НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ

Завідувач кафедри
комп'ютерних наук

_____/Голуб Б.Л., доц.,к.т.н. /
підпис ПІБ, вчене звання і ступінь

«___»_____2025 р

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«Програмне забезпечення системи аналізу погоди на маршруті»

Спеціальність 121 «Інженерія програмного забезпечення»

Гарант освітньої програми
к.т.н., доцент _____ / Вайганг Г.О./
Науковий ступень та вчене звання підпис ПІБ

Керівник бакалаврської кваліфікаційної роботи : _____ / Руденський Р.А./
підпис ПІБ

Виконав: _____ / Радчук С.І./
підпис ПІБ

КИЇВ-2025

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ
І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ЗАТВЕРДЖУЮ
Завідувач кафедри
комп'ютерних наук

_____/ Голуб Б.Л., доцент, к.т.н /
підпис
"____"_____2025 р.

ЗАВДАННЯ

на виконання бакалаврської кваліфікаційної роботи студенту
Радчуку Сергію Ігоровичу
Спеціальність 121 – «Інженерія програмного забезпечення»
Тема бакалаврської кваліфікаційної роботи «Програмне забезпечення системи аналізу погоди на маршруті»
затверджена наказом ректора НУБіП України від 16.12.2024 р. № 2248 "С" ___
Термін подання завершеної роботи на кафедру _____
(рік, місяць, число)

Вихідні дані до бакалаврської кваліфікаційної роботи

Перелік питань, які потрібно розробити:
Аналіз предметної області та постановка задачі, моделювання системи за допомогою UML-діаграм, розробка інформаційного забезпечення та структури даних, реалізація програмної системи для мобільного та серверного середовища, тестування, розгортання та впровадження системи.
Перелік графічних документів (за потреби)

Дата видачі завдання "_____"_____20___р.
Керівник бакалаврської кваліфікаційної роботи
професор _____ Руденський Р.А.
(науковий ступінь та вчене звання) (підпис) (ПІБ)

Завдання прийняв до виконання _____ Радчук С.І.
(підпис) (ПІБ студента)

ЗМІСТ

ВСТУП

У сучасному світі подорожі автомобілем стали невід'ємною частиною повсякденного життя. Щодня мільйони людей вирушають у дорогу з особистих чи професійних причин. Однак не завжди подорожі бувають комфортними, оскільки погодні умови є одним з основних факторів, що впливають на якість та безпеку поїздки. Погода може змінюватися непередбачувано: раптовий дощ, туман, снігопад або сильний вітер можуть не лише ускладнити рух, але й становити потенційну загрозу для життя водіїв та пасажирів.

Ідея розробки програмного забезпечення для аналізу погодних умов на маршруті перед поїздкою прийшла до мене після особистого досвіду. Одного літнього дня я їхав у маршрутці і відчував себе дуже некомфортно через спеку. Приблизно на півдорозі несподівано пішов дощ. Температура впала, повітря стало прохолоднішим, і їхати стало набагато комфортніше. Ця проста зміна погоди суттєво вплинула на моє відчуття комфорту, і я запитав себе: а що, якби я міг передбачити такі зміни перед поїздкою і вибрати найкращий час або маршрут?

Повернувшись, я проаналізував наявні прогнози погоди для цього маршруту. Виявилося, що більшість сервісів обмежуються загальними прогнозами для міст або пунктів на маршруті, не враховуючи час прибуття, швидкість, зупинки або зміни погоди в реальному часі. Багато рішень є занадто узагальненими або, навпаки, складними у використанні для пересічного користувача. Це підтвердило необхідність розробки індивідуального програмного забезпечення, яке фокусується на простоті, точності, адаптивності та легкості використання.

Метою даної роботи є розробка програмно-апаратного комплексу, який дозволяє користувачеві ввести маршрут, визначити параметри подорожі (швидкість, дату відправлення, зупинки тощо) та отримати детальний аналіз

погоди за маршрутом у вигляхді практичної карти та таблиці. Особливістю системи є те, що вона враховує не тільки координати, а й час у дорозі для кожної точки маршруту, щоб ці точки можна було порівняти з поточними даними про погоду. Таким чином, можна уникнути несподіваних погодних умов на маршруті, підвищуючи безпеку руху та забезпечуючи комфорт користувача.

Розроблене рішення складається з Android-додатку, що відповідає за користувацький інтерфейс, прокладання маршруту, перевірку введених даних та відображення результатів, та серверної частини, реалізованої на Spring Boot. Серверна частина обробляє запити, отримує погодні дані з API, розраховує погодні умови на маршруті та надсилає відповідь назад у мобільний додаток. Система також дозволяє зберігати історію маршруту для подальшого аналізу.

Запропоноване програмне забезпечення має на меті не лише полегшити планування поїздок, але й підвищити безпеку та зробити подорожі більш передбачуваними та комфортними. Впровадження такого рішення важливе як для приватних водіїв, так і для служб таксі, кур'єрських служб, експедиторів та інших організацій, що займаються регулярними перевезеннями.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Постановка задачі

Планування маршруту є важливим елементом безпечної та ефективної подорожі. Особливу увагу слід приділяти погодним умовам, які можуть мати значний вплив на швидкість руху, безпеку, комфорт пасажирів і рішення про початок або зміну маршруту. Враховуючи важливість проблеми, необхідно розробити програмне забезпечення, яке може спочатку проаналізувати погодні умови всього маршруту, враховуючи час у дорозі на кожній ділянці маршруту.

Основною метою даної роботи є розробка системи, яка дозволяє користувачеві побудувати маршрут від пункту відправлення до пункту призначення (з можливістю додавання зупинок), визначити параметри маршруту (швидкість, дату відправлення, тривалість зупинок) та отримати аналіз погодних умов на маршруті, а також прогнози, у вигляді позначених карт та таблиць, що містять метеорологічні показники.

Для досягнення цієї мети необхідно вирішити наступні основні завдання:

1. **Аналіз предметної області**, вивчення існуючих сервісів аналізу погоди, визначення переваг і недоліків існуючих рішень.
2. **Проектування загальної архітектури системи** - вибір моделі взаємодії між клієнтом і сервером.
3. **Розробка клієнтської частини:**
    - Створення інтерфейсу Android додатку для введення параметрів маршруту.
    - Реалізація побудови маршруту на карті за допомогою сторонніх API.
    - Створення механізму перевірки даних користувача та передачі запиту на сервер.
4. Розробка серверної частини:
    - Отримувати запит від клієнта через REST.

- Отримання прогнозів погоди із зовнішнього API погоди.
- Розрахунок прогнозу погоди в заданих точках маршруту з урахуванням часу прибуття.
- Підготовка відповідей клієнтам з інформацією про погоду та рекомендаціями.

2. Інформаційне забезпечення:
    - Розробка структури бази даних для зберігання історії запитів і результатів аналізу.
    - Реалізація функцій збереження, перегляду та редагування маршрутів.
3. Візуалізація результатів:
    - Відображення погоди у вигляді кольорових маркерів на карті (зелений – сприятлива погода, жовтий – помірний ризик, червоний – ризик).
    - Створення таблиці з параметрами погоди для кожної точки маршруту (температура, вітер, опади, стан неба тощо).
4. Тестування та оцінка системи:
    - Проведення функціонального тестування програми.
    - Аналіз точності та зручності прогнозу.
    - Дати рекомендації щодо подальшого розвитку системи.

1.2 Моделювання предметної області

Моделювання предметної області є фундаментальним етапом розробки будь-якого програмного забезпечення, зокрема системи аналізу погоди на маршруті. Воно дозволяє виділити основні об’єкти, процеси, учасників та їхні взаємозв’язки, що виникають у межах функціонування системи. Ретельне моделювання допомагає краще зрозуміти логіку роботи, оптимізувати архітектуру програмного продукту, уникнути логічних помилок та забезпечити надійність розробленого рішення.

Предметна область цього проєкту охоплює процес планування маршруту з урахуванням погодних умов. Користувач, вводячи початкову і кінцеву точки маршруту, параметри швидкості, дату виїзду та тривалість зупинок, ініціює запит, який система обробляє, обчислює погодні умови в часово-просторовому розрізі та виводить результати у зручному вигляді: на карті та у вигляді табличної аналітики.

Щоб ефективно представити структуру та поведінку системи, у роботі використовується мова моделювання UML (Unified Modeling Language) — уніфікована мова моделювання, яка є загальноприйнятим стандартом в індустрії розробки програмного забезпечення. UML дозволяє формалізувати функціональність системи, її структуру та взаємодію компонентів за допомогою діаграм.

В рамках цього проєкту було використано такі типи UML-діаграм:

- Діаграма прецедентів (Use Case Diagram) — демонструє взаємодію зовнішніх учасників (акторів) із функціональністю системи. Вона допомагає зрозуміти, які дії доступні користувачам і як система реагує на ці дії.
- Діаграма діяльності (Activity Diagram) — показує послідовність дій при обробці запиту користувача, від моменту введення параметрів маршруту до отримання прогнозу погоди.

- Діаграма послідовності (Sequence Diagram) — відображає обмін повідомленнями між компонентами системи у часовій послідовності, включаючи взаємодію між клієнтом, сервером та погодним API.
- Діаграма класів (Class Diagram) — моделює основні сутності предметної області, їх атрибути, методи та зв’язки між ними.
- Діаграма пакетів (Package Diagram) — ілюструє логічну структуру проєкту, розбиття коду на модулі та підсистеми.
- Діаграма компонентів (Component Diagram) — відображає високорівневі компоненти системи та залежності між ними (наприклад, мобільний застосунок, сервер, база даних, сторонні API).
- Діаграма розгортання (Deployment Diagram) — описує фізичну архітектуру розміщення системи: де саме розгорнуто сервер, як він взаємодіє з клієнтами, які програмно-апаратні ресурси задіяні.

Завдяки застосуванню UML, вдалось не лише структурувати технічне бачення системи, а й забезпечити зрозумілу документацію, яка може бути використана як для подальшого розвитку проєкту, так і для командної роботи або передачі системи іншим розробникам.

Моделювання предметної області в контексті цієї системи є ключем до глибокого розуміння її функціонування та є невід'ємною частиною процесу її розробки.

1.3 Діаграма прецедентів

Діаграма прецедентів використання є одним з ключових інструментів для моделювання системи за допомогою мови UML. Вона використовується для формалізації взаємодії між зовнішніми користувачами і функціональністю, що надається системою. Цей тип діаграм використовується для визначення основних сценаріїв використання системи з точки зору кінцевого користувача.

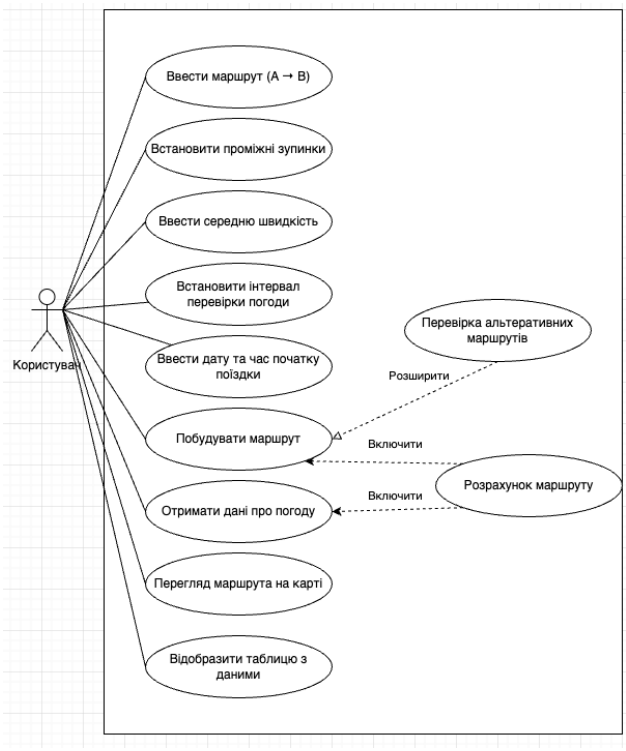

Рисунок 1.1 – Діаграма прецедентів для системи аналізу погоди на маршруті

У контексті програмного забезпечення системи аналізу погоди на маршруті головною дійовою особою є користувач, який взаємодіє з системою для встановлення маршруту та отримання погодних даних. Нижче наведено опис основних прецедентів:

Основні прецеденти:

- **Ввести маршрут (A → B) – Користувач визначає початкову та кінцеву точки маршруту.**

- Визначити проміжні зупинки - На додаток до основного маршруту, користувач може визначити зупинки та тривалість перебування на них.
- Ввести середню швидкість - Значення, яке використовується системою для розрахунку тривалості маршруту.
- Визначити інтервал перевірки погоди - Користувач визначає кількість кілометрів або хвилин, протягом яких погодні умови повинні перевірятися щодня.
- Введіть дату і час початку подорожі - це дозволить системі коректно порівняти погодні дані з тривалістю маршруту.
- Побудувати маршрут - система генерує оптимальний маршрут на основі введених параметрів.
- Відображення таблиці даних - на додаток до карти, інформація про погоду доступна у вигляді структурованої таблиці.

Інтегровані та розширені попередники:

- Розрахунок маршруту - Входить до попереднього Прокласти маршрут. Це технічний підпроцес, який виконується системою для розрахунку маршруту на основі заданих параметрів.
- Перевірка альтернативних маршрутів - є розширенням попереднього етапу Прокладання маршруту і передбачає пошук кращих варіантів на основі погодних або дорожніх критеріїв.

Ця діаграма дозволяє зрозуміти основні функції, реалізовані системою, і наочно показує, як користувач взаємодіє з додатком. Вона також корисна для створення технічних вимог і перевірки повноти реалізованого функціоналу під час тестування.

1.4 Діаграма діяльності

UML-діаграма діяльності - це графічне представлення робочого процесу системи: від дії, ініційованої користувачем, до обробки системою і повернення результату. Вона дозволяє формалізувати логіку послідовності дій як з боку користувача, так і з боку системи.

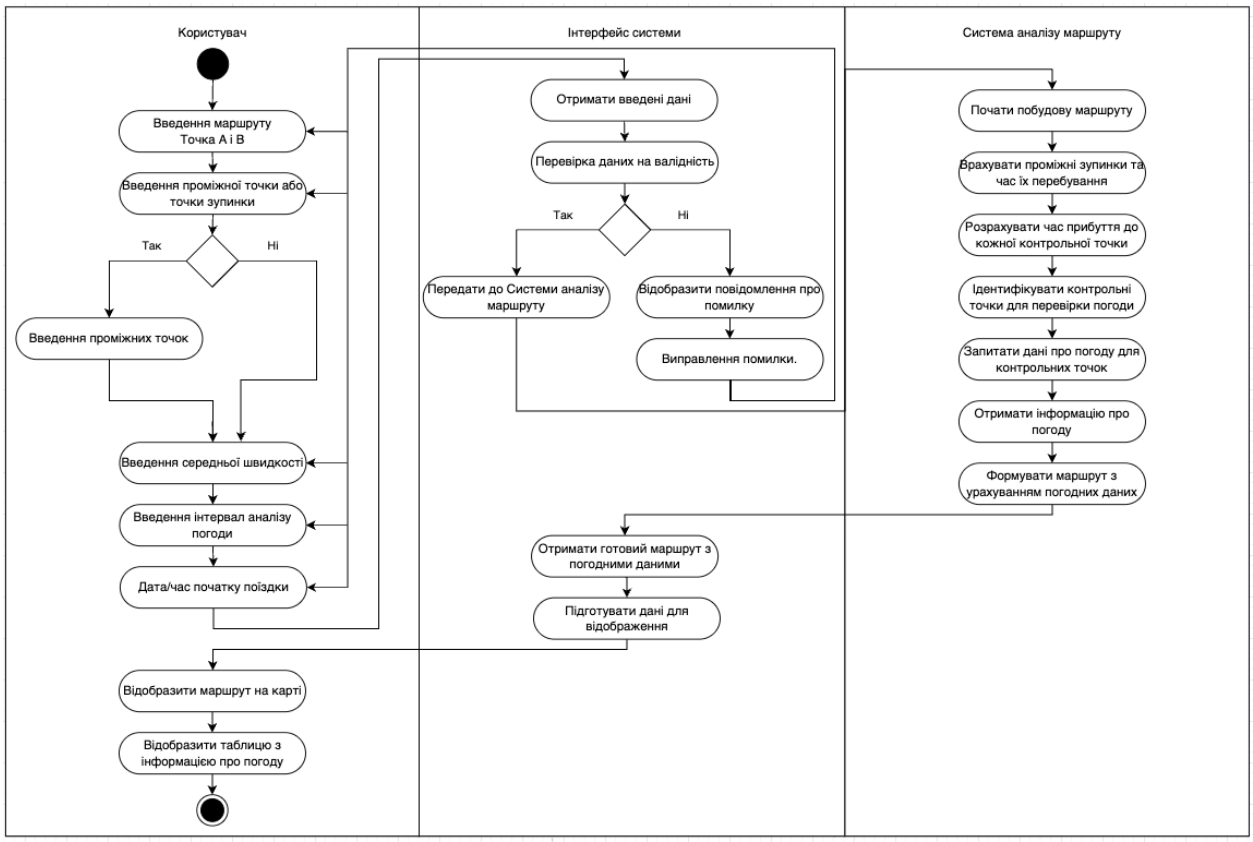

Рисунок 1.2 – Діаграма діяльності системи аналізу погоди на маршруті

У випадку з нашою системою аналізу погоди на маршруті процес починається з того, що користувач вказує основні параметри подорожі: пункти відправлення та прибуття, зупинки, середню швидкість, інтервал перевірки погоди та час відправлення. Потім система перевіряє точність цих даних і, якщо виявляє помилку, користувач може її виправити. Якщо перевірка завершена, система будує маршрут з урахуванням погодних умов і готує результат до відображення.

Основні етапи роботи:

1. Сторона користувача:

- Введіть маршрут (пункти А і Б).
- При необхідності додайте проміжні зупинки.
- Встановити середню швидкість.
- Встановити інтервал аналізу погоди.
- Виберіть дату і час початку подорожі.
- Перейдіть до відображення результатів.

2 Інтерфейс системи :

- Отримайте дані з форми.
- Перевірте дані.
- Якщо є помилка, відобразиться повідомлення, і ви зможете її виправити.
- Відображається повідомлення про помилку.

3. Система аналізу маршруту:

- Починає будувати маршрут.
- Враховує зупинки та їх тривалість.
- Розраховує час прибуття на контрольно-пропускні пункти.
- Запитує погодні дані з API.
- Формування остаточного маршруту з урахуванням погодних умов.

4. Результат:

- Результат (маршрут з погодою) надсилається назад у додаток.
- Дані відображаються на карті та в таблиці з погодними параметрами.

Переваги використання діаграми активності:

- Дає повне уявлення про логіку обробки маршруту.

- Чітко розмежовує зони відповідальності користувача, інтерфейсу та бекенду.
- Допомагає виявити вузькі місця в логіці, які можуть потребувати оптимізації.
- Забезпечує хорошу основу для тестування (ви можете створювати тести на основі кожного переходу).

1.5 Діаграма послідовності

Діаграма послідовності в UML використовується для моделювання процесу обміну повідомленнями між об'єктами або компонентами системи в часовій послідовності. Вона особливо корисна для розуміння того, як дані рухаються через систему під час виконання певного сценарію - від ініціювання дії до отримання результату.

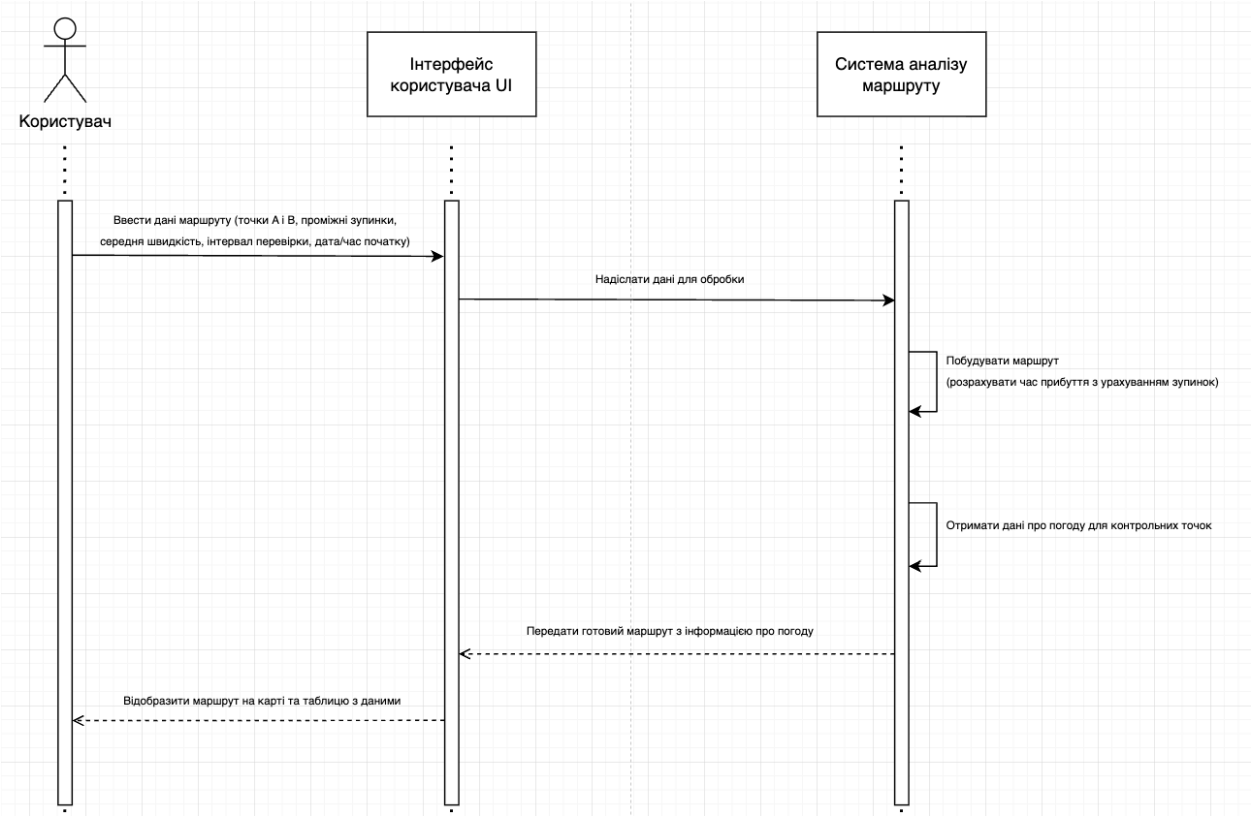

Рисунок 1.3 – Діаграма послідовності системи аналізу погоди на маршруті

У контексті системи аналізу погодних маршрутів діаграма послідовності показує взаємодію між користувачем, інтерфейсом користувача (UI) і системою аналізу маршрутів, починаючи з моменту введення параметрів маршруту і закінчуючи відображенням результату на карті і в таблиці.

Учасники послідовності:

1) Користувач - ініціює процес шляхом введення даних, необхідних для прокладання маршруту.
2) Інтерфейс користувача (UI) приймає введені дані і надсилає запит на сервер.
3) Система аналізу маршруту обробляє запит, розраховує маршрут, отримує погодні дані та формує відповідь.

Опис послідовності дій:

1. Користувач вводить параметри маршруту:

- пункти відправлення та прибуття (A → B),
- проміжні зупинки,
- середню швидкість,
- інтервал перевірки погоди,
- дату та час початку подорожі.

2. Інтерфейс користувача отримує ці дані і передає їх на сервер (систему аналізу маршруту).

3. Система аналізу маршруту виконує наступні дії:

- Прокладає маршрут із зупинками.
- Розраховує час прибуття в кожну контрольну точку на маршруті.
- Запитує погодні дані (через сторонній погодний API) для кожної контрольної точки.
- Формує повний маршрут з інформацією про погоду.

4. Сформований маршрут з погодними умовами передається в інтерфейс користувача.

5. В інтерфейсі користувача відображається маршрут на карті та таблиця з погодними даними.

Переваги цієї моделі:

- Чітко визначає, на якому етапі і який компонент системи за що відповідає.
- Відокремлює логіку фронтенду від логіки бекенду.
- Спрощує налагодження системи, коли мова йде про визначення джерела помилки або затримки.

Діаграма послідовності в UML використовується для моделювання процесу обміну повідомленнями між об'єктами або компонентами системи в часовій послідовності. Вона особливо корисна для розуміння того, як дані розміщуються в системі під час виконання певного сценарію, від ініціації дії до отримання результату.

1.6 Діаграма класів

Діаграма класів є важливим інструментом об'єктно-орієнтованого аналізу та проектування, за допомогою якого можна формалізувати основні сутності предметної області, їх атрибути, методи та логічні зв'язки між ними. У контексті розробки програмного забезпечення для аналізу погодних умов на маршруті, діаграма класів відображає логічну структуру системи, яка обробляє маршрут, погодні дані та візуалізацію результату.

