оскільки саме визначення «фічі» залежить від конкретної прикладної галузі та бізнес-вимог.

Тому при переході на FSD потрібно впроваджувати «фічі» тільки за умови повної впевненості, що це не внесе додаткову складність для розробників.

4. Widgets – самостійні та повноцінні блоки сторінок з конкретними діями. Шар також добре підходить для початку застосування FSD: у різних сторінках однієї програми багато частин часто повторюються. Щоб не дублювати їхню реалізацію, частини можна виносити у віджети. Але було вирішено не використовувати цей шар у проекті.

5. З шаром Pages все просто і зрозуміло – це сторінки нашої програми. Методологія радить, щоб кожна сторінка мала максимально просту структуру, а бізнес-логіку переносити на нижчі верстви. Тому сторінка – це композиція з віджетів та/або фіч, яка відображає взаємодію між шарами, що нижчі.

6. У папці Processes знаходиться логіка, яка стосується кількох сторінок або всі програми. У проекті було вирішено не використовувати цей шар.

7. У App знаходиться загальна логіка програми, що ініціалізує, – різні обгортки, глобальні стори і стилі. Загалом все те, що впливає на роботу всього додатка.

Було створено архітектуру додатку за використанням FSD (рис. 2.2):

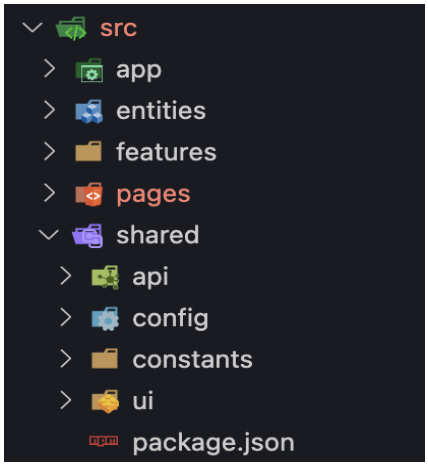


Рисунок 2.2 – Архітектура додатка за використанням FSD

Основна складність FSD полягає у вищому порозі входу в порівнянні з іншими поширеними підходами. Розробнику доводиться вчитися розуміти потреби користувача та мети бізнесу.

### 2.3. Zustand як state management й react-query для виконання запитів

Нами було вирішено використовувати Zustand як state management для збереження даних у додатку.

Zustand – це бібліотека стану для React Native, яка дозволяє зберігати стан додатку в окремому контейнері стану (store) і забезпечує легку інтеграцію з React Native компонентами. Zustand підтримує використання глобального стану, який може бути локалізованим для конкретних компонентів та легко тестується.

Основними поняттями в Zustand є сторонній контейнер стану (store) і функції-редюсери, які забезпечують зміну стану. Щоб створити контейнер стану, необхідно використовувати функцію `createStore`, яка приймає в себе початковий стан та обробник стану. Функції-редюсери дозволяють змінювати стан контейнера і повертають новий стан.

Однією з головних переваг Zustand є його простота. Він дозволяє швидко розробляти додатки та використовувати глобальний стан, щоб забезпечити спільний стан між компонентами. Також, Zustand має досить невеликий розмір та невимогливий до налаштування.

Іншою перевагою Zustand є його хороша підтримка асинхронних запитів та сайд-ефектів. Для зберігання стану, Zustand використовує контейнер, який підтримує асинхронність та здатний забезпечити плавне оновлення стану. Завдяки цьому, з Zustand можна легко працювати з асинхронними операціями, такими як запити до сервера, підписки на події тощо.

Загалом, zustand є простим та потужним інструментом для керування станом додатків, що дозволяє зберігати код компактним та простим для розуміння, що сприяє розвитку та підтримці додатків у майбутньому.