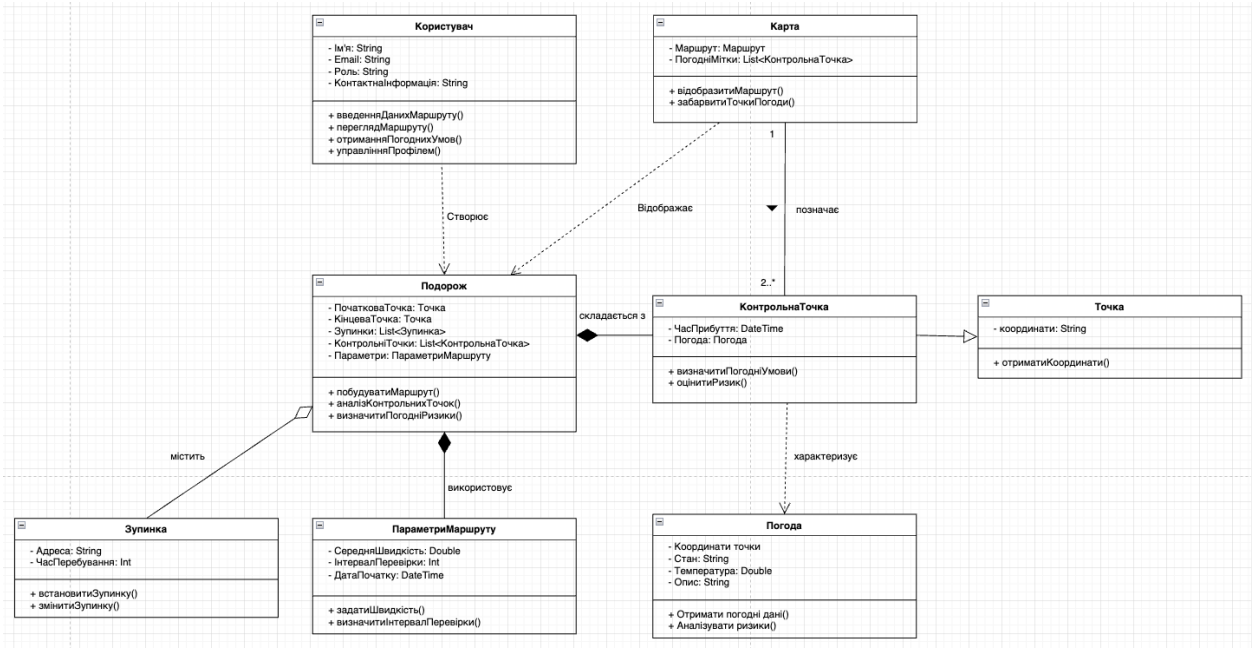

Рисунок 1.4 – Діаграма класів системи аналізу погоди на маршруті

Основні класи системи:

Користувач

Атрибути :

- Ім'я
- Адреса електронної пошти
- Роль
- Контактні дані

Методи :

- введенняДанихМаршруту()

- переглядМаршруту()

- отриманняПогодниxУмов()

- управлінняПрофілем()

Підключення: створює об'єкт Подорож.

Користувач є ініціатором всього процесу - він визначає маршрут, параметри та переглядає результати.

Подорож

Атрибути:

– ПочатковаТочка

– КінцеваТочка

– Зупинки: List<Зупинка>

– КонтрольніТочки: List<КонтрольнаТочка>

– Параметри: ПараметриМаршруту

Методи:

– побудуватиМаршрут()

– аналізКонтрольнихТочок()

– визначитиПогодніРизики()

Зв’язки:

– Складається з (composite) зупинок, контрольних точок і параметрів.

– Використовує клас Погода через КонтрольнаТочка.

Центральний клас, що агрегує всю логіку маршруту, взаємодіє з іншими сутностями для аналізу.

Точка

Атрибути:

– координати: String

Методи:

– отриматиКоординати()

Використовується в Подорож, КонтрольнаТочка і Зупинка.

Універсальний клас, що зберігає координати — основа для побудови просторової логіки.

Зупинка

Атрибути:

– Адреса

– ЧасПеребування

Методи:

– встановитиЗупинку()

– змінитиЗупинку()

Входить до маршруту (Подорож) — визначає точки, де водій зупиняється.

КонтрольнаТочка

Атрибути:

– ЧасПрибуття

– Погода

Методи:

– визначитиПогодніУмови()

– оцінитиРизик()

Зв’язки:

– асоційована з Точка

– включає об’єкт Погода.

Контрольні точки генеруються автоматично залежно від інтервалу перевірки погоди. Вони є критично важливими для отримання погодних умов у ключових місцях маршруту.

ПараметриМаршруту

Атрибути:

– СередняШвидкість

– ІнтервалПеревірки

– ДатаПочатку

Методи:

– задатиШвидкість()

– визначитиІнтервалПеревірки()

Цей клас конфігурує логіку розрахунку часу і частоти погодних перевірок.

Погода

Атрибути:

– координати точки

– стан

– температура

– опис

Методи:

– отриматиПогодніДані()

– аналізуватиРизики()

Є результатом API-запитів, який впливає на рівень ризику та візуалізацію маршруту.

Карта

Атрибути:

– маршрут

– погодні мітки

Методи:

– відобразитиМаршрут()

– забарвитиТочкиПогоди()

Клас, що відповідає за візуалізацію на карті, надає користувачу доступну інтерпретацію даних.

Пояснення зв’язків між класами

`Користувач → Подорож`

Зв’язок: асоціація (створює)

Чому: користувач ініціює подорож, вводячи дані маршруту.

`Подорож → Точка`

Зв’язок: асоціація (початкова і кінцева точки)

Чому: маршрут складається з точки A та точки B — це обов’язкові координати.

`Подорож → Зупинка`

Зв’язок: композиція

Чому: зупинки є невід’ємною частиною подорожі — не існують окремо.

`Подорож → КонтрольнаТочка`

Зв’язок: композиція

Чому: контрольні точки генеруються автоматично під конкретну подорож (за параметрами швидкості, інтервалу, тощо).

`КонтрольнаТочка → Точка`

Зв’язок: асоціація

Чому: кожна контрольна точка має координати (місце на карті), отримані з Точка.

`КонтрольнаТочка → Погода`

Зв’язок: агрегація

Чому: погодна інформація додається до контрольної точки на основі API, але існує як окремий об’єкт.

`Подорож → ПараметриМаршруту`

Зв’язок: композиція

Чому: параметри (швидкість, інтервал, дата) належать лише до конкретної подорожі.

`Карта → Маршрут, КонтрольнаТочка`

Зв’язок: асоціація

Чому: карта відображає маршрут і позначає погодні мітки на основі контрольних точок.

1.7 Діаграма пакетів

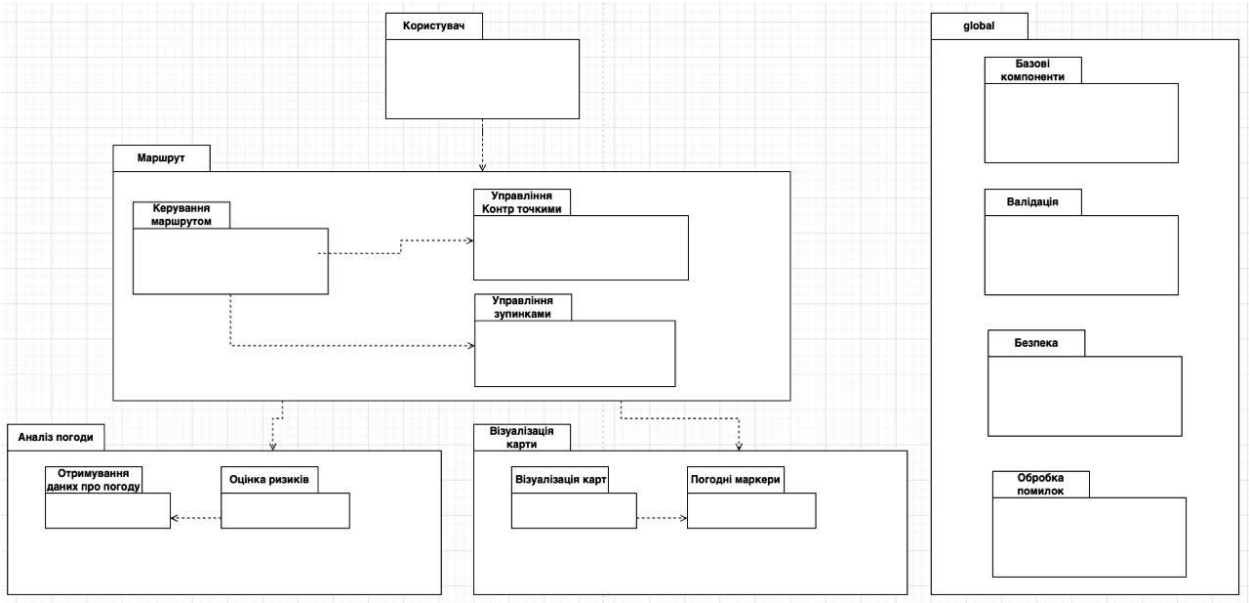

Рисунок 1.5 – Діаграма пакетів системи аналізу погоди на маршруті

Основні пакети:

Користувач

- Відповідає за взаємодію користувача із системою.

Маршрут

- Керування маршрутом – логіка створення, зміни, оновлення маршрутів.
- Контрольні точки – перевірка контрольних точок маршруту.
- Управління зупинками – додавання, зміна зупинок.

Аналіз погоди

- Дані про погоду – отримує інформацію про погоду з API.
- Оцінка ризиків – аналіз погодних умов на маршруті.

Візуалізація карти

- Візуалізація карт – рендеринг карти та маршрутів.
- Погодні маркери – відображення погодних умов на карті.

global (Глобальні сервіси)

- Базові компоненти – загальні утиліти та сервіси.
- Валідація – перевірка введених користувачем даних.
- Безпека – механізми автентифікації та захисту даних.
- Обробка помилок – централізоване управління винятками.

Як взаємодіють пакети:

1. Користувач створює маршрут у Маршруті.
2. Маршрут взаємодіє з Аналізом погоди, щоб отримати погодні дані та оцінити ризики.
3. Маршрут передає дані до Візуалізації карти, щоб маршрут та погодні умови відображалися.
4. Глобальні сервіси взаємодіють із усіма модулями для забезпечення коректності, безпеки та стійкості до помилок.

Діаграма пакетів, розроблена для системи аналізу погоди на маршруті, демонструє логічну структуру архітектури проєкту.

1.8 Діаграма компонентів

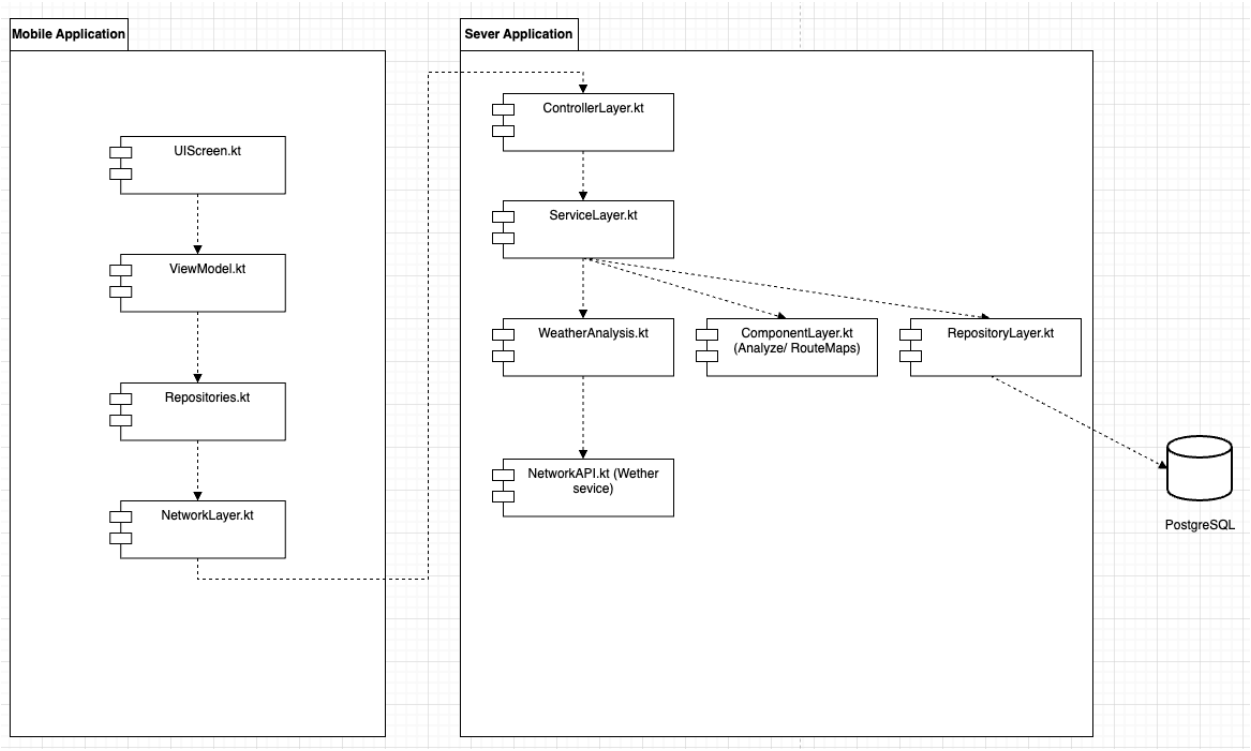

Рисунок 1.6 – Діаграма компонентів системи аналізу погоди на маршруті

Система складається з двох основних частин:

- Мобільний додаток на Android (клієнтська частина).
- Серверна частина на Spring Boot (обробка маршрутів та погоди).

Основний сценарій роботи:

1. Користувач вводить дані маршруту в додатку.
2. Дані відправляються на сервер за допомогою HTTP-запиту.
3. Сервер розраховує маршрут і збирає інформацію про погоду вздовж маршруту.
4. Сервер повертає результат назад у додаток.
5. Мобільний додаток відображає дані в інтерфейсі (маршрут + погодні умови).

Ця компонентна діаграма моделює архітектуру інформаційної системи аналізу погоди в дорозі, яка складається з:

- Мобільний додаток (Android)

- Серверний додаток (Spring Boot)

- База даних (PostgreSQL)

- Зовнішній погодний сервіс (Weather API)

Система працює за принципом клієнт-серверної взаємодії: мобільний додаток ініціює запит, сервер обробляє дані про маршрут і погодні умови, зберігає дані в базі даних і повертає результат користувачеві.

Мобільний додаток

Компоненти:

- UIScreen.kt - відображення користувацького інтерфейсу для введення маршруту та перегляду результатів.

- ViewModel.kt - логіка управління станом екранів. Отримує дані від користувача та передає їх до репозиторіїв.

- Repositories.kt - обробка бізнес-логіки на клієнті: виклики API, обробка відповідей.

- NetworkLayer.kt - безпосередня взаємодія з мережею (HTTP-запити через Retrofit або подібну бібліотеку).

Потік даних:

1. Користувач взаємодіє з UIScreen.kt.
2. UIScreen.kt передає дані до Repositories.kt через ViewModel.kt.
3. Repositories.kt викликає методи NetworkLayer.kt, який надсилає запит до сервера.

Server Application

Компоненти:

- ControllerLayer.kt — REST-контролери, які обробляють вхідні HTTP-запити з мобільного застосунку.
- ServiceLayer.kt — бізнес-логіка: обробка маршруту, виклик аналізу погоди, агрегація даних.
- WeatherAnalysis.kt — окремий компонент для аналізу погодних умов на основі отриманої інформації.
- ComponentLayer.kt (Analyze/RouteMaps) — логіка побудови маршруту і проведення різного виду аналізів маршруту.
- NetworkAPI.kt (Weather service) — модуль для взаємодії із зовнішнім API погодних сервісів (наприклад, OpenWeatherMap).
- RepositoryLayer.kt — доступ до бази даних PostgreSQL (JPA репозиторії), зберігання і вибірка даних маршрутів і погоди.

Потік даних:

1. ControllerLayer.kt приймає дані маршруту.
2. ServiceLayer.kt обробляє маршрут і викликає WeatherAnalysis.kt для аналізу погодних умов.
3. WeatherAnalysis.kt отримує погодні дані через NetworkAPI.kt.
4. ServiceLayer.kt за необхідності взаємодіє з ComponentLayer.kt для складнішої обробки даних маршруту (наприклад, побудови оптимальних точок перевірки).
5. Дані зберігаються через RepositoryLayer.kt у базі PostgreSQL.
6. Після обробки результат надсилається назад на мобільний додаток.

Database

PostgreSQL: Зберігає інформацію про маршрути, погодні умови на маршрутах, користувацькі запити. Сервер взаємодіє з базою даних через RepositoryLayer.kt.

Взаємодія системи

- Мобільний застосунок ініціює запит через мережевий рівень (NetworkLayer.kt).
- Сервер обробляє запит через контролери (ControllerLayer.kt), сервіси (ServiceLayer.kt) та репозиторії (RepositoryLayer.kt).
- Сервер у разі потреби звертається до зовнішнього погодного API (NetworkAPI.kt).
- Результат обробки маршруту і погоди повертається назад у мобільний застосунок для відображення у UIScreen.kt.

Дана діаграма дозволяє краще зрозуміти архітектуру програми, полегшує обслуговування коду та сприяє його масштабуванню в майбутньому.

1.9 Діаграма розгортання

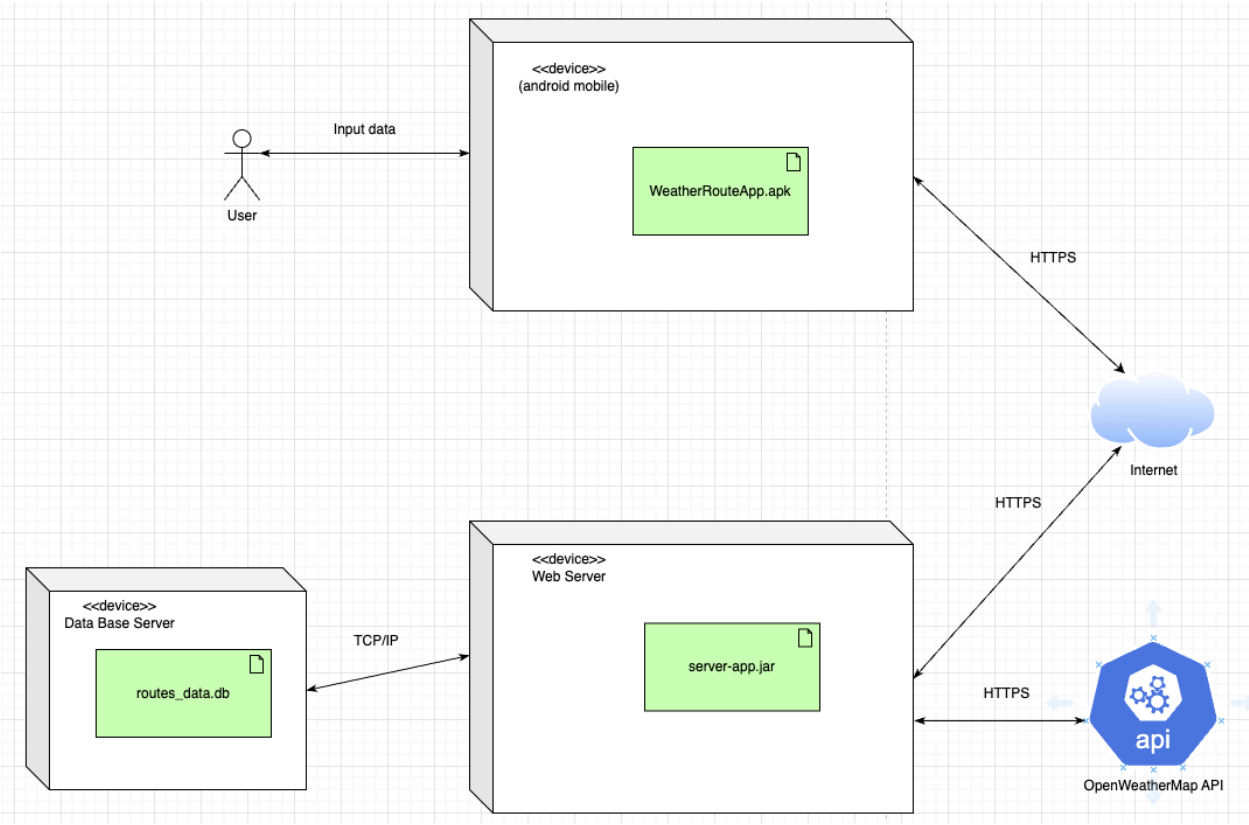

Рисунок 1.7 – Діаграма розгортання системи аналізу погоди на маршруті

Опис діаграми розгортання

На діаграмі розгортання зображено фізичну інфраструктуру роботи програмного забезпечення системи аналізу погоди на маршруті:

- Користувач взаємодіє з мобільним пристроєм (android mobile), на якому встановлений артефакт WeatherRouteApp.apk.
- Через Інтернет (з'єднання за протоколом HTTPS) Android-додаток надсилає запити на вебсервер (Web Server), де розгорнуто артефакт server-app.jar.
- Web Server обробляє запити:
    - Використовуючи TCP/IP з'єднання, зберігає та отримує дані маршрутів з серверу бази даних (Data Base Server), який містить артефакт routes_data.db.

- Для аналізу погодних умов Web Server надсилає запити до зовнішнього погодного API (OpenWeatherMap API) через HTTPS-з'єднання.

- Після отримання інформації сервер обробляє дані та повертає результати аналізу назад до Android-додатку, де користувач може переглянути результати своєї подорожі.

Дана діаграма відображає фізичне розгортання компонентів системи, типи взаємодій між ними та спосіб обміну даними.

Опис зв’язку між сервером додатку і базою даних:

PostgreSQL JDBC (Java Database Connectivity) — це стандартний драйвер, який використовується для передачі даних між сервером додатку (Spring Boot) і базою даних PostgreSQL. Він працює поверх протоколу TCP/IP (порт 5432 за замовчуванням).

Як це працює у Spring Boot.

1. Spring Boot формує SQL-запити

- Коли сервер обробляє запит користувача, використовуючи сервіси або репозиторії (@Repository), Spring Data JPA або власні DAO-класи формують SQL-запити.
- Якщо використовується ORM (наприклад, Hibernate), запити можуть бути сформовані автоматично з допомогою JPQL або Criteria API.

2. SQL-запити передаються через драйвер JDBC

- Spring Boot використовує PostgreSQL JDBC драйвер (org.postgresql.Driver) для комунікації з базою даних.
- Цей драйвер реалізує стандарт інтерфейсу JDBC і забезпечує роботу протоколу з PostgreSQL.

3. Передача запиту через TCP/IP

- Драйвер відкриває TCP/IP-з'єднання з сервером бази даних на порт 5432.
- Запити надсилаються у стандартизованому текстовому форматі SQL через захищене або незахищене з'єднання (залежно від конфігурації SSL/TLS).

4. PostgreSQL сервер обробляє запит і повертає результат

    - СУБД виконує отриманий SQL-запит і надсилає відповідь у вигляді рядків результату або повідомлення про успішність операції назад через те саме TCP/IP-з'єднання.

5. Spring Boot розбирає відповідь і перетворює її у об'єкти Java

- Якщо використовується ORM (Hibernate), результати автоматично мапляться у відповідні Java-об'єкти (наприклад, ентіті-класи, що позначені @Entity).
- Якщо використовується чистий JDBC (JdbcTemplate), результати обробляються вручну через ResultSet.

2 ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ СИСТЕМИ

2.1 Опис структури даних для маршруту та погоди

У процесі розробки програмного забезпечення важливу роль відіграє правильне проектування структури даних. Для зберігання, опрацювання та аналізу інформації про маршрути, точки зупинки, параметри поїздки та погодні умови необхідно побудувати логічну модель бази даних. Така модель дає змогу ефективно структурувати інформацію, уникати дублювання даних і забезпечувати надійність і масштабованість системи.

Для цього застосовується ER-діаграма (Entity-Relationship diagram) - діаграма «сутність-зв'язок», яка є класичним інструментом моделювання структури бази даних. Вона відображає сутності (таблиці), їхні атрибути (поля) і зв'язки між ними (асоціації, залежності, підпорядкування). Кожна сутність представляє об'єкт реального світу, а зв'язки показують, як ці об'єкти взаємодіють один з одним.

Що таке ER-діаграма?

ER-діаграма (Entity-Relationship diagram) - це графічне представлення логічної структури даних, що складається з таких основних елементів:

- Сутність (Entity) - об'єкт або поняття, яке має значення для предметної області (наприклад: Користувач, Маршрут, Зупинка, Погода).
- Атрибути (Attributes) - характеристики сутності (наприклад: ім'я користувача, координати точки, температура).
- Зв'язки (Relationships) - асоціації між сутностями (наприклад: користувач створює маршрут; маршрут має зупинки).

ER-діаграма є основою для створення реляційної бази даних. На її основі формується схема БД, що включає таблиці, зовнішні ключі, індекси, первинні ключі тощо.

Структура даних у системі аналізу погоди на маршруті

У нашій системі потрібно зберігати дані про:

- користувачів, які створюють маршрути;
- самі маршрути і пов'язані з ними зупинки
- контрольні точки маршруту для аналізу погоди;
- параметри маршруту, які впливають на розрахунок;
- погодні дані, отримані з API.