Представлено приклад store з додатку (рис. 2.3):

```
export const useWeatherStore = create(
  devtools(
    persist<WeatherStore>(
      (set, get) => ({
        currentCityWeatherData: null,
        actions: {
          setCurrentCityData: currentCityWeatherData =>
            set({
              currentCityWeatherData,
            }),
          resetCurrentCity: () => set({ currentCityWeatherData: null }),
        },
      }),
      {
        name: 'weather-store',
        storage: createJSONStorage(() => AsyncStorage),
      },
    ),
  ),
);
```

Рисунок 2.3 – Weather store за використанням Zustand

Для використання запитів було використано axios й react-query.

React Query – це бібліотека управління станом та кешування для React Native, яка спрощує отримання, зберігання та оновлення даних, особливо тих, що отримуються з сервера. Вона надає потужні засоби для виконання запитів, кешування результатів, вирішення проблем зі станом навантаження та оновлення даних.

Основна ідея за React Query полягає у використанні концепції "запитів" (queries) та "мутацій" (mutations). Запити використовуються для отримання даних з сервера, в той час як мутації використовуються для оновлення даних на сервері. Бібліотека автоматично керує кешуванням та оновленням даних, що дозволяє ефективно управляти станом додатку.

React Query надає безліч додаткових функцій, таких як обробка помилок, автоматичне повторне виконання запитів, підтримка пагінації та багато іншого, що робить роботу з даними більш ефективною та зручною.

Представлено приклад використання react-query з додатку (рис. 2.4):

```
export const useGetCityWeatherByLocation = (
  position: TPosition,
  options?: UseQueryOptions<
    AxiosPromise<WeatherData>,
    AxiosError,
    WeatherData,
    readonly (string | number)[]
  >,
) => {
  return useQuery({
    queryFn: () => getCityWeather({ position }),
    queryKey: [...cityKeyFactory.citiesByLocation(position)],
    ...options,
  });
};
```

Рисунок 2.4 – Запит на отримання погоди з використанням react-query

У цілому, React Query є потужною та гнучкою бібліотекою для роботи з даними в React-додатках, яка спрощує керування станом та підвищує продуктивність програми.

## 2.4. Огляд інструментів для розробки

Для розробки цього додатку знадобляться такі інструменти:

- Visual Studio Code – редактор коду
- XCode – для запуску додатка на IOS
- Android Studio – для запуску додатка на Android

Було обрано саме ці інструменти, через те що протягом 4-х років навчання й праці було засвоєно навички React й React Native.

# 3. ПРОГРАМНА РЕАЛІЗАЦІЯ

Програмна реалізація додатку виглядає наступним чином:

## 3.1. Створення структури проекту

Використовуючи підхід Feature-Sliced Design, було розділено функціональність додатка на окремі модулі або "фічі". Кожен модуль включає компоненти, стилі, логіку та стан, пов'язаний з цим модулем.

Ниже представлено структуру додатку:

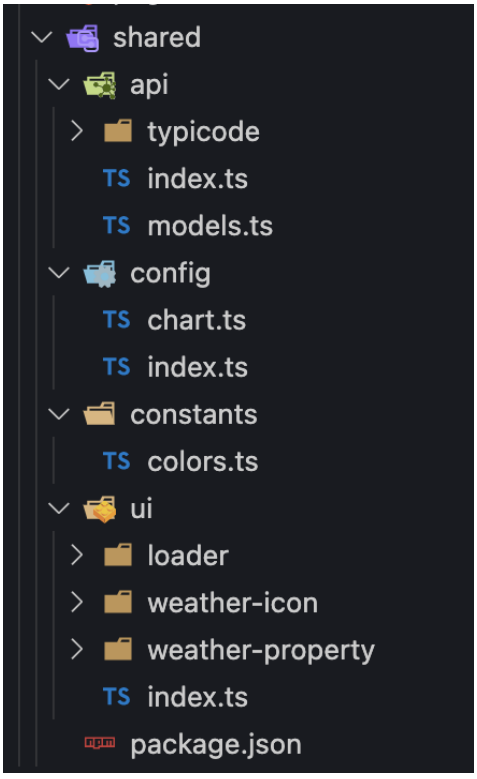


Рисунок 3.1 – Структура шару shared

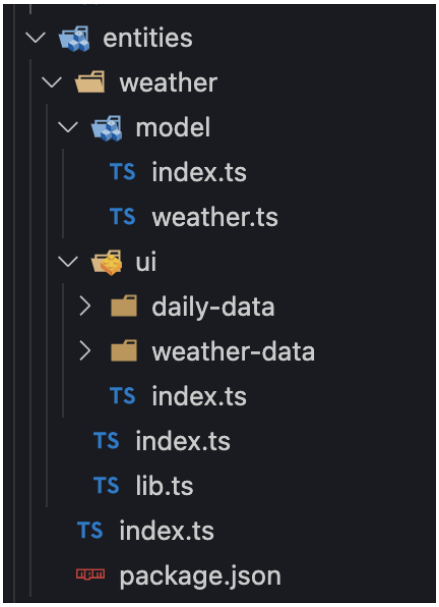


Рисунок 3.2 – Структура шару entities

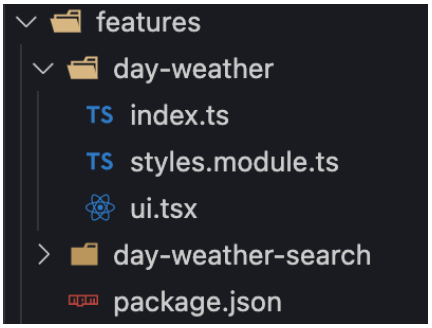


Рисунок 3.3 – Структура шару features

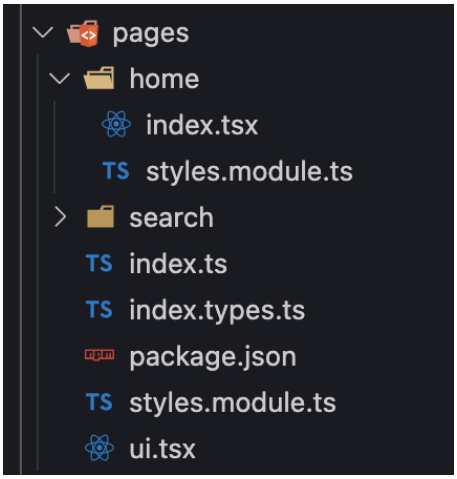


Рисунок 3.4 – Структура шару pages

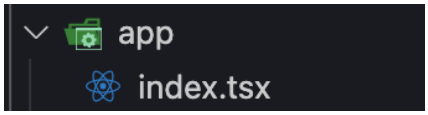


Рисунок 3.5 – Структура шару app

## 3.2. Управління станом з Zustand

### 3.2.1. Налаштування Zustand store

На рисунку 3.6 зображено ініціалізацію store, actions й selector.

```
export const useWeatherStore = create(
  devtools(
    persist<WeatherStore>(
      (set, get) => ({
        currentCityWeatherData: null,
        actions: {
          setCurrentCityData: currentCityWeatherData =>
            set({
              currentCityWeatherData,
            }),
          resetCurrentCity: () => set({ currentCityWeatherData: null }),
        },
      }),
      {
        name: 'weather-store',
        storage: createJSONStorage(() => AsyncStorage),
      },
    ),
  ),
);

export const useWeatherActions = () => useWeatherStore(state => state.actions);

export const useCurrentCityWeather = () =>
  useWeatherStore(state => state.currentCityWeatherData, shallow);
```

Рисунок 3.6 – Zustand store з actions й selector

У цьому сторі зберігаються дані про погоду у місті за геолокацією й є 2 actions для зміни й видалення даних. Також при ініціалізації store було використано 2 middlewares. Перша – це devtools. Вона дозволяє зручно дебажити додаток при розробці. А друга – це persist. Ця middleware використовується для збереження store у AsyncStorage й дозволяє зберігати дані після перезаходу до додатку.

### 3.2.2. Використання Zustand actions й selector

На рисунку 3.7 можна побачити, що визиваємо selector currentCityWeather й action setCurrentCityData. Й далі по коду, якщо після запросу не буде data, то буде використовуватись значення currentCityWeather, а якщо є – запишеться data у store за допомоги action setCurrentCityData.

```
const { data, isLoading, isError } = useGetCityWeatherByLocation(position);
const currentCityWeather = useCurrentCityWeather();
const { setCurrentCityData } = useWeatherActions();
```

Рисунок 3.7 – Використання Zustand store з actions й selector

## 3.3. Отримання даних з OpenWeatherAPI за допомогою React Query

На рисунку 3.8 можна побачити, що створено запит useGetCityWeatherByLocation куди передається поточна геопозиція й виконується axios запит на сервер OpenWeatherAPI [17, 18] й кешується за допомоги cityKeyFactory.

```
const cityKeyFactory = {
  cities: ['all-cities'],
  citiesByLocation: (position: TPosition) => [
    ...cityKeyFactory.cities,
    `${position.latitude}:${position.longitude}`,
  ],
  citiesByName: (name: string) => [...cityKeyFactory.cities, name],
} as const;

export const useGetCityWeatherByLocation = (
  position: TPosition,
  options?: UseQueryOptions<
    AxiosPromise<WeatherData>,
    AxiosError,
    WeatherData,
    readonly (string | number)[]
  >,
) => {
  return useQuery({
    queryFn: () => getCityWeather({ position }),
    queryKey: [...cityKeyFactory.citiesByLocation(position)],
    ...options,
  });
};
```

Рисунок 3.8 – Ініціалізація запиту з React Query

### 3.4. Огляд додатку й тестування

Цей додаток – зручний інструмент для отримання актуальної інформації про погоду у будь-якому місці світу. Додаток пропонує користувачеві детальну прогнозу погоди, включаючи температуру, вологість, швидкість вітру та інші важливі показники. Завдяки інтуїтивно зрозумілому інтерфейсу, користувачі зможуть легко переглядати та порівнювати прогнози для різних місць.

1. Головна сторінка:

Після запуску додатка на головній сторінці виконується запит для отримання даних про погоду з поточної геопозиції. Після отримання даних, відображається загальна інформація про погоду, така як температура, погодні умови та час сходу/заходу сонця.


Рисунок 3.9 – Головна сторінка додатку

2. Детальний прогноз погоди:

У розділі детального прогнозу користувач отримує більш повну інформацію про погоду на декілька наступних днів. Тут відображаються графік температури, що дозволяє користувачу зрозуміти тенденції погоди протягом тижня.


Рисунок 3.10 – Графік температури на тиждень уперед

3. Пошук погоди за містом:

В Weather App є можливість шукати погоду для конкретного міста. Користувач може ввести назву міста, щоб знайти погоду у бажаному регіоні. Після виконання пошуку, відображається детальна інформація про погоду для вибраного міста.


Рисунок 3.11 – Пошук погоди за назвою міста

4. Віджет поточної погоди:

Віджет погоди є важливою частиною додатку, оскільки надає користувачам швидкий та зручний доступ до загальної інформації про погоду. Віджет погоди відображає основні показники погоди, такі як температура, дату та статус погоди, безпосередньо на головній сторінці вашого телефону. Користувачам не потрібно виконувати додаткові кроки або переходити на інші сторінки, щоб дізнатися поточну погоду. Також віджет погоди оновлюється автоматично, що дозволяє користувачам отримувати актуальну інформацію про погоду у режимі реального часу. Користувачі можуть швидко переглянути погоду без затримок та зусиль.