2.2 Побудова ER- діаграми

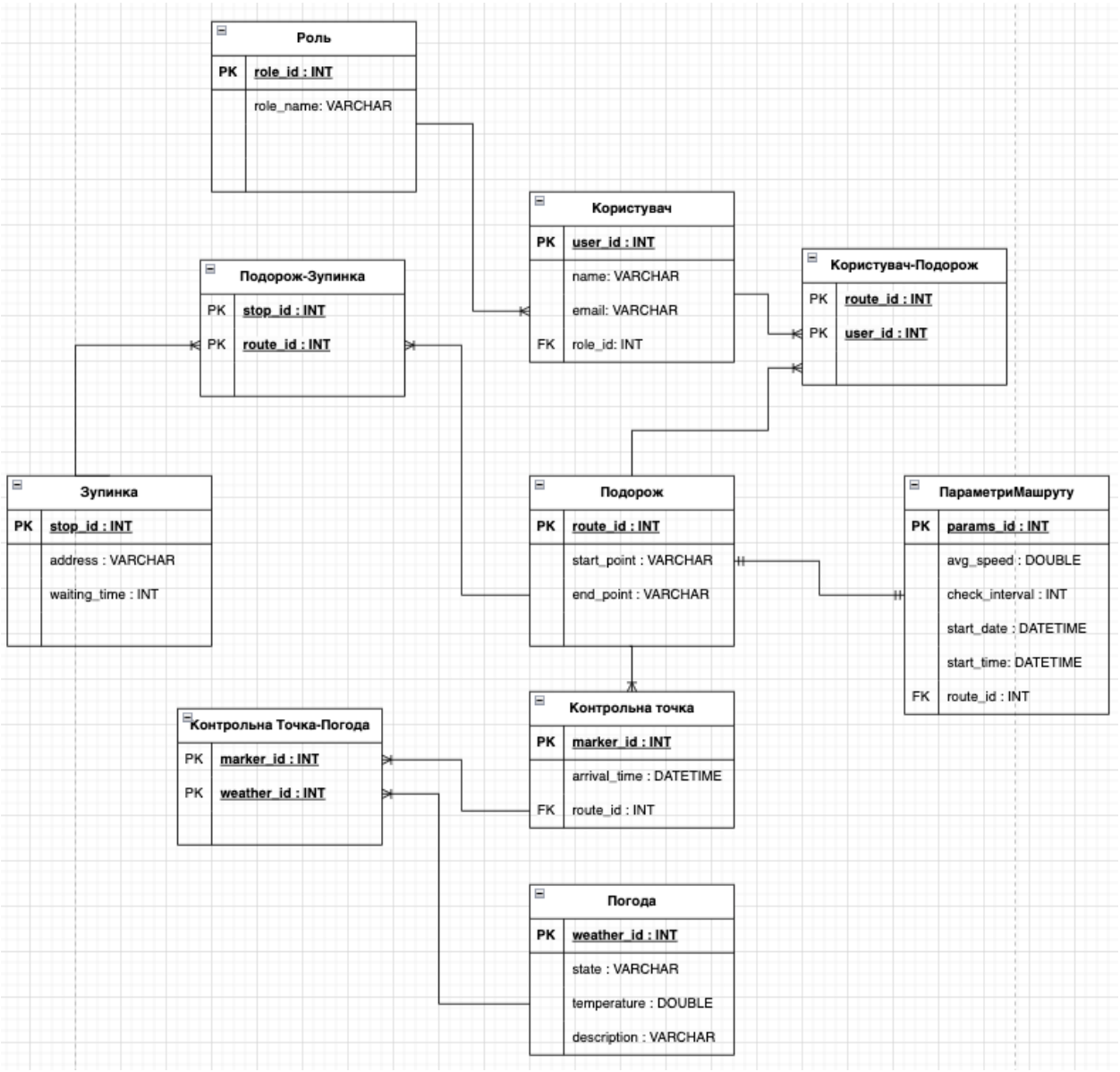

Рисунок 1.3 – ER діаграма системи аналізу погоди на маршруті

Детальний опис таблиць та атрибутів:

Role (Роль)

- role_id – первинний ключ (автоінкрементний ідентифікатор ролі).
- role_name – назва ролі (наприклад, "користувач", "адмін").

User (Користувач)

- user_id – первинний ключ користувача.
- name – ім'я користувача.
- email – унікальний email користувача.
- role_id – зовнішній ключ до ролі.

Trip (Подорож)

- route_id – первинний ключ маршруту.
- start_point – точка старту маршруту.
- end_point – точка закінчення маршруту.

UserTrip (Зв’язок користувача з подорожами)

- user_id – зовнішній ключ до користувача.
- route_id – зовнішній ключ до маршруту.
- Первинний ключ: комбінація user_id, route_id.

RouteParams (Параметри маршруту)

- params_id – первинний ключ параметрів маршруту.
- avg_speed – середня швидкість руху.
- check_interval – інтервал перевірки погоди.
- start_datetime – дата та час початку подорожі.
- route_id – зовнішній ключ до конкретного маршруту (один маршрут – одні параметри).

Checkpoint (Контрольна точка)

- marker_id – первинний ключ точки.
- arrival_time – час прибуття в контрольну точку.
- route_id – зовнішній ключ до маршруту.

Weather (Погода)

- weather_id – первинний ключ погоди.
- state – стан погоди (сонячно, дощ тощо).
- temperature – температура повітря.
- description – детальний опис погодних умов.

CheckpointWeather (Погода контрольної точки)

- marker_id – зовнішній ключ до контрольної точки.
- weather_id – зовнішній ключ до погоди.
- Первинний ключ: комбінація marker_id, weather_id.

Stop (Зупинка)

- stop_id – первинний ключ зупинки.
- address – адреса зупинки.
- waiting_time – час очікування.
- lat, lng – координати (широта та довгота).

RouteStops (Маршрутні зупинки)

- stop_id – зовнішній ключ до зупинки.
- route_id – зовнішній ключ до маршруту.
- Первинний ключ: комбінація stop_id, route_id.

Опис зв'язків між таблицями:

Один-до-багатьох (1:N)

`Role→User`

`Одна роль може бути у багатьох користувачів.`

Trip→RouteParams

Один маршрут має один набір параметрів.

Trip→Checkpoint

Один маршрут має багато контрольних точок.

Багато-до-багатьох (M:N) через проміжні таблиці

User↔Trip(UserTrip)

Один користувач може мати кілька маршрутів, один маршрут може бути у кількох користувачів.

Checkpoint↔Weather(CheckpointWeather)

Одна погода може бути у кількох контрольних точках, одна точка може мати кілька прогнозів погоди.

Trip↔Stop(RouteStops)

Один маршрут може мати багато зупинок, одна зупинка може бути частиною кількох маршрутів.

Перевірка на відповідність 3НФ:

1НФ(Перша нормальна форма)

Всі атрибути атомарні.

Відсутні повторювані групи атрибутів.

2НФ (Друга нормальна форма)

Кожна таблиця має первинний ключ.

Кожне поле залежить повністю від первинного ключа (немає часткових залежностей).

3НФ (Третя нормальна форма)

Відсутні транзитивні залежності (кожне поле залежить лише від первинного

ключа).

Всі сутності, що можуть повторюватися, винесені в окремі таблиці.

2.3 Вибір та обґрунтування СУБД

Система управління базами даних (СУБД) - це ключовий компонент інформаційної системи, який відповідає за зберігання, доступ, обробку та захист даних. При розробці програмного забезпечення для аналізу погоди на маршруті потрібно вибрати СУБД, яка забезпечує

- стабільну роботу в архітектурі клієнт-сервер,
- підтримку географічних координат і часових даних,
- масштабованість і продуктивність,
- безпеку та гнучкість у роботі з погодними та маршрутними даними.

Для цього виконується порівняльний аналіз різних типів СУБД за критеріями відповідності поставленим завданням.

Порівння основних СУБД

Назва	Тип	Переваги	Недоліки
PostgreSQL	Реляційна SQL	Підтримка геоданих (PostGIS), часових поясів, JSON Надійність Розширюваність	Вимагає більше ресурсів для розгортання та налаштування
MySQL	Реляційна SQL	Швидка та проста в налаштуванні Велика спільнота	Обмежена підтримка географічних операцій Менше можливостей з JSON
SQLite	Вбудованна SQL	Легка, не потребує сервера Підходить для мобільного кешування	Не підходить для складних серверних обчислень Однокористувацький режим

MongoDB	Документно-орієнтована (NoSQL)	Гнучка структура документів Добре підходить для даних JSON	Немає транзакцій на рівні SQL Складні агрегації та геоаналітика
Firebase Realtime / Firestore	NoSQL	Інтеграція з Android Робота в режимі реального часу	Не підходить для складних аналітичних запитів Вартість залежить від обсягу
Redis	In-memory ключ-значення	Висока швидкість Підходить для кешу	Не використовується для довготривалого зберігання маршрутних даних

Системні вимоги до СУБД:

- Система аналізу погоди на маршруті повинна зберігати
- Користувачів (аутентифікація, історія маршрутів),
- Маршрути (початок, кінець, зупинки, параметри),
- Контрольні точки (координати, час, прогноз),
- Погодні дані (стан, температура, вітер, опис),
- Історія запитів.

Також необхідна підтримка для:

- зберігання часових форматів (DateTime) для розрахунку часу проходження;
- географічних координат (широта, довгота);
- JSON-структур для тимчасових відповідей про погоду з API;
- індекси для швидкого пошуку на маршрутах;
- масштабованість для розширення системи.

Обґрунтування вибору: PostgreSQL

Проаналізувавши можливі варіанти, найбільш підходящою СУБД для реалізації даної системи є PostgreSQL, з наступних причин:

1. Підтримка геоданих (PostGIS) - необхідна для роботи з координатами точок маршруту.
2. Розширена підтримка часу - з точними часовими поясами (для розрахунку погодних даних у часі).
3. JSON + реляційна модель - дозволяє зберігати та обробляти дані з API у гнучкий, структурований спосіб.
4. Надійність та властивості ACID - гарантує, що важливі історичні дані зберігаються без втрат.
5. Безкоштовний та відкритий код - відсутність ліцензійних обмежень для майбутнього масштабування.
6. Можна розгортати в хмарі або локально - підтримує сервери Docker, Kubernetes, Ubuntu/Rocky Linux.

Виходячи з вимог до функціональності, складності обробки даних, структури предметної області та очікуваного робочого навантаження, PostgreSQL є найкращим вибором в якості основної СУБД для зберігання маршрутних та погодних даних. Вона поєднує в собі гнучкість, стабільність і розширюваність, що робить її ідеальною платформою для побудови надійного бекенду для аналізу погодних маршрутів.

3 РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ СИСТЕМИ АНАЛІЗУ ПОГОДИ НА МАШРУТІ

3.1 Архітектура та складові частини програмної системи

Програмне забезпечення системи аналізу погоди на маршруті побудовано за принципом клієнт-серверної архітектури. Вибір саме такої моделі був зумовлений низкою чинників:

- розподілом відповідальностей між пристроєм користувача та серверною частиною;
- можливістю централізовано обробляти запити до зовнішніх погодних API;
- підвищенням безпеки (вся логіка і доступ до ключів API винесені на сервер);
- масштабованістю, яка дає змогу додавати нові компоненти без зміни клієнта;
- спрощенням оновлень: нова логіка впроваджується тільки на сервері.

Загальна структура системи

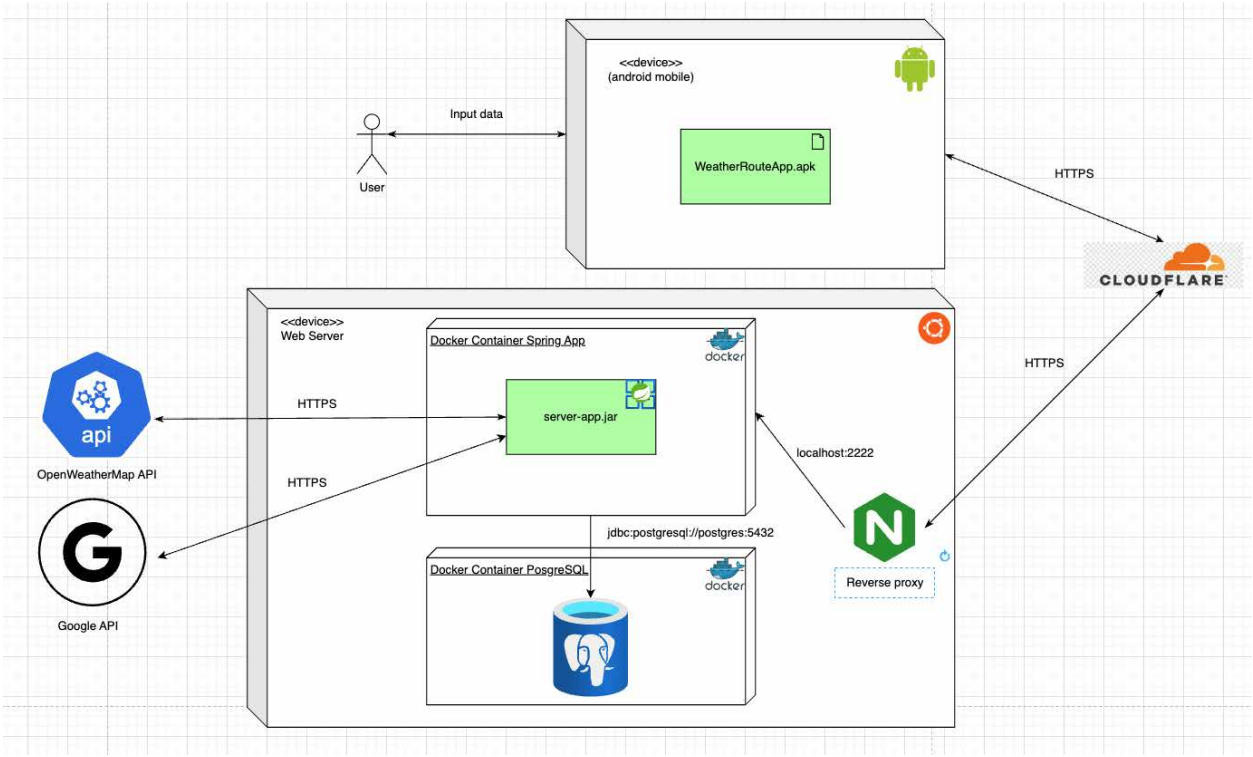

Рисунок 3.1 – Загальна структура системи

1. Клієнт: Android-додаток WeatherRouteApp.apk

Платформа: Android

Мова програмування: Kotlin

Фреймворки: Jetpack Compose (UI), Coroutines , Retrofit.

Роль у системі: користувацький інтерфейс:

- Користувач вводить маршрут (початкова і кінцева точка, зупинки, швидкість, дата).
- Додаток формує HTTPS-запит до сервера.
- Отримує відповідь із погодними даними та ризиками.
- Візуалізує маршрут на карті (Google Maps SDK) і показує таблицю.

Чому Android: Це найпопулярніша мобільна платформа з гнучкими інструментами розробки, чудово підходить для польових додатків.

2. Cloudflare

Роль: захисний DNS-проксі та HTTPS-шифрування.

Що робить:

- Приймає запити від Android-додатка.
- Шифрує з'єднання через HTTPS.
- Захищає сервер від DDoS-атак і несанкціонованого доступу.
- Сховає реальну IP-адресу сервера.

Чому Cloudflare: це безкоштовний сервіс для SSL, DNS і базового веб-захисту, який легко інтегрується з доменом.

3. Nginx

Роль: реверс-проксі, маршрутизатор запитів

Що робить:

- Приймає HTTPS-запити з Cloudflare.
- Перенаправляє їх на внутрішній порт Docker-контейнера Spring Boot (localhost:2222).
- Може обробляти кілька сервісів одночасно.

Чому саме Nginx: він швидкий, стабільний і дуже гнучкий для проксингу Docker-сервісів, простий у налаштуванні.

4. Docker-контейнер Spring Boot Kotlin (server-app.jar)

Фреймворк: Spring Boot (Kotlin)

Формат запуску: .jar файл у Docker-контейнері

Роль: обробка всієї бізнес-логіки

Що робить:

- Отримує запит від Android-додатку.
- Виконує обробку маршруту (розрахунок точок, часу прибуття, зупинок).
- Надсилає запити в OpenWeatherMap API і Google Maps API.
- Обробляє дані та оцінює погодні ризики.
- Зберігає результат у базу PostgreSQL.
- Повертає відповідь клієнту.

Чому Kotlin + Spring Boot:

Kotlin - сучасна, безпечна та лаконічна мова.

Spring Boot - надійний фреймворк із широким ком'юніті та підтримкою REST/WebSocket, JPA.

5. Docker-контейнер PostgreSQL

СУБД: PostgreSQL

Роль: постійне зберігання даних

Що зберігається:

- Дані про маршрути.
- Контрольні точки з погодою.
- Історія користувацьких запитів.
- Вхідні параметри подорожі.

Чому PostgreSQL:

- Потужна підтримка timestamp, double, jsonb.
- Безкоштовна, надійна, сумісна з JPA/Hibernate.
- Можливість додати PostGIS для просторового аналізу.

6. OpenWeatherMap API та Google Maps API

- OpenWeatherMap - отримання погодних даних (температура, вітер, опади, опис).
- Google Maps API - розрахунок маршруту, відстаней, координат.

Чому саме ці API:

- OpenWeatherMap має безкоштовний тариф і надає погодні дані з деталізацією по годинах.
- Google API гарантує точність побудови маршруту та географічних координат.

Загальний потік даних:

1. Користувач вводить дані → Android-додаток → HTTPS-запит → Cloudflare → Nginx → Spring Boot (Docker).
2. Spring Boot розраховує точки, звертається до погодних API → отримує прогноз → зберігає в PostgreSQL.
3. Spring Boot формує відповідь → повертає дані на Android-додаток → карта + таблиця ризиків.

Представлена архітектура є сучасною, безпечною та гнучкою. Вона дає змогу:

- централізовано керувати логікою та даними;
- ізолювати сервіси через Docker;
- забезпечити масштабування;
- швидко адаптувати систему до нових потреб (додавання аналітики, нових API тощо).

Завдяки використанню Cloudflare + HTTPS + Nginx, забезпечено високий рівень безпеки, а застосування Spring Boot і PostgreSQL у Docker дає змогу розгортати систему навіть у домашніх умовах без складної конфігурації.

3.2 Реалізація інтерфейсу користувача для мобільного пристрою

Інтерфейс користувача (UI) є ключовим елементом будь-якого мобільного додатку, особливо при взаємодії з сервісом, який вимагає введення даних маршруту, вибору опцій і перегляду результатів. Щоб забезпечити найкращий користувацький досвід (UX), я використовував інструмент Figma для проектування, створення прототипу та затвердження зовнішнього вигляду кожного екрану.

Figma: чому я обрав саме його?

Figma - це сучасна хмарна платформа для UI та UX дизайну, яка дозволяє:

- створювати макети екранів з інтерактивними прототипами ;
- організувати командну роботу в реальному часі;
- експортувати готові до використання ресурси (іконки, кольори, шрифти)
- використовувати систему компонентів, що прискорює проектування
- одразу візуалізувати інтерфейс на пристрої за допомогою мобільного додатку Figma Mirror.

Цей інструмент дозволив мені швидко протестувати UX-рішення, перевірити інтуїтивність взаємодії та адаптувати дизайн до сучасних рекомендацій щодо користувацького інтерфейсу Android.

Детальний опис екранів у мобільному додатку

Екран 1: Пошук місця

Елемент: поле введення з підказками.

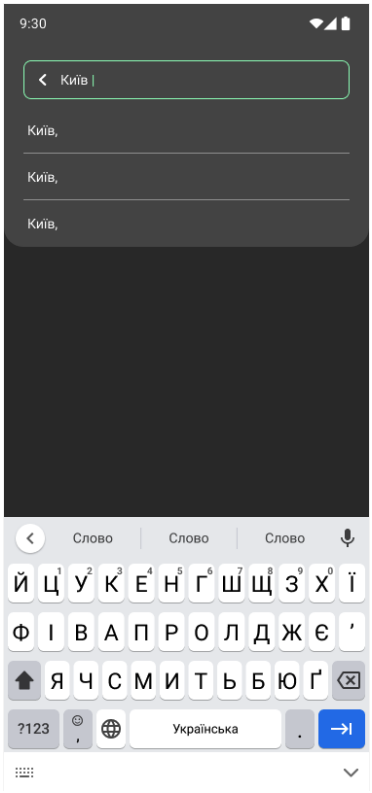

Рисунок 3.2 – Екран пошуку місця

UX рішення: після введення назви міста з'являються підказки (автозаповнення), що скорочує час введення.

Деталі користувацького інтерфейсу:

- Текст білого кольору на темному фоні.
- Активне поле підсвічується зеленим кольором.
- Клавіатура - українська локалізація (приклад: Київ).

Мета UX: зробити пошук максимально швидким і безпомилковим.

Екран 2: Початкове планування маршруту

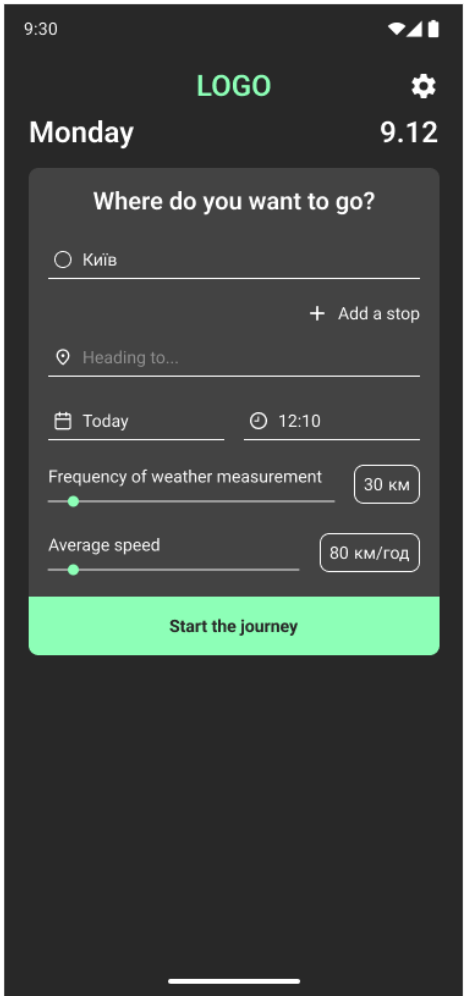

Рисунок 3.3 – Екран планування машруту

Блоки:

- Дата та час початку подорожі
- Початковий пункт (автозаповнення)
- Кнопка "Додати зупинку
- Кінцевий пункт

- Курсор "Частота вимірювання погоди" (інтервал перевірки погоди, наприклад, кожні 30 км)
- Повзунок «Середня швидкість» (наприклад, 80 км/год)
- Кнопка початку маршруту - Почати подорож

Принцип UX: все зібрано в одному вікні - жодних вкладок чи переходів. Мінімум дій перед стартом.

UI:

- Основний акцентний колір - м'який зелений (виділяє активні елементи).
- Темна тема - знижує зорову втому, особливо вночі.
- Контрастна типографіка - шрифт легко читається.

Екран 3: Введення зупинки

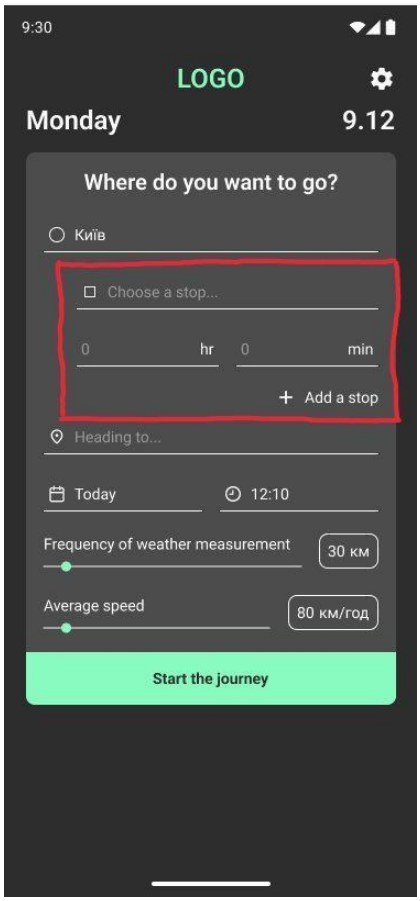

Рисунок 3.4 – Екран введення зупинки

Додаткове поле після «Додати зупинку» дозволяє вказати:

- Назву зупинки
- Тривалість перебування в годинах/хвилинах

Після введення вона відображається в списку маршрутів.

UX перевага: користувачі можуть створити маршрут з декількома пунктами без необхідності вносити складні зміни.