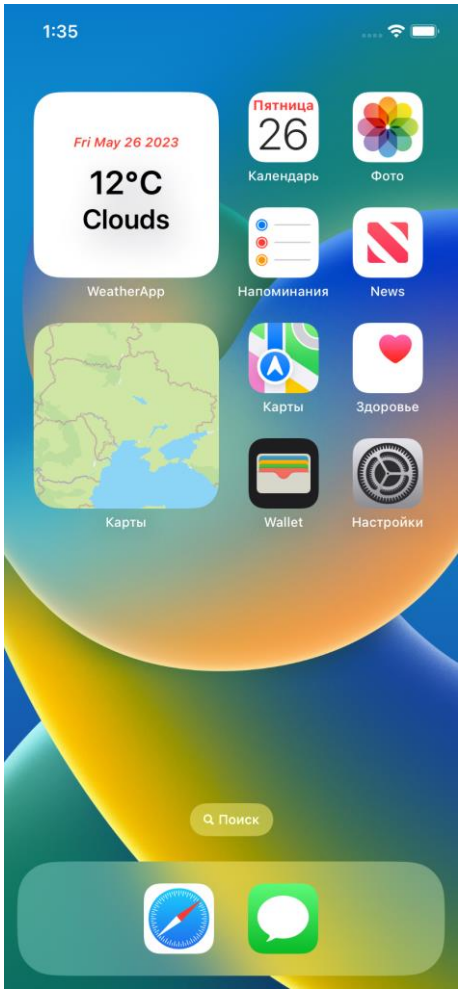


Рисунок 3.12 – Віджет додатку

Weather App є надійним джерелом інформації про погоду, а його зручний та інтуїтивно зрозумілий інтерфейс робить його доступним для будь-

якого користувача. Він забезпечує точність та актуальність погодних даних, дозволяючи користувачам завжди бути в курсі погодних умов у будь-якому місці, коли це потрібно.

Тестування додатка Weather App включає різноманітні аспекти, щоб переконатися, що він функціонує належним чином та надає точну та актуальну інформацію про погоду. Ось кілька складових тестування, які були виконані:

1. Функціональні тести: Перевірка основних функцій додатка, таких як вибір геопозиції, отримання загального прогнозу погоди, пошук погоди за містом та перегляд детального прогнозу. Впевненість у тому, що всі основні функції працюють належним чином, є важливим аспектом тестування.

2. Тести візуального інтерфейсу: Переконалися, що дизайн додатка відповідає вимогам та надає приємний користувацький досвід. Перевірка розміщення елементів, зручності навігації, зчитування інформації та загального візуального враження.

3. Тести взаємодії з користувачем: Перевірка реакції додатка на взаємодію з користувачем, таку як введення тексту, вибір пунктів меню, прокручування списків та інші дії. Переконалися, що додаток правильно реагує на дії користувача і надає зручний та швидкий досвід використання.

4. Тести на забезпечення якості даних: Перевірка точності та актуальності погодних даних, які надаються додатком. Перевірка, чи відповідають отримані дані даним з надійних джерел та чи оновлюються вони належним чином.

5. Тести на помилки та стабільність: Тестування додатка на виявлення можливих помилок, таких як збої, некоректна поведінка або відсутність відповіді. Перевірка стабільності додатка під час навантаження та взаємодії з різними пристроями та операційними системами.

6. Тести на різні сценарії використання: Виконання тестів, що моделюють різні сценарії використання додатка, наприклад, перший запуск, перехід між різними екранами, використання функцій пошуку та фільтрації погоди та інші.

Ці різноманітні тести допомогли переконатися, що додаток Weather App працює належним чином, надає коректну та актуальну інформацію про погоду та забезпечує зручний та задовільний користувацький досвід.

Додаток для встановлення на ОС Android можна завантажити за посиланням:

http://surl.li/hyyod

Додаток для встановлення на IOS можна завантажити за посиланням: https://apps.apple.com/us/app/weather-app-get-your-weather/id6449674316

Повний код розробленого додатку знаходиться за посиланням:

https://github.com/vdzeer/weather_app

# ВИСНОВКИ

У результаті виконання кваліфікаційної роботи було розглянуто програмну реалізацію Weather App з використанням Feature-Sliced Design, Zustand, React Query та OpenWeatherAPI. Кожна з цих технологій внесла свій внесок у створення потужного та ефективного додатка.

Feature-Sliced Design дозволив розділити функціональність додатка на окремі модулі, що спрощує розробку та підтримку. Zustand надав простий та зрозумілий спосіб управління станом додатка, забезпечуючи легку передачу даних між компонентами. React Query дозволив ефективно виконувати запити до сервера, кешувати результати та автоматично оновлювати дані. OpenWeatherAPI забезпечив доступ до актуальних погодних даних для різних місць.