Екран 4: Перегляд маршруту на карті

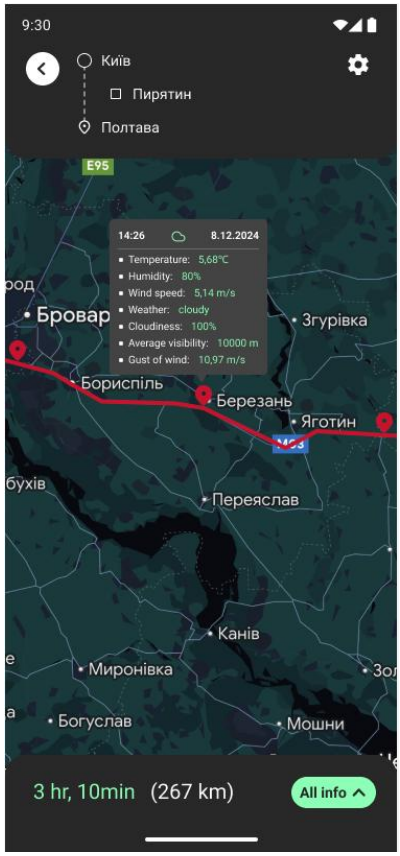

Рисунок 3.5 – Екран перегляду машруту на карті з показниками погоди

Карта з маршрутом (Google Maps).

Червоні маркери – визначення погоди у відповідній точці.

При натисканні на маркер відкривається спливаюче вікно з даними:

- час

- температура
- вітер
- вологість
- Хмарність
- Видимість

Перевага: карта - це не просто візуалізація, а інтерактивний інструмент аналізу.

Екран 5: Статистика маршруту

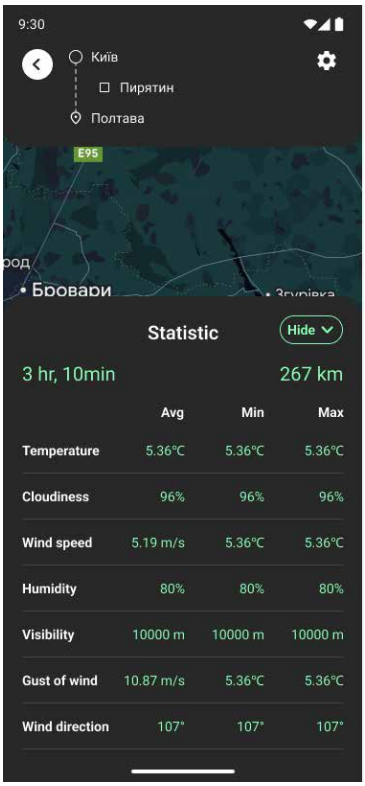

Рисунок 3.6 – Екран ввиведення статистики погоди на маршруті

Відображається:

- Загальний час
- Відстань
- Таблиця: середні / мінімальні / максимальні значення температури, вологості, вітру, напрямку.

Кнопка приховати / показати - зменшує статистику.

UI/UX plus :

- Інформативне представлення аналізів без перевантаження.
- Дизайн в дусі Google Weather / AccuWeather знайомий користувачеві.

Загальні принципи дизайну

- Темна тема - краща читабельність для мобільного використання.
- Мінімалізм - на екрані відображаються лише необхідні елементи.
- Іконки - використовуються універсальні символи (календар, стрілки, плюс).
- Адаптивність - інтерфейс оптимізовано під різні розміри екрану.
- Інтерактивність - всі дії відбуваються на одному екрані, без зайвої навігації.

Інтерфейс мобільного додатку був створений за допомогою інструменту Figma, що дозволило нам створити логічно послідовний, візуально чистий і функціональний UX. Кожен елемент призначений для швидкого, зручного та безпечного введення даних про маршрут та візуалізації погодної ситуації. Цей дизайн орієнтований на водіїв, туристів та логістів - користувачів, які цінують швидку взаємодію та простоту.

3.3 Реалізація серверної частини системи на базі Spring Boot

Серверна частина системи реалізована з використанням Spring Boot, популярного фреймворку для побудови надійних і масштабованих бекенд-додатків. Мова програмування - Kotlin, яка є сучасною, лаконічною та безпечною альтернативою Java.

Чому саме Spring Boot і Kotlin?

- Spring Boot - Забезпечує автонастроювання, REST контролери, базу даних, DI, безпеку.
- Kotlin - Менше коду, краща читабельність, сумісність з екосистемою Java.
- Швидкий запуск - Простий @SpringBootApplication запускає сервіс за 1-2 секунди.
- Гнучка архітектура - Чіткий поділ на компоненти, сервіси та моделі.
- Хороша інтеграція з PostgreSQL, REST API, зовнішніми клієнтами (наприклад, OpenWeatherMap)

Структура проєкта:

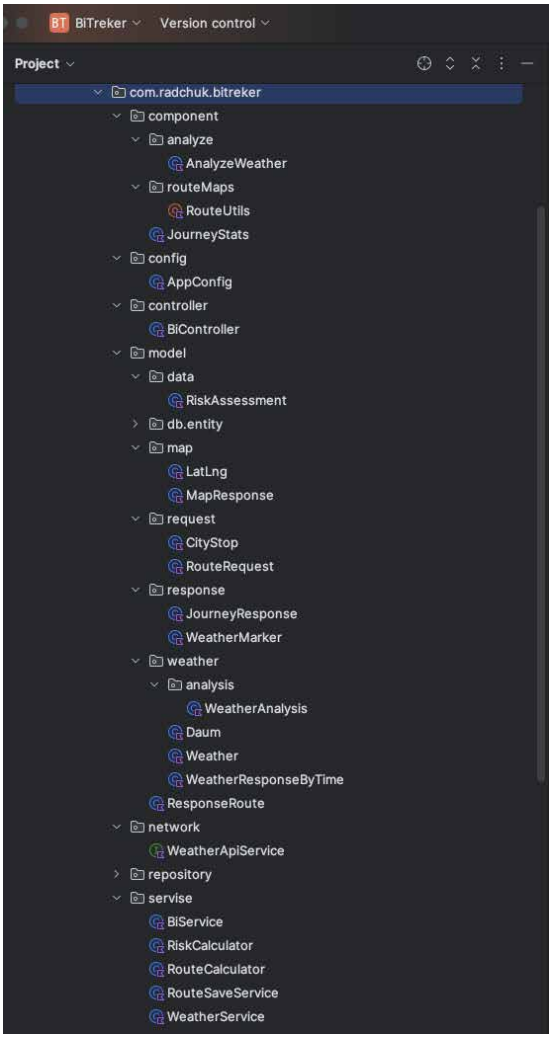

Рисунок 3.7 – Структура проєкта Spring

Package - controller

- BiController: головний REST-контролер, який приймає запити з Android-додатку (RouteRequest) і повертає відповіді (JourneyResponse, ResponseRoute).
- Реалізовано через @RestController.

Package - service

Відповідає за бізнес-логіку:

- BiService — головний сервіс, який координує обчислення та відповіді.
- RouteCalculator — розрахунок точок маршруту та часу прибуття.
- WeatherService — інтеграція з OpenWeatherMap API.
- RouteSaveService — збереження маршруту та результатів у базу.
- RiskCalculator — оцінка погодних ризиків.

Package - repository

- DAO-рівень для збереження в PostgreSQL (через Spring Data JPA).

Package - network

- WeatherApiService: клас, який відповідає за відправку HTTP-запитів до зовнішнього погодного API та обробку JSON-відповідей.

Package - model

Сутності, DTO та моделі:

- RouteRequest, CityStop — вхідні дані.
- JourneyResponse, WeatherMarker, WeatherResponseByTime — відповіді до Android.
- LatLng, MapResponse — координати і мапінг.

- Weather, Daum, WeatherAnalysis — погодні моделі.
- RiskAssessment — структура з рівнем погодного ризику.
- ResponseRoute — фінальний об’єкт-відповідь на клієнт.

Package - component

- AnalyzeWeather, RouteUtils, JourneyStats: допоміжна логіка обробки маршруту, агрегація статистики, підрахунки (відстань, середня швидкість, час прибуття).

Package - config

- AppConfig: конфігурація Bean-компонентів (наприклад, WebClient, ObjectMapper).

3.4 Розгортання власного сервера в умовах домашньої мережі

Реалізувавши серверну частину системи на Spring Boot, мені потрібно було зробити її загальнодоступною, щоб будь-який мобільний клієнт міг надсилати HTTP-запити до API. Найпростішим способом зробити це було б використання орендованого віртуального сервера (VPS), але я вирішив піти складнішим, але більш повчальним шляхом - створити власний сервер вдома.

Що таке Деплой?

Деплой (від англ. deploy) - це процес розміщення готового програмного продукту (додатку, сервісу) на сервері з метою його запуску, надання доступу користувачам та інтеграції з іншими системами. У нашому випадку розгортання складалося з

- запуску Spring Boot-додатку на Linux-сервері ;
- надання йому доступу через Інтернет
- налаштування з'єднання з базою даних;
- управління мережею, портами, доменом та проксі.

Вибір та підготовка обладнання

Я використовував звичайний настільний комп'ютер:

Рисунок 3.8 – Настільки комп'ютер який виступить як сервер.

- Модель: Lenovo ThinkCentre
- Процесор: Intel i5-7500T з тактовою частотою 2,7 ГГц (4 ядра)
- ОПЕРАТИВНА ПАМ'ЯТЬ: 8 ГБ
- Накопичувач: SSD на 128 ГБ

Цієї конфігурації достатньо для запуску Docker-контейнерів з Spring Boot та PostgreSQL, а також для обробки запитів від декількох користувачів одночасно.

Встановлення операційної системи - Ubuntu Server

Я встановив на свій комп'ютер Ubuntu Server 24.04.2 LTS - стабільну серверну версію Linux без графічного інтерфейсу, оптимізовану для продуктивної роботи сервісів.

Переваги Ubuntu Server :

- мінімальне використання ресурсів ;
- легке оновлення через apt;
- відмінна документація та підтримка;
- просте управління через термінал (SSH).

Підключення до локальної та IP-мережі

Комп'ютер підключається до роутера за допомогою кабелю Ethernet надійніше, ніж через Wi-Fi.

У налаштуваннях роутера я одразу налаштовую статичну локальну IP-адресу 192.168.1.195. Це необхідно для того, щоб інші пристрої в мережі мали постійний доступ до сервера. І після перезапуску сервера не змінювалася локальна мережа.

Перевірив IP-адресу на тип

Після підключення до інтернету я перевірив, який тип зовнішньої IP-адреси надає мій провайдер:

- Сіра IP-адреса - це внутрішня адреса, яку спільно використовують кілька користувачів. Сервер, який залежить від неї, не доступний безпосередньо з Інтернету.

- Біла IP-адреса - це унікальна глобальна адреса, доступна з Інтернету.

Мені пощастило: мій провайдер надав мені білу IP-адресу, тому мій сервер був безпосередньо доступний ззовні.

Налаштування переадресації портів

Після входу в панель адміністрування роутера TP-Link Archer C64 я налаштував переадресацію портів NAT, яка дозволяє перенаправляти зовнішній трафік з інтернету на потрібний порт на локальному комп'ютері.

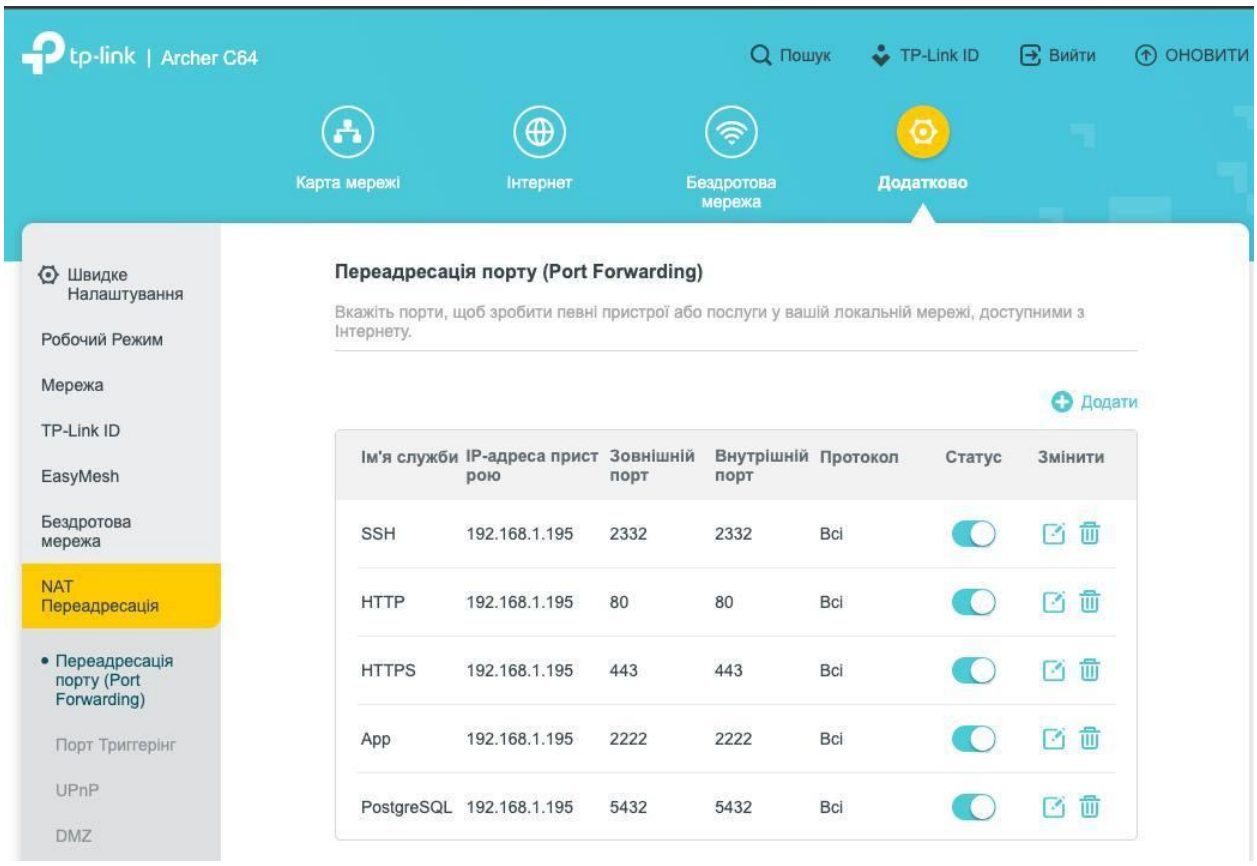

Рисунок 3.8 – Настройки роутера Archer C64.

Переадресація SSH-портів

Налаштовано так, що всі зовнішні запити на порт 2332 автоматично перенаправляються на локальний порт 2332 на пристрої з IP-адресою 192.168.1.195.

Це дає можливість підключитися до сервера з будь-якої точки світу через SSH-з'єднання. Такий підхід дозволяє віддалено адмініструвати сервер, виконувати команди, оновлювати контейнери тощо.

Порт SSH встановлений на 2332 - нестандартний, щоб зменшити кількість бот-атак.

Переадресація HTTP-портів

Для підтримки звичайного незашифрованого HTTP-з'єднання дозволено перенаправляти зовнішні запити на порт 80 на внутрішній порт 80 на сервері 192.168.1.195.

Це може бути корисно для початкового тестування сервісу або автоматичного перенаправлення на HTTPS через Nginx.

Перенаправлення на порт HTTPS

Весь трафік на порт 443 (зовнішній) перенаправляється на внутрішній порт 443 сервера.

Це підтримує безпечні HTTPS-з'єднання - завдяки використанню SSL-сертифікатів (через Cloudflare або Let's Encrypt), сервер приймає трафік через зашифрований протокол, що є важливим для безпеки користувачів.

Перенаправлення портів додатків (Spring Boot)

Порт 2222 виділено окремо для роботи програми Spring Boot, яка обслуговує REST API.

Всі запити, адресовані на порт 2222 з інтернету, перенаправляються на внутрішній порт 2222 сервера. Цей порт також служить внутрішнім інтерфейсом, для якого Nginx виступає проксі-сервером при обробці запитів до домену babakdriver.xyz.

Перенаправлення портів PostgreSQL

Порт 5432 відкритий для зовнішнього підключення до бази даних PostgreSQL, яка також розгорнута на сервері.

Це дозволяє легко підключатися до бази даних за допомогою графічного клієнта, такого як pgAdmin, з будь-якого пристрою - переглядати дані, редагувати таблиці, створювати запити або резервні копії.

Загальне призначення параметрів

Ці п'ять переданих портів дозволяють

- адміністрування (через SSH)
- Доступ до
- підключення до бази даних
- можливість запускати ручні тести або визначати HTTP-параметри.

Такий підхід дозволяє перетворити звичайний домашній комп'ютер на повноцінний віддалений сервер, що обслуговує мобільний додаток, базу даних та API через захищений домен.

3.5 Реалізація мобільного застосунку на платформі Android

Реалізувавши API у Spring Boot та розгорнувши його на власному сервері, наступним кроком стала розробка Android-додатку, що забезпечує зручний доступ до функціоналу системи аналізу погоди в дорозі. Додаток було реалізовано з нуля з використанням сучасного технологічного стеку та бібліотек.

Архітектура: MVVM

MVVM (Model-View-ViewModel) - це популярна архітектурна модель, яка використовується для створення добре структурованих, масштабованих і тестованих Android-додатків. Її основна ідея полягає в тому, щоб відокремити користувальницький інтерфейс від логіки та управління даними.

Мобільний додаток реалізує архітектуру MVVM (Model-View-ViewModel):

- Model - робота з даними, API, репозиторіями.
- ViewModel - бізнес-логіка, управління станами, асинхронна логіка.
- View - екрани, елементи інтерфейсу, Jetpack Compose.

Які бібліотеки використовуються в проєкті

Jetpack Compose - це сучасний декларативний фреймворк для створення інтерфейсу.

Мінімум коду, простий в підтримці, чудово підходить для анімації, адаптивний дизайн.

Retrofit - основний HTTP-клієнт для виклику REST API.

Простий синтаксис, інтеграція з Gson, підтримка підпрограм.

OkHttp3 - підключається як транспорт до Retrofit.

Підтримка перехоплювачів, логування, таймаутів.

Google Maps SDK і Places API - для інтерактивного вибору місця, побудови маршруту та відображення погодних маркерів.

Офіційна, надійна та масштабована бібліотека карт з підтримкою всіх необхідних функцій.

Kotlin Coroutines - для обробки асинхронних запитів.

Спрощує роботу з потоками, дозволяє мати чистий код у ViewModel.

Структура проекту

Рисунок 3.8 – Структура мобільного проєкта.

Проект має добре організовану, багаторівневу структуру, що відповідає принципам чистої архітектури.

com.radchuk.babakdriver – основний пакет застосунка

Package - app

- App.kt — стартова точка додатку, ініціалізація навігації.

Package - model

Package - consts

- зберігаються ключі для SavedState, назви екранів.

Package - request

- DTO-класи для запиту: JourneyRequest, CityABC.

Package - response

- відповіді від сервера (JourneyResponse, WeatherAnalysis, MapResponse).

Package - map

- LatLng і пов'язані координатні структури.

Package - screens

Package - inputlocation

- AddressInputScreen.kt — ручне введення точки маршруту.
- MapPickerScreen.kt — вибір точки на мапі через інтерфейс.

Package - info

- InfoScreen.kt - екран статистики та результатів.

Package - view - компоненти інтерфейсу:

- WeatherCard.kt, WeatherInfoItem.kt — вивід погоди;
- MapView.kt, Markers.kt — карта й погодні маркери;
- BottomBar.kt, TopBar.kt — навігаційні елементи;
- RouteSelection.kt — екран вибору маршруту.

Package – utils

- MapUtils.kt — допоміжні функції для карти (маркерів, центрів, маршрутів).

Package - permissions

- LocationPermissionManager.kt — логіка запиту дозволу на геолокацію.

Package - journey

- Реалізує основний сценарій поїздки та інтеграцію з API.

Package - components - UI-компоненти:

- AverageSpeedSliderElement.kt
- FrequencySliderElement.kt
- DatePickerElement.kt
- TimePickerElement.kt
- InputLocation.kt

Всі ці компоненти адаптивні, виконані через Jetpack Compose.

Package - data

- JourneyData.kt, IntermediatePoint.kt, JourneyUIState.kt — внутрішні структури з даними маршруту, станом UI.

Package - network

- ApiService.kt — Retrofit API інтерфейс.
- NetworkModule.kt — конфігурація Retrofit + OkHttp.

Package - repository

- JourneyRepository.kt – клас для отримування даних з запиту

Package - viewmodels

- JourneyViewModel.kt — основна ViewModel:
  - керує всією логікою маршруту;
  - зберігає стан;
  - викликає API;
  - обробляє відповіді.
- JourneyRequestBuilder.kt - побудова JourneyRequest із UI-даних.
- JourneyViewModelFactory.kt - фабрика.
- JourneyScreen.kt - головний екран побудови маршруту.

Package - settings

- SettingsScreen.kt — налаштування користувача (можливо, вкл/викл погоди, мови, вигляду).

Package - ui.theme

- Файл MainActivity.kt — вхідна точка Android-застосунку, хост Composable-екранів, навігація.

4 ВПРОВАДЖЕННЯ СИСТЕМИ

4.1 Тестування системи

Після завершення розробки мобільного додатку та сервера важливим кроком стало тестування всієї системи в умовах, наближених до реальних. Метою цих тестів було

Перевірка стабільності та продуктивності мобільного клієнта.

Перевірка обробки запитів сервером.

Переконатися, що розрахунки маршруту та погодних даних є правильними.

Перевірити роботу системи у змінних умовах: різні дані, кілька одночасних користувачів, затримки в мережі тощо.

Типи проведених тестів

1. Функціональні тести

Мета: переконатися, що система працює відповідно до вимог користувача.

Що тестувалося:

- Введення початкової та кінцевої точок маршруту.
- Додавання проміжних зупинок.
- Вибір дати та часу поїздки.
- Зміна середньої швидкості.
- Встановлення інтервалу перевірки погоди (кожні 10-50 км).
- Отримання коректної відповіді від сервера: маршрут + погодні дані.
- Відображення інформації на карті.
- Відображення статистики погоди у вигляді таблиці.

Результат: всі функції працюють стабільно, як і очікувалося.

2. Інтеграційні тести

Мета: протестувати взаємодію між клієнтом і сервером.

Було протестовано :

- Підключення до сервера за допомогою Retrofit.
- Доступ до сервера через HTTPS домен babakdriver.xyz.
- Відповіді на запити від Android додатку (код 200, тіло JSON).
- Коректний аналіз відповіді в JourneyResponse.

Тестові сценарії:

- Запит без зупинок.
- Запит з декількома зупинками.
- Запит з майбутньою та поточною датою.
- Зміна інтервалу погоди (10, 20, 50 км).

Результат: Інтеграція Android ↔ Spring Boot працює надійно. Не було зафіксовано жодних JSON помилок або збоїв.

3. Стрес-тест

Мета: перевірити поведінку системи під навантаженням.

Як тестувалося :

- Запущено 4 емулятори Android.
- Одночасно було запущено три паралельні запити на побудову маршруту.
- Моніторинг системи здійснювався через htop.

Результати:

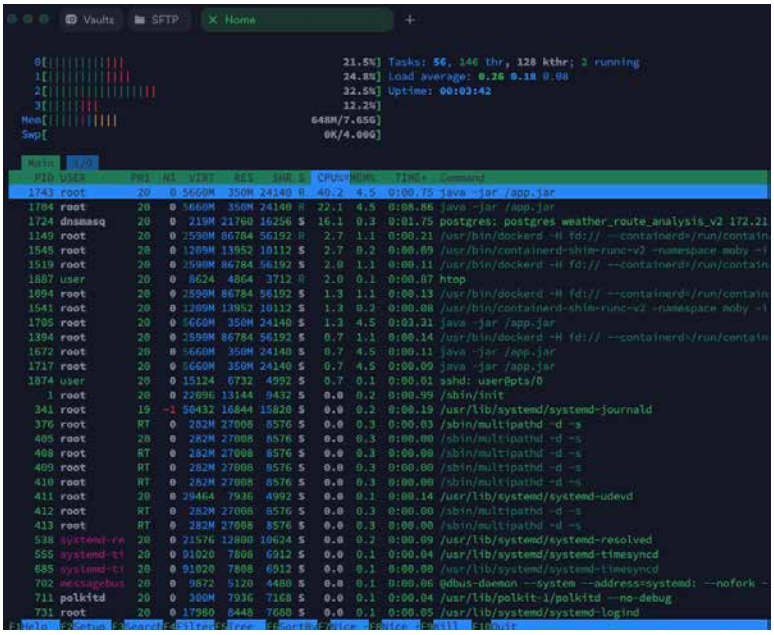

Рисунок 4.1 – Нагрузка сервера під час тестів

PID	CPU%	MEM%	Команда
1743	40.2%	4.5%	java -jar /app.jar
1704	22.1%	4.5%	java -jar /app.jar
1705	4.0%	4.5%	java -jar /app.jar
1717	4.5%	4.5%	java -jar /app.jar

1723	3.0%	1.0%	postgres: postgres weather_route_analysis_v2

Опис навантаження :

- Spring-додаток /app.jar (PID 1743, 1704, 1705, 1717):
- Загальне навантаження на процесор: ~71%.
- Загальне використання оперативної пам'яті: ~18%.
- 4 активних екземпляри є основним джерелом навантаження на систему(одночасно було під’єднано 4 клієнта).

PostgreSQL (PID 1723):

- CPU: ~3%.
- ОПЕРАТИВНА ПАМ'ЯТЬ: ~1%.

Навантаження низьке, що відповідає нормальній роботі бази даних.

Система працює стабільно, основне навантаження створює Spring-додаток.

Час відгуку API: в середньому 350-450 мс

Сервер не висів, не зависав і не видавав помилок.

4. Тестування користувацького інтерфейсу

Мета: перевірити зручність і точність інтерфейсу.

Що тестувалося:

- Адаптація до різних екранів.
- Відображення української мови.
- Робота курсорів (швидкість, інтервал).
- Вибір дати та часу.
- Відображення маршруту на карті Google.
- Відображення погодних маркерів.
- Відображення детальної статистики у вигляді таблиці.

Результат: UX практичний, UI читабельний, компоненти працюють.

5. Валідація даних

Мета: перевірити, як система реагує на неправильні або неповні дані.

Приклади перевірок:

- Порожні поля маршруту.
- Не вказана швидкість.
- Більш рання дата.
- Надмірна кількість пунктів (>50).

Очікувані результати:

- Користувачеві буде показано повідомлення про помилку.
- Запит не відправлено.
- ViewModel блокує дію.

Результат: Валідація працює, користувач не може відправити некоректний запит.

6. Тестування дозволів

Мета: перевірити, чи правильно працюють авторизації геолокації в системі.

Компонент: LocationPermissionManager.kt

Тест:

- Початкова відмова користувача.
- Новий запит.
- Дозвіл надано - коректне відображення карти з поточним положенням.

Результат: Логіка дозволів працює відповідно до рекомендацій Android.

Всі основні етапи тестування показали, що система працює стабільно, швидко і надійно, як в локальних умовах, так і при віддаленому підключенні. Були протестовані всі основні сценарії, включаючи варіанти з навантаженням, некоректним введенням і відсутністю дозволів. Завдяки використанню MVVM, Jetpack Compose, Retrofit, Docker та Spring Boot, система ідеально підходить для розгортання у виробництві та подальшого масштабування.

4.2 Апаратні та технічні ресурси

Розроблена система складається з двох основних елементів:

- Клієнтська частина - Android-додаток, що запускається на мобільному пристрої користувача.
- Серверна частина - це веб-сервер з API, базою даних та логікою обробки маршрутів та погодних умов.

Для стабільної роботи системи необхідно враховувати вимоги до апаратного та програмного забезпечення.

Клієнтська частина (додаток для Android)

Мобільний додаток реалізований з використанням Jetpack Compose, Google Maps, Places API, тому вимагає певних ресурсів пристрою.

Параметр	**Мінімальне значення**	**Рекомендоване значення**
Операційна система	Android 8.0 (API 26)	Android 12 (API 31) або вище

Оперативна пам'ять	2 ГБ	4 ГБ або більше
Процесор (CPU)	ARM Cortex-A53 або подібний	ARM Cortex-A75 або Snapdragon 7xx
Інтернет	3G/4G LTE	Wi-Fi або 4G LTE
Дисплей	5« HD (720p)	6» FHD (1080p)
Сервіси Google	Повна підтримка сервісів Google	Повна підтримка сервісів Google

Серверна частина (Spring Boot + PostgreSQL)

Для зручності обслуговування сервер розгорнуто на домашньому ПК Lenovo ThinkCentre, який діє як повноцінне хостингове рішення (подібно до VPS).

Характеристики реального сервера:

1. Процесор: Intel i5-7500T (4 ядра, до 3,3 ГГц)
2. Оперативна пам’ять: 8 ГБ
3. Пристрій зберігання даних: 128 ГБ SSD
4. Платформа: Ubuntu Server 24.04.2 LTS
5. Мережеве підключення: Ethernet + біла IP-адреса

Мінімальна та рекомендована конфігурація сервера :

Параметр	Мінімальне значення	Рекомендоване значення
Операційна система	Ubuntu Server 20.04 LTS	Ubuntu Server 22.04 LTS

Процесор (CPU)	2 ядра по 2,0 ГГц	4 ядра по 2,5-3,0 ГГц
Оперативна пам'ять	4 ГБ	8 ГБ або більше
Жорсткий диск	SSD/HDD 50 ГБ	SSD 200+ ГБ

Використовуване програмне забезпечення

На сервері :

- Java 17 (OpenJDK) - для запуску Spring Boot додатку.
- Spring Boot - фреймворк для створення REST API.
- PostgreSQL 15 - база даних.
- Docker / Docker Compose - для ізоляції сервісів.
- Nginx - для HTTPS і проксі запитів.
- Cloudflare DNS - для SSL.
- htop - моніторинг навантаження в реальному часі.
- pgAdmin - віддалене управління базами даних.

Доступно для Android:

- Jetpack Compose, Navigation
- Retrofit + OkHttp
- Google Maps SDK
- Places API
- Модель перегляду + StateFlow
- Матеріал 3 компонентів інтерфейсу користувача

Для повноцінної роботи системи потрібно лише:

- Будь-який Android пристрій з підтримкою Google Maps ;
- ПК або сервер з мінімальними ресурсами (навіть домашній комп'ютер) ;
- стабільне інтернет-з'єднання з білою IP-адресою або DNS-проксі;

- знання Docker та Linux для керування сервером.

Завдяки простоті архітектури, вся система може працювати навіть на локальному сервері без втрати продуктивності.

4.3 Опис роботи програми

Програмне забезпечення системи аналізу погоди BabakDriver- це мобільний додаток для Android, який дозволяє користувачеві побудувати маршрут подорожі, вказати швидкість, зупинки, дату відправлення та отримати аналіз погоди в точках маршруту. Додаток працює в інтеграції з власним сервером Spring Boot, який розгортається через Docker в домашній мережі.

Загальна логіка роботи

1. Користувач запускає Android додаток.
2. Вводить параметри для поїздки.
3. Запит відправляється на сервер по HTTPS.
4. Сервер обробляє запит:
    - визначає маршрут;
    - розбиває його на контрольні пункти;
    - отримує прогноз погоди через OpenWeatherMap API;
    - оцінює погодні ризики.
5. Сервер надсилає відповідь клієнту.
6. Клієнт відображає

- карту з маршрутом і погодними маркерами;
- таблицю погодних даних та зведену статистику.

Введення параметрів маршруту

На головному екрані користувач вводить

1. Початковий пункт (автозаповнення або вибір на карті).
2. Проміжні зупинки (із зазначенням тривалості перебування).
3. Кінцевий пункт маршруту.
4. Дату і час початку подорожі.
5. Середня швидкість (курсор).
6. Інтервал перевірки погоди (наприклад, кожні 30 км).

Реалізація в таких файлах:

- InputLocation.kt, AddressInputScreen.kt, FrequencySliderElement.kt, TimePickerElement.kt.
- JourneyViewModel.kt збирає дані та генерує JourneyRequest.

Відправлення запиту на сервер

При натисканні на кнопку Почати поїздку :

- Проводить валідацію даних.
- Додаток генерує POST-запит типу /api/analysis/trip.
- Використовується Retrofit + OkHttp + підпрограми.
- Запит шифрується і відправляється через домен https://babakdriver.xyz.

Дані запиту включають в себе :

- startPoint, endPoint, зупинки
- avgSpeed, weatherCheckInterval, startTime

Отримання та аналіз відповіді

Додаток отримує відповідь у форматі JSON через Retrofit.

Відповідь містить

- Маршрут на карті (MapResponse)
- Таблицю метеостанцій (WeatherMarker)
- Детальні значення погоди (WeatherAnalysis)
- Зведену таблицю (середні/мінімальні/максимальні значення).

Аналіз реалізовано у форматі :

- JourneyResponse.kt
- WeatherAnalysis.kt

Відображення карти з погодними маркерами

Координати та погодні дані передаються до :

- MapView.kt - відображення маршруту.
- Markers.kt - малювання метеорологічних точок.

Кожен маркер відображає

- Час прибуття в точку.
- Температура, хмарність, вітер.
- Статус погоди.

Якщо ризик високий, маркер може бути підсвічений червоним кольором. Жовтий колір показується про можливі ризики. Зелений колір маркета безпечна погода.

Відображення таблиці погодних умов

Таблиця відображається під мапою:

Статистика за весь маршрут :

- Температура
- Вітер

- Вологість
- Напрямок вітру
- Хмарність
- Видимість.

Таблиця містить середні, мінімальні та максимальні значення.

Компоненти:

- WeatherCard.kt, WeatherInfoItem.kt, InfoScreen.kt

Мобільний додаток повністю автоматизує процес планування поїздки з урахуванням погодних умов. Все, що потрібно зробити користувачеві - це ввести маршрут. Додаток

- будує маршрут
- враховує зупинки та швидкість руху
- отримує прогноз погоди на час прибуття в кожну точку
- аналізує потенційні погодні ризики,
- відображаючи все це на карті та в таблиці.

Завдяки сучасній архітектурі (MVVM), Jetpack Compose та інтеграції з сервером за допомогою Retrofit, система працює швидко, стабільно та зручно.

4.4 Графічне представлення додатку

Щоб краще пояснити, як працює система, варто представити її візуальне представлення, яке відображає не тільки інтерфейс додатку, але й логіку взаємодії користувача з функціоналом - від введення даних до аналізу погоди в точках маршруту.

Інтерфейс користувача

На зображеннях нижче показані основні екрани додатку:

- Екран введення даних маршруту

Користувач вводить пункт відправлення, зупинки, пункт прибуття, обирає час/дату відправлення, середню швидкість та інтервал перевірки погоди (20, 30, 50 км).

- Автодоповнення адреси з інтеграцією Google Places API

  Додаток дозволяє легко знайти і вибрати точку на карті або за допомогою текстового пошуку.

- Карта маршруту з погодними маркерами

  На маршруті розміщені три кольорові маркери:

  - Зелений - нормальні погодні умови
  - Жовтий - середній ризик
  - Червоний - потенційно небезпечні погодні умови (наприклад, дощ, сильний вітер)

- Вікно з інформацією про погоду

  При натисканні на маркер з'являється спливаюче вікно з інформацією:

  - температура
  - хмарність
  - швидкість вітру
  - вологість
  - напрямок вітру
  - пориви вітру
  - прогноз видимості

- Екран статистики погоди

  Відображає середнє, мінімальне та максимальне значення погодних умов на всьому маршруті.

  Також відображаються агреговані коефіцієнти:

  - Безпека: рівень безпеки маршруту (наприклад, 40%)
  - Комфорт: рівень комфортності погодних умов (наприклад, 50%)

Зображення програми

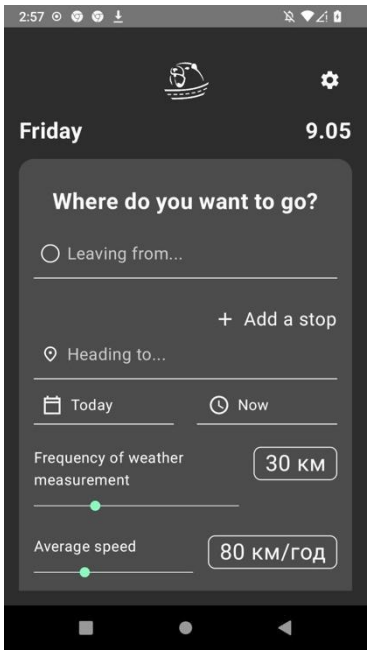

Рисунок 4.2 – Головний екран

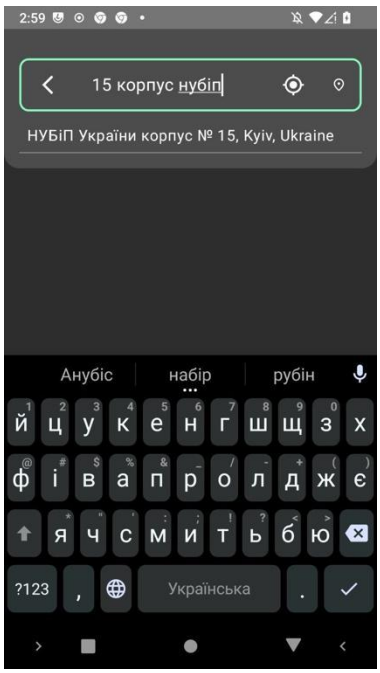

Рисунок 4.3 – Введення місце за допомогою підказок

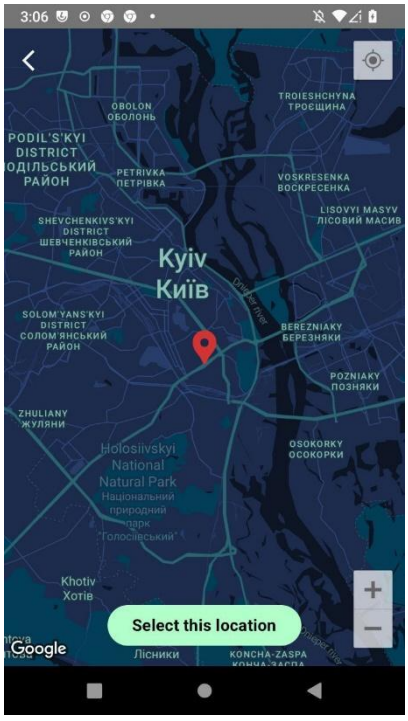

Рисунок 4.4 – Введення місця за допомогою знаходження на карті

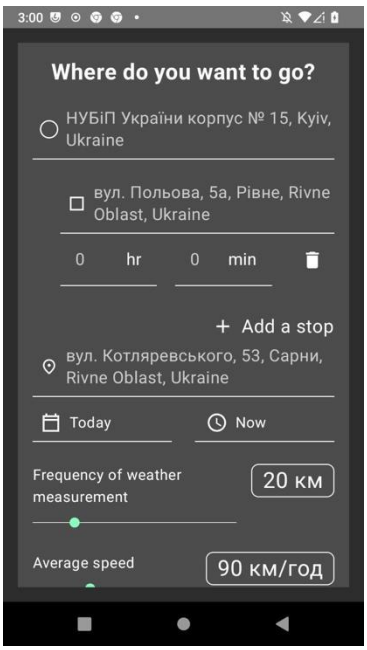

Рисунок 4.5 – Заповнення всіх даних для подорожі

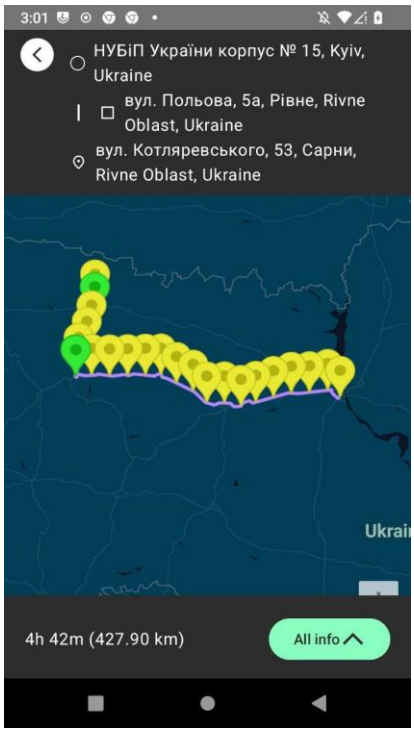

Рисунок 4.6 – Результат подорожі

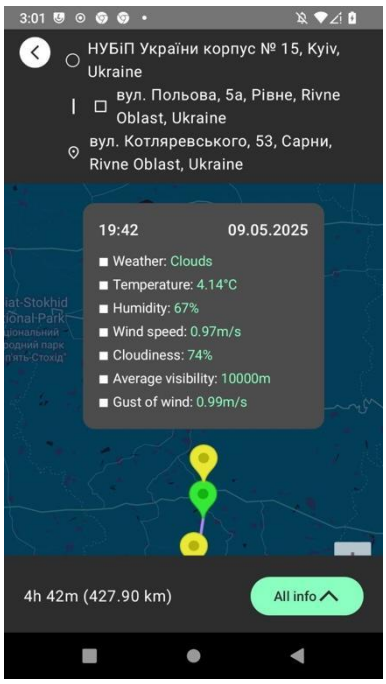

Рисунок 4.7 – Перегляд погоди за допомою маркера

Рисунок 4.8 – Вілізація даних у табличному вигляді

ВИСНОВКИ

У процесі виконання дипломного проєкту на тему «Програмне забезпечення системи аналізу погоди на маршруті» було повністю реалізовано функціональний комплекс, який дозволяє користувачеві здійснювати планування подорожей з урахуванням погодних умов у реальному часі. Система є практичним прикладом сучасного застосування інформаційних технологій у повсякденному житті, а також демонструє ефективність архітектури клієнт-сервер у мобільних рішеннях.

Головною метою проєкту було створити зручний Android-застосунок, що дозволяє вводити маршрут подорожі, включаючи початкову й кінцеву точки, проміжні зупинки, середню швидкість руху, інтервал перевірки погоди та дату/час виїзду. Основна ідея полягала в тому, щоб надати користувачеві максимально релевантну й актуальну інформацію про погодні умови в тих точках маршруту, де він, згідно з розрахунками, перебуватиме у певний момент часу. Таким чином, система забезпечує не лише зручність, а й підвищення рівня безпеки поїздок, зважаючи на погодні ризики.

Для реалізації цього функціоналу було створено мобільний застосунок з інтерфейсом на основі Jetpack Compose. Завдяки використанню архітектурного

патерну MVVM (Model-View-ViewModel), додаток має чітку структуровану логіку, забезпечує збереження станів між екранами, асинхронну обробку даних і швидку взаємодію з API. Взаємодія з картами реалізована через Google Maps SDK та Places API, що забезпечує точність геолокацій, автозаповнення адрес і зручний вибір місцеположення.

Зі сторони серверу було реалізовано REST API на базі Spring Boot та Kotlin, що обробляє запити від клієнта, розраховує прогнозований час прибуття до контрольних точок, отримує погодні дані з OpenWeatherMap API, аналізує погодні ризики та формує відповідь у форматі JSON. Серверна частина розгорнута у Docker-контейнерах і включає окремий контейнер з PostgreSQL для зберігання та обробки даних. Для маршрутизації трафіку та безпечного з'єднання було використано Nginx разом із Cloudflare і власним доменом. Це рішення дозволяє не лише забезпечити безпеку та стабільність роботи API, а й підтримувати гнучке масштабування у майбутньому.

Особливу увагу в роботі було приділено практичному досвіду розгортання серверної інфраструктури в умовах домашньої мережі. Замість готового VPS-рішення було обрано шлях побудови власного сервера на базі ПК з Ubuntu Server. Це дозволило не лише глибше зануритися в теми системного адміністрування, Docker-оркестрації, мережевих протоколів і NAT-переадресації, але й реалізувати повноцінну production-систему, доступну з будь-якої точки світу.

У ході тестування система продемонструвала стабільну роботу навіть при одночасному використанні декількома клієнтами. Завдяки використанню асинхронної обробки даних і правильній архітектурі, час відповіді сервера залишався прийнятним, а завантаженість системи не перевищувала допустимі межі.

Кінцевий результат — це високоякісний програмний продукт, що має практичну цінність. Система дозволяє:

– будувати оптимальний маршрут;

– аналізувати погодні умови на всьому шляху;

– наочно бачити зони ризику;

– приймати обґрунтовані рішення перед поїздкою.

У межах проєкту було реалізовано повний цикл розробки: від аналізу предметної області, проєктування архітектури, створення UML-діаграм і моделювання бази даних до реалізації мобільного клієнта, серверної частини, впровадження API та розгортання власного хостингу. Це свідчить не лише про набуті технічні навички, а й про здатність самостійно організовувати та реалізовувати складні проєкти повного циклу.

Отже, створене програмне забезпечення може бути використано як практичний інструмент для щоденного планування поїздок або як базовий приклад для подальшого вдосконалення й розширення у комерційних або дослідницьких проєктах.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Методичні вказівки до виконання лабораторних робіт з дисципліни «Проєктний практикум» / уклад.: к.т.н. Голуб Б. Л., асист. Лендєл М. І. – К. : НУБіП України, 2023. – 45 с.
2. Простий посібник зі схем UML і моделювання баз даних [Електронний ресурс]. – Офіційний сайт Microsoft. – Режим доступу: http://surl.li/pgfnj
3. Життєвий цикл Agile розробки програмного забезпечення [Електронний ресурс]. – Офіційний сайт компанії EPAM / Training Portal. – Режим доступу: https://training.epam.com/en/blog/581
4. Методичні вказівки до виконання лабораторних робіт з дисципліни «Методи об'єктно-орієнтованого проектування програмних систем» для студентів ОС «Бакалавр», що навчаються за спеціальністю «Інженерія програмного забезпечення» / уклад.: к.т.н. Голуб Б. Л., асист. Лендєл М. І. – К. : НУБіП України, 2023. – 77 с.
5. 1. Буч Г. UML. Руководство пользователя / Буч Г., Рамбо Дж., Джекобсон А. - М. : ДМК Пресс, 2003. - 432 с.: ил.
6. Мартін Фаулер. UML. Основи та практики. – Київ: Диалектика, 2020. – 320 с.

7. КрэгЛарман. Применение UML 2.0 шаблонов проектирования = Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design and Iterative Development. — 3-е изд. — М.: Вильямс, 2006. — 736 с.
8. Object Management Group (OMG): UML 2.5 Specification [Електронний ресурс] – Режим доступу: https://www.omg.org/spec/UML/2.5.1
9. Граді Буч, Джеймс Рамбо, Івар Якобсон. UML. Керівництво користувача. — М.: Вид-во «Вільямс», 2006. — 512 с.
10. Документація Spring Boot [Електронний ресурс] – Режим доступу: https://spring.io/projects/spring-boot
11. Kotlin Language Documentation [Електронний ресурс] – Режим доступу: https://kotlinlang.org/docs/home.html
12. Jetpack Compose Documentation – Android Developers [Електронний ресурс] – Режим доступу: https://developer.android.com/jetpack/compose
13. PostgreSQL Documentation [Електронний ресурс] – Режим доступу: https://www.postgresql.org/docs/
14. OpenWeatherMap API Documentation [Електронний ресурс] – Режим доступу: https://openweathermap.org/api
15. Google Maps Platform Documentation [Електронний ресурс] – Режим доступу: https://developers.google.com/maps/documentation
16. Cloudflare Documentation [Електронний ресурс] – Режим доступу: https://developers.cloudflare.com/
17. Docker Documentation [Електронний ресурс] – Режим доступу: https://docs.docker.com/
18. MVVM Design Pattern in Android [Електронний ресурс] – Режим доступу: https://developer.android.com/topic/architecture
19. Retrofit: Type-safe HTTP client for Android and Java [Електронний ресурс] – Режим доступу: https://square.github.io/retrofit/
20. OkHttp — Square Open Source [Електронний ресурс] – Режим доступу: https://square.github.io/okhttp/

21. Android Material Design Guidelines [Електронний ресурс] – Режим доступу: https://m3.material.io/
22. Figma Design Tool – офіційний сайт [Електронний ресурс] – Режим доступу: https://www.figma.com/
23. Cloudflare SSL and DNS Routing [Електронний ресурс] – Режим доступу: https://developers.cloudflare.com/ssl/edge-certificates/universal-ssl/
24. Nginx Documentation [Електронний ресурс] – Режим доступу: https://nginx.org/en/docs/
25. GitHub – Spring Boot Kotlin Example Projects [Електронний ресурс] – Режим доступу: https://github.com/JetBrains/kotlin-examples
26. ChatGPT, штучний інтелект [Електронний ресурс] – Режим доступу: https://chat.openai.com/
27. Документація Android SDK [Електронний ресурс] – Режим доступу: https://developer.android.com/studio
28. PostgreSQL GUI tool — pgAdmin [Електронний ресурс] – Режим доступу: https://www.pgadmin.org/
29. NGINX + Docker Reverse Proxy Tutorial [Електронний ресурс] – Режим доступу: https://www.digitalocean.com/community/tutorials/how-to-set-up-nginx-reverse-proxy-with-docker-containers
30. Android Open Source Project — AOSP [Електронний ресурс] – Режим доступу: https://source.android.com/