Застосування цих технологій у реалізації Weather App дозволило створити зручний та функціональний додаток, який надає користувачам актуальну інформацію про погоду. Крім того, здійснення подібної реалізації дозволило створити масштабовану та легкозмінну архітектуру, що полегшує розширення функціональності та збереження коду додатка.

Протягом роботи було наведено приклади використання кожної технології у контексті Weather App. Показано, як за допомогою Feature-Sliced Design можна організувати структуру додатка, розділивши його на модулі для кращої підтримки та розширюваності. Застосування Zustand дозволило легко управляти станом додатка, забезпечуючи одночасну реактивність та швидкий доступ до даних. React Query додав можливість виконувати запити до сервера, кешувати результати та автоматично оновлювати дані. Використання OpenWeatherAPI надавало можливість отримувати актуальні погодні дані для відображення користувачам.

У загальному, поєднання Feature-Sliced Design, Zustand, React Query та OpenWeatherAPI створило потужну та ефективну реалізацію Weather App. Вона надає зручний спосіб отримання та відображення погодних даних,

керування налаштуваннями та надає зручний інтерфейс користувачу. Застосування цих технологій може бути корисним для подібних проектів, де потрібно ефективно керувати станом та отримувати дані з віддалених джерел.

## СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ


1. Про мобільну розробку: веб-сайт. URL: https://www.freelancermap.com/blog/what-does-mobile-developer-do/ (дата звернення: 07.01.2023)

2. React Native: веб-сайт. URL: https://reactnative.dev/ (дата звернення: 14.04.2023)

3. React Native Architecture: веб-сайт. URL: https://reactnative.dev/docs/architecture (дата звернення: 14.04.2023)

4. Про React Native: веб-сайт. URL: https://www.oreilly.com/library/view/learning-react-native/9781491929049/ch01.html (дата звернення: 07.01.2023)

5. JavaScript: веб-сайт. URL: https://www.hostinger.com/tutorials/what-is-javascript (дата звернення: 07.01.2023)

6. TypeScript Docs: веб-сайт. URL: https://www.typescriptlang.org/ (дата звернення: 14.04.2023)

7. Класична архітектура React Native: веб-сайт. URL: https://dev.to/rubemfsv/clean-architecture-applying-with-react-40h6 (дата звернення: 14.04.2023)

8. Atomic архітектура React Native: веб-сайт. URL: https://medium.com/@wheeler.katia/thinking-about-react-atomically-608c865d2262 (дата звернення: 14.04.2023)

9. Feature-Sliced Design архітектура React Native: веб-сайт. URL: https://feature-sliced.design/ (дата звернення: 14.04.2023)

10. Монорепна архітектура React Native: веб-сайт. URL: https://www.toptal.com/front-end/guide-to-monorepos (дата звернення: 14.04.2023)

**11. Redux Docs: веб-сайт. URL: https://redux.js.org/ (дата звернення: 14.04.2023)**

**12. MobX Docs: веб-сайт. URL: https://mobx.js.org/ (дата звернення: 14.04.2023)**

**13. Context API Docs: веб-сайт. URL: https://kentcdodds.com/blog/how-to-use-react-context-effectively (дата звернення: 14.04.2023)**

**14. Zustand Docs: веб-сайт. URL: https://docs.pmnd.rs/zustand/getting-started/introduction (дата звернення: 14.04.2023)**

**15. Axios Docs: веб-сайт. URL: https://axios-http.com/ (дата звернення: 14.04.2023)**

**16. React Query Docs: веб-сайт. URL: https://tanstack.com/query/v3/docs/react/overview (дата звернення: 14.04.2023)**

**17. Weather API Docs: веб-сайт. URL: https://openweathermap.org/api (дата звернення: 14.04.2023)**

**18. Weather API приклади використання: веб-сайт. URL: https://github.com/topics/openweathermap-api (дата звернення: 14.04.2023)**