ДОДАТОК А

База даних

Київ-2025


```
-- Створення бази даних

CREATE DATABASE weather_route_analysis;

-- Видалення таблиць, якщо вони вже існують

DROP TABLE IF EXISTS RouteStops;

DROP TABLE IF EXISTS Stop;

DROP TABLE IF EXISTS CheckpointWeather;

DROP TABLE IF EXISTS Weather;

DROP TABLE IF EXISTS Checkpoint;

DROP TABLE IF EXISTS RouteParams;

DROP TABLE IF EXISTS UserTrip;

DROP TABLE IF EXISTS Trip;
```

```
DROP TABLE IF EXISTS "User";

DROP TABLE IF EXISTS Role;

-- Створення таблиці Role

CREATE TABLE Role (

    role_id SERIAL PRIMARY KEY,

    role_name VARCHAR(50) NOT NULL

);
```



```
-- Створення таблиці User

CREATE TABLE "User" (

    user_id SERIAL PRIMARY KEY,

    name VARCHAR(100) NOT NULL,

    email VARCHAR(100) UNIQUE NOT NULL,

    role_id INT REFERENCES Role(role_id)

);

-- Створення таблиці Trip

CREATE TABLE Trip (

    route_id SERIAL PRIMARY KEY,

    start_point VARCHAR(100) NOT NULL,
```

```
    end_point VARCHAR(100) NOT NULL
);
-- Створення таблиці UserTrip
CREATE TABLE UserTrip (
    user_id INT REFERENCES "User"(user_id),
    route_id INT REFERENCES Trip(route_id),
    PRIMARY KEY (user_id, route_id));
```



```
-- Створення таблиці RouteParams
CREATE TABLE RouteParams (
    params_id SERIAL PRIMARY KEY,
    avg_speed DOUBLE PRECISION NOT NULL,
    check_interval INT NOT NULL,
    start_datetime TIMESTAMP NOT NULL,
    route_id INT UNIQUE REFERENCES Trip(route_id)
);
-- Створення таблиці Checkpoint
CREATE TABLE Checkpoint (
    marker_id SERIAL PRIMARY KEY,
```

```
    arrival_time TIMESTAMP NOT NULL,
    route_id INT REFERENCES Trip(route_id)
);
-- Створення таблиці Weather
CREATE TABLE Weather (
    weather_id SERIAL PRIMARY KEY,
    state VARCHAR(50) NOT NULL,
    temperature DOUBLE PRECISION NOT NULL,
description VARCHAR(255));
```



```
-- Створення таблиці CheckpointWeather
CREATE TABLE CheckpointWeather (
    marker_id INT REFERENCES Checkpoint(marker_id),
    weather_id INT REFERENCES Weather(weather_id),
    PRIMARY KEY (marker_id, weather
_id)
);
-- Створення таблиці Stop
CREATE TABLE Stop (
    stop_id SERIAL PRIMARY KEY,
    address VARCHAR(255) NOT NULL,
```

```
    waiting_time INT,
    lat DOUBLE PRECISION NOT NULL,
    lng DOUBLE PRECISION NOT NULL
);
-- Створення таблиці RouteStops
CREATE TABLE RouteStops (
    stop_id INT REFERENCES Stop(stop_id),
    route_id INT REFERENCES Trip(route_id),
    PRIMARY KEY (stop_id, route_id));
```

ДОДАТОК Б

Код програми Android Project

Київ-2025


```
class MainActivity : ComponentActivity() {
    private val locationPermissionRequest =
        registerForActivityResult(ActivityResultContracts.RequestPermission()) {
granted ->
            if (granted) {

                permissionManager.checkLocationPermission()
            } else {

            }
        }

    private lateinit var permissionManager: LocationPermissionManager

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        permissionManager = LocationPermissionManager(
            context = this,
            permissionRequest = locationPermissionRequest,
            onPermissionGranted = {
                setContent {
                    BabakDriverComposeUITheme {
                        AppNavigator()
                    }
                }
            },
            onPermissionDenied = {
            }
        )


        permissionManager.checkLocationPermission()
    }
```

```
}

@Composable
fun AppNavigator() {
    val navController = rememberNavController()
    val journeyViewModel: JourneyViewModel = viewModel(
        factory = JourneyViewModelFactory(repository =
JourneyRepository(NetworkModule.apiService))
    )
    NavHost(navController = navController, startDestination =
ScreenRoutes.JOURNEY_SCREEN) {
        composable(ScreenRoutes.JOURNEY_SCREEN) {
            JourneyScreen(navController = navController, viewModel =
journeyViewModel)
        }
        composable(ScreenRoutes.ADDRESS_INPUT_SCREEN) { backStackEntry ->
            val addressType = backStackEntry.arguments?.getString("type") ?:
"unknown"
            AddressInputScreen(
                onBackPressed = { navController.popBackStack() },
                onAddressSelected = { address ->

navController.previousBackStackEntry?.savedStateHandle?.set(addressType,
address)
                    navController.popBackStack()
                },
                onMapPick = {
                    navController.navigate("mapPicker")
```



```
                },
                navController
            )
        }
        composable(ScreenRoutes.INFO_SCREEN) {
            InfoScreen(navController = navController, viewModel =
journeyViewModel)
        }
        composable(ScreenRoutes.SETTINGS_SCREEN) {
            SettingsScreen(navController = navController)
        }
        composable("mapPicker") {
            MapPickerScreen(
                onLocationSelected = { latLng ->
                    val chosenLocation =
"${latLng.latitude},${latLng.longitude}"

                    navController.previousBackStackEntry
                        ?.savedStateHandle
                        ?.set("mapLocation", chosenLocation)

                    navController.popBackStack()
                },
                onBackPressed = {
                    navController.popBackStack()
                }
            )
        }
    }

}
```

```
@Composable
fun JourneyScreen(
    navController: NavController,
    viewModel: JourneyViewModel
) {

    val uiState by viewModel.uiState.collectAsState()

    val leavingFrom =
viewModel.leavingFrom.observeAsState(Constants.LEAVING_FROM)
    val headingTo = viewModel.headingTo.observeAsState(Constants.HEADING_TO)
    val intermediatePoints by
viewModel.intermediatePoints.observeAsState(emptyList())

    val selectedDate by
viewModel.selectedDate.observeAsState(Constants.SELECTED_DATE)
    val selectedTime by
viewModel.selectedTime.observeAsState(Constants.SELECTED_TIME)

    val sliderValueFrequencyElement by
viewModel.sliderValueFrequencyElement.observeAsState(30f)
    val sliderValueAverangeSpeedElement by
viewModel.sliderValueAverangeSpeedElement.observeAsState(80f)
    LaunchedEffect(Unit) {

        navController.currentBackStackEntry?.savedStateHandle?.let {
savedStateHandle ->

savedStateHandle.getLiveData<String>(SavedStateKeys.LEAVING_FROM_ADDRESS).observ
eForever { address ->
```



```
                viewModel.updateLeavingFrom(address)
            }


savedStateHandle.getLiveData<String>(SavedStateKeys.HEADING_TO_ADDRESS).observeF
orever { address ->
                viewModel.updateHeadingTo(address)
            }

            intermediatePoints.forEachIndexed { index, _ ->

savedStateHandle.getLiveData<String>("${SavedStateKeys.INTERMEDIATE_POINT}$index
")
                    .observeForever { address ->
                        viewModel.updateIntermediatePoint(index, address)
                    }
            }
        }
    }


    if(uiState != JourneyUiState.Idle) {
        CircularProgressIndicator(
            modifier = Modifier
                .background(Color.White)
                .fillMaxSize()
                .wrapContentSize(Alignment.Center)
        )
    }else {
```

```
        Column(
            modifier = Modifier
                .fillMaxSize()
                .background(colorResource(id = R.color.back))
                .padding(WindowInsets.systemBars.asPaddingValues())
                .padding(16.dp)
                .verticalScroll(rememberScrollState()),
            horizontalAlignment = Alignment.CenterHorizontally
        ) {
            LogoAndSettingsSection(navController)

            Spacer(modifier = Modifier.height(16.dp))

            if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
                DateAndTimeSection()
            }

            Spacer(modifier = Modifier.height(16.dp))

            JourneyDetailsCard(
                leavingFrom = leavingFrom.value,
                headingTo = headingTo.value,
                intermediatePoints = intermediatePoints,
                navController = navController,
                selectedDate = selectedDate,
                selectedTime = selectedTime,
                sliderValueFrequencyElement = sliderValueFrequencyElement,
                sliderValueAverangeSpeedElement =
sliderValueAverangeSpeedElement,
```



```
                viewModel = viewModel
            )
        }
    }


    LaunchedEffect(uiState) {
        Log.d("MyLog1","State")
        if (uiState is JourneyUiState.Success) {
            viewModel.setJourneyRequest(viewModel.collectJourneyData()!!)
            viewModel.setResponse((uiState as JourneyUiState.Success).response)
            navController.navigate(ScreenRoutes.INFO_SCREEN)
        }
    }
}
class JourneyViewModel(private val repository: JourneyRepository) : ViewModel()
{

    private val journeyRequestBuilder = JourneyRequestBuilder()

    private val _uiState = MutableStateFlow<JourneyUiState>(JourneyUiState.Idle)
    val uiState: StateFlow<JourneyUiState> = _uiState

    private val _leavingFrom = MutableLiveData(Constants.LEAVING_FROM)
    val leavingFrom: LiveData<String> = _leavingFrom

    private val _headingTo = MutableLiveData(Constants.HEADING_TO)
    val headingTo: LiveData<String> = _headingTo
```

```
    private val _selectedDate = MutableLiveData(Constants.SELECTED_DATE)
    val selectedDate: LiveData<String> = _selectedDate

    private val _selectedTime = MutableLiveData(Constants.SELECTED_TIME)
    val selectedTime: LiveData<String> = _selectedTime

    private val _sliderValueFrequencyElement = MutableLiveData(30f)
    val sliderValueFrequencyElement: LiveData<Float> =
_sliderValueFrequencyElement

    private val _sliderValueAverangeSpeedElement = MutableLiveData(80f)
    val sliderValueAverangeSpeedElement: LiveData<Float> =
_sliderValueAverangeSpeedElement

    private val _intermediatePoints =
MutableLiveData<List<IntermediatePoint>>(listOf(
    ))
    val intermediatePoints: LiveData<List<IntermediatePoint>> =
_intermediatePoints

    private val _errorMessages = MutableLiveData<String>()
    val errorMessages: LiveData<String> = _errorMessages

    private val _journeyRequestJson = MutableStateFlow<JourneyRequest?>(null)
    val journeyRequestJson: StateFlow<JourneyRequest?> get() =
_journeyRequestJson

    private val _responseJson = MutableStateFlow<JourneyResponse?>(null)
    val responseJson: StateFlow<JourneyResponse?> = _responseJson

    fun setResponse(response: JourneyResponse) {
```



```
        viewModelScope.launch {
            _responseJson.emit(response)
        }
    }

    fun setJourneyRequest(journeyRequest: JourneyRequest) {
        viewModelScope.launch {
            _journeyRequestJson.emit(journeyRequest)
        }
    }

    fun submitJourney(request: JourneyRequest) {
        _uiState.value = JourneyUiState.Loading
        viewModelScope.launch {
            try {
                val response = repository.submitJourney(request)
                _uiState.value = JourneyUiState.Success(response)
            } catch (e: Exception) {
                _uiState.value = JourneyUiState.Error(e.message ?: "Unknown
error")
            }
        }

    }

    fun setNewUIStateIdle(){
```

```
        _uiState.value = JourneyUiState.Idle
    }

    fun updateSliderValueFrequencyElement(newValue: Float) {
        _sliderValueFrequencyElement.value = newValue
    }

    fun updateSliderValueAverangeSpeedElement(newValue: Float) {
        _sliderValueAverangeSpeedElement.value = newValue
    }

    fun removeIntermediatePoint(index: Int) {
        _intermediatePoints.value =
_intermediatePoints.value!!.toMutableList().apply {
            if (index in indices) {
                removeAt(index)
            }
        }
    }

    fun collectJourneyData(): JourneyRequest? {
        val leavingFrom = _leavingFrom.value ?: ""
        val headingTo = _headingTo.value ?: ""
        val intermediatePoints = _intermediatePoints.value ?: emptyList()
        val selectedDate = _selectedDate.value ?: ""
        val selectedTime = _selectedTime.value ?: ""
        val sliderValueFrequency =
_sliderValueFrequencyElement.value?.toDouble() ?: 30.0
        val sliderValueAverageSpeed =
_sliderValueAverangeSpeedElement.value?.toDouble() ?: 80.0

        val isValid = journeyRequestBuilder.validateInputs(
            leavingFrom = leavingFrom,
            headingTo = headingTo,
            selectedDate = selectedDate,
```



```
            selectedTime = selectedTime,
            sliderValueFrequency = sliderValueFrequency,
            sliderValueAverageSpeed = sliderValueAverageSpeed,
            intermediatePoints = intermediatePoints
        )

        if (isValid) {
            val request = journeyRequestBuilder.formTravelData(
                startLocation = leavingFrom,
                endLocation = headingTo,
                startDate = selectedDate,
                startTime = selectedTime,
                speed = sliderValueAverageSpeed,
                stepKM = sliderValueFrequency,
                intermediatePoints = intermediatePoints
            )
            return request
        } else {
            _errorMessages.value = journeyRequestBuilder._errorMessages.value
            return null
        }
    }

    fun updateSelectedTime(newTime: String) {
        _selectedTime.value = newTime
```

```
    }

    fun updateSelectedDate(newDate: String) {
        _selectedDate.value = newDate
    }

    fun updateIntermediatePointStopHours(index: Int, stopHours: String) {
        val updatedList = _intermediatePoints.value?.toMutableList() ?:
mutableListOf()
        if (index in updatedList.indices) {
            updatedList[index] = updatedList[index].copy(stopHours = stopHours)
            _intermediatePoints.value = updatedList
        }
    }

    fun updateIntermediatePointStopMinutes(index: Int, stopMinutes: String) {
        val updatedList = _intermediatePoints.value?.toMutableList() ?:
mutableListOf()
        if (index in updatedList.indices) {
            updatedList[index] = updatedList[index].copy(stopMinutes =
stopMinutes)
            _intermediatePoints.value = updatedList
        }
    }

    fun addIntermediatePoint() {
        val updatedList = _intermediatePoints.value?.toMutableList() ?:
mutableListOf()
        updatedList.add(IntermediatePoint(Constants.CHOOSE_A_STOP, "", ""))
        _intermediatePoints.value = updatedList
    }

    fun updateIntermediatePoint(index: Int, location: String) {
        val updatedList = _intermediatePoints.value?.toMutableList() ?:
mutableListOf()
        if (index in updatedList.indices) {
```



```
            updatedList[index] = updatedList[index].copy(location = location)
            _intermediatePoints.value = updatedList
        }
    }

    fun updateHeadingTo(address: String) {
        _headingTo.value = address
    }

    fun updateLeavingFrom(address: String) {
        _leavingFrom.value = address
    }

}
class JourneyRequestBuilder {

    var _errorMessages: MutableState<String?> = mutableStateOf(null)

    fun validateInputs(
        leavingFrom: String?,
        headingTo: String?,
        selectedDate: String?,
        selectedTime: String?,
        sliderValueFrequency: Double?,
```

```
        sliderValueAverageSpeed: Double?,
        intermediatePoints: List<IntermediatePoint>
    ): Boolean {
        var errorMessage = ""

        if (leavingFrom.isNullOrEmpty() || leavingFrom ==
Constants.LEAVING_FROM) {
            errorMessage += "Please specify the departure location.\n"
        }

        if (headingTo.isNullOrEmpty() || headingTo == Constants.HEADING_TO) {
            errorMessage += "Please specify the destination.\n"
        }

        if (selectedDate.isNullOrEmpty()) {
            errorMessage += "Please select a valid date.\n"
        }

        if (selectedTime.isNullOrEmpty()) {
            errorMessage += "Please select a valid time.\n"
        }

        val actualDate = getActualDate(selectedDate!!)
        val actualTime = getActualTime(selectedTime!!)
        if (!isDateWithinLimit(actualDate, actualTime)) {
            errorMessage += "The date must not be more than 5 days from
today.\n"
        }

        if (sliderValueFrequency == null || sliderValueFrequency <= 0) {
            errorMessage += "The frequency value must be greater than 0.\n"
        }

        if (sliderValueAverageSpeed == null || sliderValueAverageSpeed <= 0) {
            errorMessage += "The average speed value must be greater than 0.\n"
        }
```



```


        if (intermediatePoints.isNotEmpty()) {
            intermediatePoints.forEachIndexed { index, point ->
                if (point.location.isNullOrEmpty() || point.location ==
Constants.CHOOSE_A_STOP) {
                    errorMessage += "Intermediate point at position ${index + 1}
is missing location.\n"
                }
            }
        }

        if (errorMessage.isNotEmpty()) {
            _errorMessages.value = errorMessage
            return false
        }

        return true
    }

    fun formTravelData(
        startLocation: String,
        endLocation: String,
```

```
        startDate: String,
        startTime: String,
        speed: Double,
        stepKM: Double,
        intermediatePoints: List<IntermediatePoint>
    ): JourneyRequest? {
        val actualDate = getActualDate(startDate)
        val actualTime = getActualTime(startTime)

        val formattedDateTime = formatDateTime(actualDate, actualTime) ?: return
null

        val listCity = buildCityList(startLocation, endLocation,
intermediatePoints)

        return JourneyRequest(
            cities = listCity,
            departureTime = formattedDateTime,
            stepDistance = stepKM * 1000.0,
            averageSpeed = speed,
            metricWeather = "metric"
        )
    }

    private fun getActualDate(startDate: String): String {
        return if (startDate == Constants.SELECTED_DATE) {
            SimpleDateFormat("d/M/yyyy",
Locale.getDefault()).format(System.currentTimeMillis())
        } else {
            startDate
        }
    }

    private fun getActualTime(startTime: String): String {
        return if (startTime == Constants.SELECTED_TIME) {
            SimpleDateFormat("HH:mm",
Locale.getDefault()).format(System.currentTimeMillis())
        } else {
            startTime
```



```
        }
    }

    private fun isDateWithinLimit(date: String, time: String): Boolean {
        val inputFormat = SimpleDateFormat("HH:mm d/M/yyyy",
Locale.getDefault())
        val currentDate = Calendar.getInstance()
        val inputDate = Calendar.getInstance()

        return try {
            val parsedDate = inputFormat.parse("$time $date")
            inputDate.time = parsedDate!!

            val diffInMillis = inputDate.timeInMillis - currentDate.timeInMillis
            TimeUnit.MILLISECONDS.toDays(diffInMillis) <= 5
        } catch (e: Exception) {
            false
        }
    }

    private fun formatDateTime(date: String, time: String): String? {
```

```
        val inputFormat = SimpleDateFormat("HH:mm d/M/yyyy",
Locale.getDefault())
        val outputFormat = SimpleDateFormat("HH:mm dd.MM.yyyy",
Locale.getDefault())

        return try {
            val parsedDate = inputFormat.parse("$time $date")
            outputFormat.format(parsedDate!!)
        } catch (e: Exception) {
            null
        }
    }

    private fun buildCityList(
        startLocation: String,
        endLocation: String,
        intermediatePoints: List<IntermediatePoint>
    ): List<CityABC> {
        val listCity = mutableListOf<CityABC>()

        listCity.add(CityABC(cityName = startLocation, stopDuration = "00:00"))

        for (point in intermediatePoints) {
            var stopHours = point.stopHours
            var stopMinutes = point.stopMinutes
            if(stopHours == ""){
                stopHours = "0"
            }
            if(stopMinutes == ""){
                stopMinutes = "0"
            }
            val stopTime = String.format("%02d:%02d", stopHours.toInt() / 60,
stopMinutes.toInt() % 60)


            listCity.add(CityABC(cityName = point.location, stopDuration =
stopTime))
        }

        listCity.add(CityABC(cityName = endLocation, stopDuration = "00:00"))
```



```


        return listCity
    }
}
@Composable
fun AddStopButton(viewModel: JourneyViewModel) {
    Column(
        modifier = Modifier
            .fillMaxWidth()
            .padding(top = 16.dp)
    ) {
        Row(
            modifier = Modifier
                .fillMaxWidth()
                .clickable {
                    viewModel.addIntermediatePoint()
                },
            horizontalArrangement = Arrangement.End
        ) {
            Image(
```

```
                painter = painterResource(id = R.drawable.baseline_add_24),
                contentDescription = null,
                modifier = Modifier.padding(end = 8.dp)
            )
            Text(
                text = "Add a stop",
                fontSize = 20.sp,
                color = Color.White,
            )
        }
    }
}
@RequiresApi(Build.VERSION_CODES.O)
@Composable
fun DateAndTimeSection() {
    val currentDate = LocalDate.now()

    val dayOfWeek = currentDate.format(DateTimeFormatter.ofPattern("EEEE",
Locale.ENGLISH))
    val date = currentDate.format(DateTimeFormatter.ofPattern("d.MM"))

    Row(
        modifier = Modifier.fillMaxWidth(),
        horizontalArrangement = Arrangement.SpaceBetween
    ) {
        Text(
            text = dayOfWeek,
            fontSize = 24.sp,
            color = Color.White,
            fontWeight = FontWeight.Bold
        )
        Text(
            text = date,
            fontSize = 24.sp,
            color = Color.White,
            fontWeight = FontWeight.Bold
        )
    }
}
@Composable
fun IntermediatePointRow(
```



```
    point: IntermediatePoint,
    index: Int,
    navController: NavController,
    viewModel: JourneyViewModel
) {
    Column(
        modifier = Modifier
            .fillMaxWidth()
            .padding(start = 30.dp)
    ) {
        InputLocation(
            title = point.location,
            painterResource(id = R.drawable.outline_square_24),
            onClick = {

navController.navigate("address_input_screen/intermediate_point_$index")
            }
        )
```

```
        Row(
            modifier = Modifier.fillMaxWidth(),
            horizontalArrangement = Arrangement.SpaceEvenly,
            verticalAlignment = Alignment.CenterVertically
        ) {
            TimeInputField(
                value = point.stopHours,
                onValueChange = { newValue ->
                    viewModel.updateIntermediatePointStopHours(index, newValue)
                },
                placeholder = "0",
                unitText = "hr",
                modifier = Modifier.weight(1f)
            )

            Spacer(modifier = Modifier.width(20.dp))

            TimeInputField(
                value = point.stopMinutes,
                onValueChange = { newValue ->
                    viewModel.updateIntermediatePointStopMinutes(index,
newValue)
                },
                placeholder = "0",
                unitText = "min",
                modifier = Modifier.weight(1f)
            )
            Spacer(modifier = Modifier.width(20.dp))

            IconButton(
                onClick = {
                    viewModel.removeIntermediatePoint(index)
                },
                modifier = Modifier.size(48.dp)
            ) {
                Image(
                    painterResource(R.drawable.baseline_delete_24),
                    contentDescription = "Delete"
                )
            }
        }
        Spacer(modifier = Modifier.height(16.dp))
```



```
    }
}
@Composable
fun JourneyDetailsCard(
    leavingFrom: String,
    headingTo: String,
    intermediatePoints: List<IntermediatePoint>,
    navController: NavController,
    selectedDate: String,
    selectedTime: String,
    sliderValueFrequencyElement: Float,
    sliderValueAverangeSpeedElement: Float,
    viewModel: JourneyViewModel
) {
    val context = LocalContext.current

    Column(
        modifier = Modifier
```

```
            .fillMaxWidth()
            .clip(RoundedCornerShape(16.dp))
            .background(colorResource(id = R.color.colorCard)),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally
    ){
        Column(
            modifier = Modifier
                .fillMaxWidth()
                .clip(RoundedCornerShape(16.dp))
                .background(colorResource(id = R.color.colorCard))
                .padding(16.dp),
            verticalArrangement = Arrangement.Center,
            horizontalAlignment = Alignment.CenterHorizontally
        ) {
            Text(
                modifier = Modifier.padding(16.dp),
                text = stringResource(id = R.string.where_to_go),
                fontSize = 24.sp,
                color = Color.White,
                fontWeight = FontWeight.Bold
            )

            InputLocation(
                title = leavingFrom,
                painterResource(id = R.drawable.kolo),
                onClick = {

navController.navigate("address_input_screen/leaving_from_address")
                }
            )

            Spacer(modifier = Modifier.height(16.dp))

            intermediatePoints.forEachIndexed { index, point ->
                IntermediatePointRow(
                    point = point,
                    index = index,
                    navController = navController,
                    viewModel = viewModel
                )
            }
```



```
            AddStopButton(viewModel)

            InputLocation(
                title = headingTo,
                painterResource(id = R.drawable.metka),
                onClick = {

navController.navigate("address_input_screen/heading_to_address")
                }
            )

            Row(
                modifier = Modifier
                    .fillMaxWidth(),
                verticalAlignment = Alignment.CenterVertically,
                horizontalArrangement = Arrangement.SpaceBetween
            ) {
```

```
                DatePickerElement(
                    modifier = Modifier.weight(1f),
                    selectedDate = selectedDate,
                    onDateSelected = { viewModel.updateSelectedDate(it) }
                )
                Spacer(modifier = Modifier.width(25.dp))
                TimePickerElement(
                    modifier = Modifier.weight(1f),
                    selectedTime = selectedTime,
                    onTimeSelected = { viewModel.updateSelectedTime(it) }
                )
            }
            Spacer(modifier = Modifier.height(24.dp))

            FrequencySliderElement(
                initValue = sliderValueFrequencyElement,
                onValueChange = { newValue ->
viewModel.updateSliderValueFrequencyElement(newValue) }
            )
            Spacer(modifier = Modifier.height(16.dp))

            AverangeSpeedSliderElement(
                initValue = sliderValueAverangeSpeedElement,
                onValueChange = { newValue ->
viewModel.updateSliderValueAverangeSpeedElement(newValue) }
            )

        }

        Box(
            modifier = Modifier
                .fillMaxWidth()
                .height(60.dp)
                .background(colorResource(id = R.color.colorButton))
                .clickable {
                    val journeyRequest = viewModel.collectJourneyData()
                    if(journeyRequest != null){
                        viewModel.submitJourney(journeyRequest)
                    }else{
                        viewModel.errorMessages.value?.let {
                            Toast.makeText(context, it,
Toast.LENGTH_LONG).show()
                        }
                    }
```



```
                },
            contentAlignment = Alignment.Center
        ) {
            Text(
                text = stringResource(id = R.string.start_journey),
                fontSize = 20.sp,
                color = colorResource(id = R.color.back),
                fontWeight = FontWeight.Bold
            )
        }
    }


}
@Composable
fun LogoAndSettingsSection(navController: NavController) {
```

```
    Box(
        modifier = Modifier
            .fillMaxWidth()
            .padding(top = 16.dp),
        contentAlignment = Alignment.Center
    ) {
        Row(
            modifier = Modifier.fillMaxWidth(),
            horizontalArrangement = Arrangement.End
        ) {
            IconButton(
                onClick = {
                    navController.navigate("settings_screen")
                }
            ) {
                Image(
                    painter = painterResource(id = R.drawable.settings),
                    contentDescription = null,
                    contentScale = ContentScale.Fit
                )
            }
        }
        Row(
            modifier = Modifier.fillMaxWidth(),
            horizontalArrangement = Arrangement.Center
        ) {

            Image(
                modifier = Modifier.size(50.dp),
                painter = painterResource(id = R.drawable.logo_light),
                contentDescription = null,
                contentScale = ContentScale.Fit
            )
        }
    }
}
object NetworkModule {
    private const val BASE_URL = "https://babakdriver.xyz"

    private val logging = HttpLoggingInterceptor().apply {
        level = HttpLoggingInterceptor.Level.BODY
    }

    private val httpClient = OkHttpClient.Builder()
```



```
        .addInterceptor(logging)
        .connectTimeout(3, TimeUnit.MINUTES)
        .readTimeout(3, TimeUnit.MINUTES)
        .writeTimeout(3, TimeUnit.MINUTES)
        .build()


    val apiService: ApiService by lazy {
        Retrofit.Builder()
            .baseUrl(BASE_URL)
            .client(httpClient)
            .addConverterFactory(GsonConverterFactory.create())
            .build()
            .create(ApiService::class.java)
```

```
    }
}
interface ApiService {
    @POST("api/babak")
    suspend fun submitJourney(@Body request: JourneyRequest): JourneyResponse
}
sealed class JourneyUiState {
    object Idle : JourneyUiState()
    object Loading : JourneyUiState()
    data class Success(val response: JourneyResponse) : JourneyUiState()
    data class Error(val message: String) : JourneyUiState()
}
object Constants {
    const val LEAVING_FROM = "Leaving from..."
    const val HEADING_TO = "Heading to..."
    const val SELECTED_DATE = "Today"
    const val SELECTED_TIME = "Now"
    const val CHOOSE_A_STOP = "Choose a stop"
}
@Composable
fun TimePickerElement(
    modifier: Modifier = Modifier,
    selectedTime: String,
    onTimeSelected: (String) -> Unit
) {
    val calendar = Calendar.getInstance()
    val hour = calendar.get(Calendar.HOUR_OF_DAY)
    val minute = calendar.get(Calendar.MINUTE)

    val timePickerDialog = TimePickerDialog(
        LocalContext.current,
        { _, selectedHour, selectedMinute ->
            onTimeSelected(String.format("%02d:%02d", selectedHour,
selectedMinute))
        },
        hour, minute, true
    )

    Column(
        modifier = modifier
            .fillMaxWidth()
            .padding(top = 16.dp)
    ) {
        Row(
            modifier = Modifier
                .fillMaxWidth()
                .clickable { timePickerDialog.show() },
```



```
            verticalAlignment = Alignment.CenterVertically,
            horizontalArrangement = Arrangement.SpaceBetween
        ) {
            Icon(
                painter = painterResource(id =
R.drawable.outline_access_time_24),
                contentDescription = null,
                tint = Color.White,
                modifier = Modifier.padding(end = 8.dp, start =
12.dp).size(24.dp)
            )
            Text(
                text = selectedTime,
```

```
                color = Color.White,
                fontSize = 16.sp,
                modifier = Modifier.weight(1f),
                textAlign = TextAlign.Start
            )
        }
        Spacer(modifier = Modifier.height(8.dp))
        Box(
            modifier = Modifier
                .fillMaxWidth()
                .height(1.dp)
                .background(Color.White)
        )
    }
}
@Composable
fun TimeInputField(
    value: String,
    onValueChange: (String) -> Unit,
    placeholder: String,
    unitText: String,
    modifier: Modifier = Modifier
) {
    Column(
        modifier = modifier
            .fillMaxWidth()
    ) {
        Row(
            modifier = Modifier.fillMaxWidth(),
            verticalAlignment = Alignment.CenterVertically,
            horizontalArrangement = Arrangement.SpaceBetween
        ) {
            TextField(

                value = value,
                onValueChange = {
                    if (it.all { char -> char.isDigit() }) onValueChange(it)
                },
                textStyle = LocalTextStyle.current.copy(
                    color = Color.White,
                    fontSize = 18.sp
                ),
                singleLine = true,
                modifier = Modifier.fillMaxSize().weight(1f),
                colors = TextFieldDefaults.colors(
                    focusedContainerColor = Color.Transparent,
                    unfocusedContainerColor = Color.Transparent,
                    cursorColor = Color.White,
```



```
                    focusedTextColor = Color.White,
                    unfocusedTextColor = Color.White,
                    focusedIndicatorColor = Color.Transparent,
                    unfocusedIndicatorColor = Color.Transparent
                ),
                placeholder = {
                    Text(
                        text = placeholder,
                        color = Color.LightGray,
                        fontSize = 18.sp
                    )
                }
```

```
            )


            Text(
                text = unitText,
                color = Color.White,
                fontSize = 18.sp,
                modifier = Modifier.padding(end = 16.dp)
            )
        }

        Box(
            modifier = Modifier
                .fillMaxWidth()
                .height(1.dp)
                .background(Color.White)
        )
    }
}
@Composable
fun InputLocation(title: String, icon: Painter, onClick: () -> Unit) {
    Column(
        modifier = Modifier
            .fillMaxWidth()
            .padding(top = 8.dp)
    ) {
        Box(
            modifier = Modifier
                .fillMaxWidth()
                .clickable { onClick() }
        ) {
            Row(
                modifier = Modifier.fillMaxWidth(),
                verticalAlignment = Alignment.CenterVertically
            ) {
                Image(
                    painter = icon,
                    contentDescription = null,
                    modifier = Modifier
                        .padding(8.dp)
                        .size(20.dp)
                )
                Text(
                    text = title,
                    color = Color.LightGray,
                    fontSize = 18.sp
                )
```



```
            }
        }
        Spacer(modifier = Modifier.height(8.dp))
        Box(
            modifier = Modifier
                .fillMaxWidth()
                .height(1.dp)
                .background(Color.White)
        )
```

```
    }
}
@Composable
fun CustomSliderElement(
    label: String,
    initialValue: Float,
    valueRange: ClosedFloatingPointRange<Float>,
    unit: String,
    onValueChange: (Float) -> Unit
) {
    var sliderValue by remember { mutableStateOf(initialValue) }

    Row(
        modifier = Modifier
            .fillMaxWidth()
    ) {
        Column(
            modifier = Modifier.weight(1f)
        ) {
            Text(
                text = label,
                color = Color.White,
                fontSize = 16.sp,
                modifier = Modifier.align(Alignment.Start)
            )
            CustomSlider(
                value = sliderValue,
                onValueChange = {
                    sliderValue = it
                    onValueChange(it)
                },
                valueRange = valueRange,
                trackHeight = 2f,
                thumbRadius = 15f
            )
        }

        Spacer(modifier = Modifier.width(16.dp))

        Box(
            modifier = Modifier
                .border(1.dp, Color.White, RoundedCornerShape(8.dp))
                .padding(horizontal = 12.dp, vertical = 4.dp),
            contentAlignment = Alignment.Center
        ) {
            Text(
                text = "${sliderValue.toInt()} $unit",
                color = Color.White,
                fontSize = 24.sp,
                textAlign = TextAlign.Center
            )
        }
```



```
    }
}
@Composable
fun AverangeSpeedSliderElement(initValue : Float, onValueChange: (Float) ->
Unit) {
    CustomSliderElement(
        label = "Average speed",
        initialValue = initValue,
```

```
        valueRange = 0f..250f,
        unit = "км/год",
        onValueChange = onValueChange
    )
}
@Composable
fun AddressInputScreen(
    onBackPressed: () -> Unit,
    onAddressSelected: (String) -> Unit,
    onMapPick: () -> Unit,
    navController: NavController

) {
    val savedStateHandle = navController.currentBackStackEntry?.savedStateHandle
    val newLocationLiveData =
savedStateHandle!!.getLiveData<String>("mapLocation")
    val newLocationState = newLocationLiveData.observeAsState()
    val newLocationFromMap: String? = newLocationState.value

    var searchQuery by remember { mutableStateOf(TextFieldValue("")) }
    var predictions by remember { mutableStateOf<List<String>>(emptyList()) }

    LaunchedEffect(newLocationFromMap) {
        if (!newLocationFromMap.isNullOrBlank()) {
            searchQuery = TextFieldValue(newLocationFromMap)
            onAddressSelected(newLocationFromMap)
        }
    }

    val placesClient: PlacesClient = remember {
        Places.createClient(App.INSTANCE)
    }

    val locationPermissionLauncher =

rememberLauncherForActivityResult(ActivityResultContracts.RequestPermission()) {
isGranted ->

        }

    @SuppressLint("MissingPermission")
    fun getCurrentLocation(
        onCoordinatesReceived: (latitude: Double, longitude: Double) -> Unit
    ) {
        val fusedLocationClient =
LocationServices.getFusedLocationProviderClient(App.INSTANCE)
        fusedLocationClient.lastLocation
            .addOnSuccessListener { location: Location? ->
                if (location != null) {
                    Log.d("LocationTest", "lat=${location.latitude},
lng=${location.longitude}")
                    onCoordinatesReceived(location.latitude, location.longitude)
                } else {
```



```
                    val toast = Toast.makeText(App.INSTANCE, "Your geolocation
could not be obtained. Turn on geolocation.", Toast.LENGTH_SHORT)
                    toast.setGravity(Gravity.TOP or Gravity.CENTER_HORIZONTAL,
0, 100)
                    toast.show()
                    Log.d("LocationTest", "location is null")
                }
```

```
            }
            .addOnFailureListener { e ->
                Log.e("LocationTest", "Error getting location", e)
            }

    }

    Column(
        modifier = Modifier
            .fillMaxSize()
            .background(colorResource(id = R.color.back))

    ) {
        Column(
            modifier = Modifier
                .fillMaxWidth()
                .clip(
                    RoundedCornerShape(
                        topStart = 0.dp,
                        topEnd = 0.dp,
                        bottomStart = 16.dp,
                        bottomEnd = 16.dp
                    )
                )
                .background(colorResource(id = R.color.colorCard))
                .padding(top = 15.dp)
                .padding(16.dp),
            verticalArrangement = Arrangement.Top
        ) {
            Row(
                verticalAlignment = Alignment.CenterVertically,
                modifier = Modifier
                    .fillMaxWidth()
                    .border(
                        width = 2.dp,
                        color = colorResource(id = R.color.colorButton),
                        shape = RoundedCornerShape(10.dp)
                    )
                    .padding(start = 12.dp)
            ) {
                IconButton(onClick = onBackPressed) {
                    Image(
                        painter = painterResource(id =
R.drawable.baseline_arrow_back_ios_24),
                        contentDescription = null,
                        contentScale = ContentScale.Fit
                    )
                }
                OutlinedTextField(
                    value = searchQuery.text,
                    onValueChange = { query ->
                        searchQuery = TextFieldValue(query)
                        if (query.isNotBlank()) {
                            getAutocompletePredictions(query, placesClient) {
```



```
predictionsList ->
                                predictions = predictionsList
                            }
                        } else {
                            predictions = emptyList()
                        }
```

```
                    },
                    textStyle = LocalTextStyle.current.copy(
                        color = Color.White,
                        fontSize = 18.sp
                    ),
                    singleLine = true,
                    modifier = Modifier.weight(1f),
                    placeholder = {
                        Text(
                            text = "Enter address",
                            color = Color.LightGray,
                            fontSize = 18.sp
                        )
                    },
                    colors = TextFieldDefaults.colors(
                        unfocusedContainerColor = Color.Transparent,
                        focusedContainerColor = Color.Transparent,
                        unfocusedIndicatorColor = Color.Transparent,
                        focusedIndicatorColor = Color.Transparent,
                        cursorColor = Color.White
                    )
                )

                IconButton(
                    onClick = {
                        val permissionStatus =
ContextCompat.checkSelfPermission(
                            App.INSTANCE,
                            Manifest.permission.ACCESS_FINE_LOCATION
                        )
                        if (permissionStatus ==
PackageManager.PERMISSION_GRANTED) {
                            getCurrentLocation { lat, lng ->
                                val coords = "$lat,$lng"
                                onAddressSelected(coords)
                            }
                        } else {

locationPermissionLauncher.launch(Manifest.permission.ACCESS_FINE_LOCATION)
                        }
                    }
                ) {
                    Image(
                        painter = painterResource(id =
R.drawable.baseline_my_location_24), // Додайте власну іконку
                        contentDescription = "My Location",
                        contentScale = ContentScale.Fit
                    )
                }
                IconButton(
                    onClick = {
                        onMapPick()
                    }
                ) {
                    Image(
```



```
                        painter = painterResource(id = R.drawable.metka),
                        contentDescription = "Pick from Map",
                        contentScale = ContentScale.Fit
                    )
                }
```

```
            }

            Spacer(modifier = Modifier.height(8.dp))

            LazyColumn {
                items(predictions) { prediction ->
                    Text(
                        text = prediction,
                        style = MaterialTheme.typography.bodyLarge,
                        color = Color.White,
                        modifier = Modifier
                            .clickable {
                                onAddressSelected(prediction)
                            }
                            .padding(8.dp)
                    )
                    Divider(color = Color.Gray, thickness = 1.dp)
                }
            }
        }
    }
}



private fun getAutocompletePredictions(
    query: String,
    placesClient: PlacesClient,
    onResult: (List<String>) -> Unit
) {
    val request = FindAutocompletePredictionsRequest.builder()
        .setQuery(query)
        .build()

    placesClient.findAutocompletePredictions(request)
        .addOnSuccessListener { response ->
            onResult(response.autocompletePredictions.map {
it.getFullText(null).toString() })
        }
        .addOnFailureListener {

        }
}
@Composable
fun MapPickerScreen(
    onLocationSelected: (LatLng) -> Unit,
    onBackPressed: () -> Unit
) {
    val context = LocalContext.current
    val kievLatLng = LatLng(50.4501, 30.5234)

    val cameraPositionState = rememberCameraPositionState {
        position = CameraPosition.fromLatLngZoom(kievLatLng, 10f)
    }
```



```
    val mapStyleJson =
context.resources.openRawResource(R.raw.style1).bufferedReader().use {
it.readText() }
    val mapProperties = MapProperties(
```

```
        isMyLocationEnabled = true,
        mapStyleOptions = MapStyleOptions(mapStyleJson)
    )

    Box(
        modifier = Modifier
            .fillMaxSize()
    ) {
        GoogleMap(
            modifier = Modifier.matchParentSize(),
            cameraPositionState = cameraPositionState,
            properties = mapProperties,
        )

        Image(
            painter = painterResource(id =
R.drawable.baseline_location_pin_24_red),
            contentDescription = "Center Marker",
            contentScale = ContentScale.Fit,
            modifier = Modifier
                .align(Alignment.Center)
                .size(40.dp)
        )

        Button(
            onClick = {
                val centerLatLng = cameraPositionState.position.target
                onLocationSelected(centerLatLng)
            },
            modifier = Modifier
                .align(Alignment.BottomCenter)
                .padding(bottom = 16.dp),
            colors = ButtonDefaults.buttonColors(
                containerColor = Color(0xFFB2FFCC),
                contentColor = Color.Black
            ),
            elevation = ButtonDefaults.buttonElevation(defaultElevation = 0.dp)
        ) {
            Text(
                text = "Select this location",
                fontSize = 16.sp,
                fontWeight = FontWeight.Bold
            )
        }

        IconButton(
            onClick = {
                onBackPressed()
            }
        ) {
            Image(
                modifier = Modifier.size(50.dp).padding(top = 15.dp),
                painter = painterResource(id =
R.drawable.outline_arrow_back_ios_24),
                contentDescription = null,
                contentScale = ContentScale.Fit
            )
```



```
        }
    }
}
```

```
@Composable
fun InfoScreen(navController: NavController, viewModel: JourneyViewModel) {
    val journeyResponse by viewModel.responseJson.collectAsState()
    val journeyRequest by viewModel.journeyRequestJson.collectAsState()
    var showBottomSheet by remember { mutableStateOf(false) }
    var topBarHeight by remember { mutableStateOf(0) }
    var bottomBarHeight by remember { mutableStateOf(0) }

    Box(
        modifier = Modifier
            .fillMaxSize()
            .background(colorResource(id = R.color.back))
            .padding(WindowInsets.systemBars.asPaddingValues())
    ) {
        Column(
            modifier = Modifier
                .fillMaxSize()
        ) {
            MapView(journeyResponse)
        }
        TopBar(
            modifier = Modifier
                .fillMaxWidth()
                .align(Alignment.TopCenter),
            onButtonSetting =  { navController.navigate("settings_screen") },
            onButtonBack = {
                viewModel.setNewUIStateIdle()
                            navController.popBackStack()
                           },
            journeyRequest = journeyRequest!!
        )
        BottomBar(
            onButtonClick = { showBottomSheet = true },
            modifier = Modifier
                .align(Alignment.BottomCenter)
                .onGloballyPositioned { coordinates ->
                    bottomBarHeight = coordinates.size.height
                },
            journeyResponse = journeyResponse!!
        )
        if (showBottomSheet) {
            BottomSheetContent(
                onDismiss = { showBottomSheet = false },
                journeyResponse = journeyResponse!!
            )
        }
    }
}
@Composable
fun BottomBar(onButtonClick: () -> Unit, modifier: Modifier = Modifier,
journeyResponse: JourneyResponse) {
    BottomAppBar(
        containerColor = colorResource(id = R.color.back),
        modifier = modifier
    ) {
        Text(
            text = "${journeyResponse.allTime}
(${journeyResponse.allDistance})",
```



```
            color = Color.White,
            modifier = Modifier.padding(16.dp)
```

```
        )
        Spacer(modifier = Modifier.weight(1f))
        Button(
            onClick = onButtonClick,
            colors = ButtonDefaults.buttonColors(containerColor =
colorResource(id = R.color.colorButton)),
            modifier = Modifier.padding(16.dp)

        ) {
            Row (
                verticalAlignment = Alignment.CenterVertically,
                horizontalArrangement = Arrangement.Center
            ){
                Text(
                    "All info",
                    color = colorResource(id = R.color.back)
                )
                Image(
                    painterResource(R.drawable.outline_arrow_back_ios_24),
                    contentDescription = "arrow_back",
                    modifier = Modifier.graphicsLayer(
                        rotationZ = 90f
                    ),
                    colorFilter = ColorFilter.tint(colorResource(R.color.back))
                )
            }

        }
    }
}
fun BottomSheetContent(onDismiss: () -> Unit, journeyResponse: JourneyResponse)
{
    //val (safety, comfort) = calculateSafetyAndComfort(journeyResponse)
    ModalBottomSheet(
        containerColor = colorResource(R.color.back),
        onDismissRequest = onDismiss
    ) {
        Column(
            modifier = Modifier
                .fillMaxWidth()
                .padding(16.dp)
                .navigationBarsPadding()
                .verticalScroll(rememberScrollState()),
            verticalArrangement = Arrangement.Top,
            horizontalAlignment = Alignment.CenterHorizontally
        ) {
            Column (
                modifier = Modifier.fillMaxWidth(),
                horizontalAlignment = Alignment.CenterHorizontally,
            ) {
                Box (
                    modifier = Modifier.fillMaxWidth()
                ){
                    Row (
                        modifier = Modifier.fillMaxWidth(),
                        horizontalArrangement = Arrangement.Center
                    ){
                        Text(
                            text = "Statistic",
```



```
                            style = MaterialTheme.typography.titleLarge,
```

```
                            modifier = Modifier.padding(bottom = 16.dp),
                            color = Color.White
                        )
                    }
                    Row (
                        modifier = Modifier.fillMaxWidth(),
                        horizontalArrangement = Arrangement.End
                    ){
                        OutlinedButton(
                            onClick = onDismiss,
                            colors = ButtonDefaults.outlinedButtonColors(
                                containerColor = Color.Transparent,
                                contentColor = colorResource(id = R.color.text)
                            ),
                            border = BorderStroke(2.dp, colorResource(id =
R.color.text)),
                            shape = RoundedCornerShape(28.dp)
                        ) {
                            Row(
                                verticalAlignment = Alignment.CenterVertically,
                                horizontalArrangement = Arrangement.Center
                            ) {
                                Text(
                                    text = "Hide",
                                    color = colorResource(id = R.color.text),
                                    style = MaterialTheme.typography.bodyMedium
                                )
                                Spacer(modifier = Modifier.width(4.dp))
                                Image(
                                    painter =
painterResource(R.drawable.outline_arrow_back_ios_24),
                                    contentDescription = "arrow_back",
                                    modifier = Modifier.graphicsLayer(rotationZ
= 270f),
                                    colorFilter =
ColorFilter.tint(colorResource(R.color.colorButton))
                                )
                            }
                        }
                    }
                }

                Row(
                    modifier = Modifier.fillMaxWidth().padding(vertical = 8.dp),
                    horizontalArrangement = Arrangement.SpaceBetween
                ){
                    Text(
                        //text = "Safety: $safety%",
                        text = "Safety: ${journeyResponse.dangerRisk}%",
                        color = Color.Red
                    )
                    Text(
                        text = "Comfort: ${journeyResponse.comfortRisk}%",
                        color = Color.Yellow
                    )
                }
                Spacer(modifier = Modifier.height(16.dp))

                Row(
```

```
                modifier = Modifier.fillMaxWidth(),
                horizontalArrangement = Arrangement.SpaceBetween
            ) {
                Text(
                    text = journeyResponse.allTime,
                    color = colorResource(R.color.text),
                    fontSize = 16.sp
                )
                Text(
                    text = journeyResponse.allDistance,
                    color = colorResource(R.color.text),
                    fontSize = 16.sp
                )
            }

            Spacer(modifier = Modifier.height(16.dp))

            Row(
                modifier = Modifier.fillMaxWidth().padding(vertical = 8.dp),
                horizontalArrangement = Arrangement.SpaceBetween
            ) {
                Text(
                    text = "",
                    modifier = Modifier.weight(1f),
                    style = MaterialTheme.typography.bodyMedium,
                    color = Color.White
                )
                Text(
                    text = "Avg",
                    modifier = Modifier.weight(1f),
                    style = MaterialTheme.typography.bodyMedium,
                    color = Color.White,
                    textAlign = TextAlign.Center
                )
                Text(
                    text = "Min",
                    modifier = Modifier.weight(1f),
                    style = MaterialTheme.typography.bodyMedium,
                    color = Color.White,
                    textAlign = TextAlign.Center
                )
                Text(
                    text = "Max",
                    modifier = Modifier.weight(1f),
                    style = MaterialTheme.typography.bodyMedium,
                    color = Color.White,
                    textAlign = TextAlign.Center
                )
            }
            Spacer(modifier = Modifier.height(8.dp))
            val stats = listOf(
                Triple("Temperature", journeyResponse.averageTemp to "°C",
journeyResponse.minTemp to journeyResponse.maxTemp to "°C"),
                Triple("Cloudiness", journeyResponse.averageCloudiness to "%",
journeyResponse.minCloudiness to journeyResponse.maxCloudiness to "%"),
                Triple("Wind Speed", journeyResponse.averageWindSpeed to "m/s",
journeyResponse.minWindSpeed to journeyResponse.maxWindSpeed to "m/s"),
                Triple("Humidity", journeyResponse.averageHumidity to "%",
journeyResponse.minHumidity to journeyResponse.maxHumidity to "%"),
                Triple("Visibility", journeyResponse.averageVisibility to "m",
journeyResponse.minVisibility to journeyResponse.maxVisibility to "m"),
```

```
                Triple("Wind Gust", journeyResponse.averageWindGust to "m/s",
journeyResponse.minWindGust to journeyResponse.maxWindGust to "m/s"),
                Triple("Wind Degree", journeyResponse.averageWindDeg to "°",
journeyResponse.minWindDeg to journeyResponse.maxWindDeg to "°")
            )
            stats.forEachIndexed { index, (parameter, avgWithUnit,
minMaxWithUnit) ->
                Row(
                    modifier = Modifier
                        .fillMaxWidth()
                        .padding(vertical = 16.dp),
                    horizontalArrangement = Arrangement.SpaceBetween
                ) {
                    Text(
                        text = parameter,
                        modifier = Modifier.weight(1f),
                        style = MaterialTheme.typography.bodySmall,
                        color = Color.White
                    )
                    Text(
                        text = "${avgWithUnit.first} ${avgWithUnit.second}",
                        modifier = Modifier.weight(1f),
                        style = MaterialTheme.typography.bodySmall,
                        textAlign = TextAlign.Center,
                        color = colorResource(R.color.text)
                    )
                    Text(
                        text = "${minMaxWithUnit.first.first}
${minMaxWithUnit.second}",
                        modifier = Modifier.weight(1f),
                        style = MaterialTheme.typography.bodySmall,
                        textAlign = TextAlign.Center,
                        color = colorResource(R.color.text)
                    )
                    Text(
                        text = "${minMaxWithUnit.first.second}
${minMaxWithUnit.second}",
                        modifier = Modifier.weight(1f),
                        style = MaterialTheme.typography.bodySmall,
                        textAlign = TextAlign.Center,
                        color = colorResource(R.color.text)
                    )
                }
                if (index < stats.size - 1) {
                    Box(
                        modifier = Modifier
                            .fillMaxWidth()
                            .height(1.dp)
                            .background(colorResource(R.color.colorCard))
                    )
                }
            }
        }
    }
}
@Composable
fun MapView(journeyResponse: JourneyResponse?) {
    val context = LocalContext.current

    val cameraPositionState = rememberCameraPositionState {
        journeyResponse?.allTravel?.let { focusOnPath(this, it) }
```

```
    }

    val mapStyleJson =
context.resources.openRawResource(R.raw.style1).bufferedReader().use {
it.readText() }
    val mapProperties = MapProperties(
        isMyLocationEnabled = true,
        mapStyleOptions = MapStyleOptions(mapStyleJson)
    )
    val mapUiSettings = MapUiSettings(zoomControlsEnabled = true)

    Box(modifier = Modifier.fillMaxSize()) {
        GoogleMap(
            modifier = Modifier.fillMaxSize(),
            properties = mapProperties,
            uiSettings = mapUiSettings,
            cameraPositionState = cameraPositionState,
            onMapLoaded = {
                journeyResponse?.allTravel?.let {
focusOnPath(cameraPositionState, it) }
            }
        ) {
            journeyResponse?.let { response ->
                Markers(response.totalWeatherMarkers)

                val route = response.allTravel
                if (route.isNotEmpty()) {
                    Polyline(
                        points = route.map { LatLng(it.latitude, it.longitude)
},
                        color = colorResource(id = R.color.purple_200),
                        width = 8f
                    )
                }
            }
        }
    }
}
@Composable
fun Markers(weatherMarkers: List<WeatherMarker>) {

    weatherMarkers.forEach { location ->
        val latLng = LatLng(location.location.latitude,
location.location.longitude)
        val weather = location.weatherInfo.data[0]
        val severity = getWeatherSeverity(
            weather = weather.weather[0].main,
            windSpeed = weather.wind_speed,
            gust = weather.wind_gust,
            visibility = weather.visibility,
            humidity = weather.humidity,
            cloudiness = weather.clouds
        )

        val iconColor = when (severity) {
            WeatherSeverity.GOOD ->
BitmapDescriptorFactory.defaultMarker(BitmapDescriptorFactory.HUE_GREEN)
            WeatherSeverity.WARNING ->
BitmapDescriptorFactory.defaultMarker(BitmapDescriptorFactory.HUE_YELLOW)
            WeatherSeverity.DANGEROUS ->
```

```
BitmapDescriptorFactory.defaultMarker(BitmapDescriptorFactory.HUE_RED)
        }

        MarkerInfoWindow(
            state = MarkerState(position = latLng),
            title = "Weather Info",
            snippet = location.weatherInfo.toString(),
            icon = iconColor,
            content = {
                val timeAndDate = location.timestamp.split(" ")
                val weather = location.weatherInfo.data[0]
                WeatherCard(
                    time = timeAndDate[0],
                    date = timeAndDate[1],
                    temperature = weather.temp.toString(),
                    humidity = weather.humidity.toString(),
                    windSpeed = weather.wind_speed.toString(),
                    weather = weather.weather[0].main,
                    cloudiness = weather.clouds.toString(),
                    averageVisibility = weather.visibility.toString(),
                    gustOfWind = weather.wind_gust.toString(),
                    icon = weather.weather[0].icon
                )
            }
        )
    }
}

enum class WeatherSeverity {
    GOOD, WARNING, DANGEROUS
}

fun getWeatherSeverity(
    weather: String,
    windSpeed: Double,
    gust: Double,
    visibility: Long,
    humidity: Long,
    cloudiness: Long
): WeatherSeverity {
    if (windSpeed > 7 || gust > 7 || visibility < 4000 ||
        weather.contains("Rain", true) || weather.contains("Thunder", true) ||
weather.contains("Snow", true)
    ) {
        return WeatherSeverity.DANGEROUS
    }

    if (windSpeed > 4 || cloudiness > 70 || humidity > 85) {
        return WeatherSeverity.WARNING
    }

    return WeatherSeverity.GOOD
}
@Composable
fun WeatherCard(
    time: String = "14:26",
    date: String = "8.12.2024",
    temperature: String = "5,68°C",
    humidity: String = "80%",
    windSpeed: String = "5,14 м/с",
```

```
    weather: String = "cloudy",
    cloudiness: String = "100%",
```



```
    averageVisibility: String = "10000 м",
    gustOfWind: String = "10,97 м/с",
    icon: String = "01d"
) {
    val cardBackgroundColor = colorResource(id = R.color.colorCard)


    Card(
        modifier = Modifier
            .padding(16.dp)
            .width(250.dp),
        shape = MaterialTheme.shapes.medium,

        colors = CardDefaults.cardColors(containerColor = cardBackgroundColor)
    ) {
        Column(
            modifier = Modifier
                .padding(16.dp)
        ) {
            Row(
                modifier = Modifier.fillMaxWidth(),
                horizontalArrangement = Arrangement.SpaceBetween
            ) {
                Text(
                    text = time,
                    color = Color.White,
                    style = MaterialTheme.typography.titleMedium
                )
                Text(
                    text = date,
                    color = Color.White,
                    style = MaterialTheme.typography.titleMedium
                )
            }
            Spacer(modifier = Modifier.height(8.dp))

            WeatherInfoItem("Weather", weather, "")
            WeatherInfoItem("Temperature", temperature, "°C")
            WeatherInfoItem("Humidity", humidity, "%")
            WeatherInfoItem("Wind speed", windSpeed, "m/s")
            WeatherInfoItem("Cloudiness", cloudiness, "%")
            WeatherInfoItem("Average visibility", averageVisibility,"m")
            WeatherInfoItem("Gust of wind", gustOfWind, "m/s")
        }
    }
}
@Composable
fun TopBar(
    modifier: Modifier = Modifier,
    onButtonSetting: () -> Unit,
    onButtonBack: () -> Unit,
    journeyRequest: JourneyRequest
) {
    Box(
        modifier = modifier
            .background(colorResource(R.color.back))
```

```
            .fillMaxWidth()
            .padding(vertical = 8.dp)
```



```
    ) {
        Row(
            modifier = Modifier
                .fillMaxWidth()
                .padding(horizontal = 16.dp)
        ) {
            IconButton(
                onClick = onButtonBack,
                modifier = Modifier
                    .background(Color.White, shape = CircleShape)
                    .size(32.dp)
                    .padding(7.dp)
            ) {
                Image(
                    painter = painterResource(id =
R.drawable.outline_arrow_back_ios_24),
                    contentDescription = "Back",
                    modifier = Modifier.fillMaxSize(),
                    colorFilter = ColorFilter.tint(colorResource(R.color.back))
                )
            }


            Spacer(modifier = Modifier.width(16.dp))

            RouteSelection(journeyRequest)

            Spacer(modifier = Modifier.weight(1f))

            IconButton(
                onClick = onButtonSetting
            ) {
                Image(
                    painter = painterResource(id = R.drawable.settings),
                    contentDescription = null,
                    contentScale = ContentScale.Fit
                )
            }
        }
    }
}
fun focusOnPath(cameraPositionState: CameraPositionState, path: List<LatLng>) {
    if (path.isNotEmpty()) {
        val boundsBuilder = LatLngBounds.Builder()
        path.forEach {
boundsBuilder.include(com.google.android.gms.maps.model.LatLng(it.latitude,
it.longitude)) }
        val bounds = boundsBuilder.build()
        val zoomLevel = calculateZoomLevel(bounds)
        cameraPositionState.position =
CameraPosition.fromLatLngZoom(bounds.center, zoomLevel)
    }
}

private fun calculateZoomLevel(bounds: LatLngBounds): Float {
    val latDiff = bounds.northeast.latitude - bounds.southwest.latitude
```

```
    val lngDiff = bounds.northeast.longitude - bounds.southwest.longitude
    val screenWidth = 360
    val latZoom = log2(screenWidth / latDiff)
    val lngZoom = log2(screenWidth / lngDiff)
```



```
    return min(latZoom, lngZoom).toFloat()
}
class LocationPermissionManager(
    private val context: Context,
    private val permissionRequest: ActivityResultLauncher<String>,
    private val onPermissionGranted: () -> Unit,
    private val onPermissionDenied: () -> Unit
) {

    fun checkLocationPermission() {
        if (ContextCompat.checkSelfPermission(
                context,
                Manifest.permission.ACCESS_FINE_LOCATION
            ) == PackageManager.PERMISSION_GRANTED
        ) {
            onPermissionGranted()
        } else {
            permissionRequest.launch(Manifest.permission.ACCESS_FINE_LOCATION)
        }
    }
}
data class JourneyResponse (
    val allTravel: List<LatLng>,
    val allTime: String,
    val allDistance: String,
    val averageTemp: Double,
    val maxTemp: Double,
    val minTemp: Double,
    val averageCloudiness: Double,
    val maxCloudiness: Long,
    val minCloudiness: Long,
    val averageWindSpeed: Double,
    val maxWindSpeed: Double,
    val minWindSpeed: Double,
    val averageHumidity: Double,
    val maxHumidity: Long,
    val minHumidity: Long,
    val averageVisibility: Double,
    val maxVisibility: Long,
    val minVisibility: Long,
    val averageWindGust: Double,
    val maxWindGust: Double,
    val minWindGust: Double,
    val averageWindDeg: Double,
    val maxWindDeg: Long,
    val minWindDeg: Long,
    val mostFrequentWeatherId: Long?,
    val mostFrequentMainCondition: String?,
    val mostFrequentDescription: String?,
    val mostFrequentIcon: String?,
    val dangerRisk: Int,
    val comfortRisk: Int,
    val totalWeatherMarkers: List<WeatherMarker>
)
```

```
data class WeatherMarker(
    val location: LatLng,
    val timestamp: String,
    val weatherInfo: WeatherResponseByTime
)
```



```
data class MapResponse(
    val path: List<LatLng>,
    val distance: String,
    val duration: String,
    val markers: List<Pair<LatLng, String>>
)
data class CityABC(
    val cityName: String,
    val stopDuration: String
)
data class JourneyRequest(
    val cities: List<CityABC>,
    val departureTime: String,
    val stepDistance: Double,
    val averageSpeed: Double,
    val metricWeather: String
)
data class LatLng(
    val latitude: Double,
    val longitude: Double
)
object SavedStateKeys {
    const val LEAVING_FROM_ADDRESS = "leaving_from_address"
    const val HEADING_TO_ADDRESS = "heading_to_address"
    const val INTERMEDIATE_POINT = "intermediate_point_"
}
object ScreenRoutes {
    const val JOURNEY_SCREEN = "journey_screen"
    const val ADDRESS_INPUT_SCREEN = "address_input_screen/{type}"
    const val INFO_SCREEN = "info_screen/{responseJson}"
    const val SETTINGS_SCREEN = "settings_screen"
}
class App : Application() {

    companion object {
        lateinit var INSTANCE: App
    }

    override fun onCreate() {
        super.onCreate()
        INSTANCE = this@App
        if (!Places.isInitialized()) {
            Places.initialize(applicationContext, BuildConfig.PLACES_API_KEY)
        }
    }
}
class JourneyRepository(private val apiService: ApiService) {
    suspend fun submitJourney(request: JourneyRequest): JourneyResponse =
        withContext(Dispatchers.IO) {
            apiService.submitJourney(request)
        }
}
```

```
@Composable
fun SettingsScreen(navController: NavController) {
    val isDarkTheme = remember { mutableStateOf(false) }

    val toggleTheme = {
        isDarkTheme.value = !isDarkTheme.value
    }

    MaterialTheme(
```



```
        colorScheme = if (isDarkTheme.value) darkColorScheme() else
lightColorScheme()
    ) {
        Column(
            modifier = Modifier
                .fillMaxSize()
                .background(colorResource(R.color.back))
                .padding(WindowInsets.systemBars.asPaddingValues())
                .padding(16.dp)
        ) {
            Box(
                modifier = Modifier
                    .fillMaxWidth()
                    .padding(horizontal = 16.dp, vertical = 8.dp)
            ) {

                IconButton(
                    onClick = { navController.popBackStack() },
                    modifier = Modifier
                        .align(Alignment.CenterStart)
                        .background(Color.White, shape = CircleShape)
                        .size(32.dp)
                        .padding(7.dp)
                ) {
                    Image(
                        painter = painterResource(id =
R.drawable.outline_arrow_back_ios_24),
                        contentDescription = "Back",
                        modifier = Modifier.fillMaxSize(),
                        colorFilter =
ColorFilter.tint(colorResource(R.color.back))
                    )
                }


                Text(
                    text = "Settings",
                    fontSize = 32.sp,
                    fontWeight = FontWeight.Bold,
                    color = Color.White,
                    modifier = Modifier.align(Alignment.Center)
                )
            }


            Spacer(modifier = Modifier.height(16.dp))
```

ДОДАТОК В

Код програми Spring Boot

Сторінок 18

Київ-2025


```
@Configuration
class AppConfig {

    @Bean
    fun retrofit(): Retrofit {
        return Retrofit.Builder()
            .baseUrl("https://api.openweathermap.org/")
            .client(OkHttpClient())
            .addConverterFactory(GsonConverterFactory.create())
            .build()
    }

    @Bean
    fun okHttpClient(): OkHttpClient {
        return OkHttpClient.Builder().build()
    }

    @Bean
    fun weatherApiService(retrofit: Retrofit): WeatherApiService {
        return retrofit.create(WeatherApiService::class.java)
    }
}
@RestController
@RequestMapping("/api")
class BiController(
    private val routeCalculator: RouteCalculator,
    private val routeSaveService: RouteSaveService
) {

    @PostMapping("/babak")
    fun calculateRoute(@RequestBody routeRequest: RouteRequest):
ResponseEntity<Any> {
        if (routeRequest.cities.size < 2) {
            return ResponseEntity(HttpStatus.NO_CONTENT)
        }

        val journeyResponse = routeCalculator.calculateJourney(routeRequest)
        routeSaveService.saveRoute(routeRequest, journeyResponse)
        return ResponseEntity(journeyResponse, HttpStatus.OK)
    }
}
class JourneyStats {
    private val allTravel = mutableListOf<LatLng>()
    private var allMarkers = listOf<Any>()
    private val totalWeatherMarkers = mutableListOf<WeatherMarker>()
    private var totalTemp = 0.0
    private var totalCloudiness = 0.0
    private var totalWindSpeed = 0.0
    private var totalHumidity = 0.0
```

```
    private var totalVisibility = 0.0
    private var totalWindGust = 0.0
    private var totalWindDeg = 0.0

    private var maxTemp = Double.MIN_VALUE
    private var minTemp = Double.MAX_VALUE
    private var maxCloudiness = Long.MIN_VALUE
    private var minCloudiness = Long.MAX_VALUE
    private var maxWindSpeed = Double.MIN_VALUE
    private var minWindSpeed = Double.MAX_VALUE
    private var maxHumidity = Long.MIN_VALUE
```



```
    private var minHumidity = Long.MAX_VALUE

    private var maxVisibility = Long.MIN_VALUE
    private var minVisibility = Long.MAX_VALUE
    private var maxWindGust = Double.MIN_VALUE
    private var minWindGust = Double.MAX_VALUE
    private var maxWindDeg = Long.MIN_VALUE
    private var minWindDeg = Long.MAX_VALUE

    private val weatherIds = mutableListOf<Long>()
    private val mainConditions = mutableListOf<String>()
    private val descriptions = mutableListOf<String>()
    private val icons = mutableListOf<String>()

    private var totalHours = 0
    private var totalMinutes = 0
    private var totalDistance = 0.0

    fun accumulate(response: ResponseRoute) {
        val (hours, minutes) = parseDuration(response.mapResponse.duration)
        totalHours += hours
        totalMinutes += minutes
        if (totalMinutes >= 60) {
            totalHours += totalMinutes / 60
            totalMinutes %= 60
        }

        totalDistance += parseDistance(response.mapResponse.distance)

        allTravel.addAll(response.mapResponse.path)
        allMarkers += response.mapResponse.markers
        totalTemp += response.analysis.averageTemp
        totalCloudiness += response.analysis.averageCloudiness
        totalWindSpeed += response.analysis.averageWindSpeed
        totalHumidity += response.analysis.averageHumidity
        totalVisibility += response.analysis.averageVisibility
        totalWindGust += response.analysis.averageWindGust
        totalWindDeg += response.analysis.averageWindDeg
        totalWeatherMarkers += response.analysis.marketWeather

        maxTemp = maxOf(maxTemp, response.analysis.maxTemp)
        minTemp = minOf(minTemp, response.analysis.minTemp)
        maxCloudiness = maxOf(maxCloudiness, response.analysis.maxCloudiness)
        minCloudiness = minOf(minCloudiness, response.analysis.minCloudiness)
        maxWindSpeed = maxOf(maxWindSpeed, response.analysis.maxWindSpeed)
```

```
        minWindSpeed = minOf(minWindSpeed, response.analysis.minWindSpeed)
        maxHumidity = maxOf(maxHumidity, response.analysis.maxHumidity)
        minHumidity = minOf(minHumidity, response.analysis.minHumidity)
        maxVisibility = maxOf(maxVisibility, response.analysis.maxVisibility)
        minVisibility = minOf(minVisibility, response.analysis.minVisibility)
        maxWindGust = maxOf(maxWindGust, response.analysis.maxWindGust)
        minWindGust = minOf(minWindGust, response.analysis.minWindGust)
        maxWindDeg = maxOf(maxWindDeg, response.analysis.maxWindDeg)
        minWindDeg = minOf(minWindDeg, response.analysis.minWindDeg)

        weatherIds.add(response.analysis.mostFrequentWeatherId)
        mainConditions.add(response.analysis.mostFrequentMainCondition)
        descriptions.add(response.analysis.mostFrequentDescription)
        icons.add(response.analysis.mostFrequentIcon)
```



```
    }

    fun toJourneyResponse(dangerRisk: Int, comfortRisk: Int, citiesSize: Int):
JourneyResponse {
        val avgTemp = totalTemp / citiesSize
        val avgCloudiness = totalCloudiness / citiesSize
        val avgWindSpeed = totalWindSpeed / citiesSize
        val avgHumidity = totalHumidity / citiesSize
        val avgVisibility = totalVisibility / citiesSize
        val avgWindGust = totalWindGust / citiesSize
        val avgWindDeg = totalWindDeg / citiesSize


        val mostFrequentWeatherId = weatherIds.groupingBy { it
}.eachCount().maxByOrNull { it.value }?.key
        val mostFrequentMainCondition = mainConditions.groupingBy { it
}.eachCount().maxByOrNull { it.value }?.key
        val mostFrequentDescription = descriptions.groupingBy { it
}.eachCount().maxByOrNull { it.value }?.key
        val mostFrequentIcon = icons.groupingBy { it }.eachCount().maxByOrNull {
it.value }?.key

        return JourneyResponse(
            allTravel = allTravel,
            allTime = formatDuration(totalHours, totalMinutes),
            allDistance = formatDistance(totalDistance),
            averageTemp = BigDecimal(avgTemp).setScale(2,
RoundingMode.HALF_UP).toDouble(),
            maxTemp = maxTemp,
            minTemp = minTemp,
            averageCloudiness = BigDecimal(avgCloudiness).setScale(2,
RoundingMode.HALF_UP).toDouble(),
            maxCloudiness = maxCloudiness,
            minCloudiness = minCloudiness,
            averageWindSpeed = BigDecimal(avgWindSpeed).setScale(2,
RoundingMode.HALF_UP).toDouble(),
            maxWindSpeed = maxWindSpeed,
            minWindSpeed = minWindSpeed,
            averageHumidity = BigDecimal(avgHumidity).setScale(2,
RoundingMode.HALF_UP).toDouble(),
            maxHumidity = maxHumidity,
            minHumidity = minHumidity,
            averageVisibility = avgVisibility,
            maxVisibility = maxVisibility,
```

```
            minVisibility = minVisibility,
            averageWindGust = BigDecimal(avgWindGust).setScale(2,
RoundingMode.HALF_UP).toDouble(),
            maxWindGust = maxWindGust,
            minWindGust = minWindGust,
            averageWindDeg = BigDecimal(avgWindDeg).setScale(2,
RoundingMode.HALF_UP).toDouble(),
            maxWindDeg = maxWindDeg,
            minWindDeg = minWindDeg,
            mostFrequentWeatherId = mostFrequentWeatherId,
            mostFrequentMainCondition = mostFrequentMainCondition,
            mostFrequentDescription = mostFrequentDescription,
            mostFrequentIcon = mostFrequentIcon,
            totalWeatherMarkers = totalWeatherMarkers,
            dangerRisk = dangerRisk,
            comfortRisk = comfortRisk
```



```
        )
    }

    private fun parseDuration(duration: String): Pair<Int, Int> {
        val regex = Regex("(\\d+)h\\s*(\\d+)m")
        val matchResult = regex.find(duration)
        return if (matchResult != null) {
            Pair(matchResult.groups[1]?.value?.toInt() ?: 0,
matchResult.groups[2]?.value?.toInt() ?: 0)
        } else {
            Pair(0, 0)
        }
    }

    private fun parseDistance(distance: String): Double {
        val regex = Regex("(\\d+\\.?\\d*)\\s*km")
        val matchResult = regex.find(distance)
        return matchResult?.groups?.get(1)?.value?.toDouble() ?: 0.0
    }

    private fun formatDistance(totalDistance: Double): String {
        return "${"%.2f".format(totalDistance)} km"
    }

    private fun formatDuration(totalHours: Int, totalMinutes: Int): String {
        return "${totalHours}h ${totalMinutes}m"
    }
}
object RouteUtils {

    fun decodePoly(encoded: String): List<LatLng> {
        val poly = mutableListOf<LatLng>()
        var index = 0
        val len = encoded.length
        var lat = 0
        var lng = 0

        while (index < len) {
            var b: Int
            var shift = 0
            var result = 0
            do {
```

```
                b = encoded[index++].code - 63
                result = result or (b and 0x1f shl shift)
                shift += 5
            } while (b >= 0x20)
            val dlat = if (result and 1 != 0) (result shr 1).inv() else result
shr 1
            lat += dlat

            shift = 0
            result = 0
            do {
                b = encoded[index++].code - 63
                result = result or (b and 0x1f shl shift)
                shift += 5
            } while (b >= 0x20)
            val dlng = if (result and 1 != 0) (result shr 1).inv() else result
shr 1
            lng += dlng
```



```
            val p = LatLng((lat.toDouble() / 1E5), (lng.toDouble() / 1E5))
            poly.add(p)
        }

        return poly
    }

    fun betweenTime(startTime: String, endTime: String): String {
        val formatter = SimpleDateFormat("HH:mm dd.MM.yyyy",
Locale.getDefault())
        val startTime = formatter.parse(startTime) ?: Date()
        val endTime = formatter.parse(endTime) ?: Date()
        val duration = endTime.time - startTime.time
        val hours = duration / (1000 * 60 * 60)
        val minutes = (duration % (1000 * 60 * 60)) / (1000 * 60)
        return "${hours}h ${minutes}m"
    }

    fun calculateDistance(start: LatLng, end: LatLng): Double {
        val earthRadius = 6371000.0 // meters
        val dLat = Math.toRadians(end.latitude - start.latitude)
        val dLng = Math.toRadians(end.longitude - start.longitude)
        val a = Math.sin(dLat / 2) * Math.sin(dLat / 2) +
                Math.cos(Math.toRadians(start.latitude)) *
Math.cos(Math.toRadians(end.latitude)) *
                Math.sin(dLng / 2) * Math.sin(dLng / 2)
        val c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a))
        return earthRadius * c
    }

    fun getMarkersEveryDistance(
        leg: JSONObject,
        intervalMeters: Double,
        averageSpeedKmh: Double,
        startTime: String
    ): List<Pair<LatLng, String>> {
        val steps = leg.getJSONArray("steps")
        val markers = mutableListOf<Pair<LatLng, String>>()
        var accumulatedDistance = 0.0

        val startLocation = LatLng(
```

```
            leg.getJSONObject("start_location").getDouble("lat"),
            leg.getJSONObject("start_location").getDouble("lng")
        )
        markers.add(Pair(startLocation, startTime))

        val sdf = SimpleDateFormat("HH:mm dd.MM.yyyy", Locale.getDefault())
        val calendar = Calendar.getInstance()
        calendar.time = sdf.parse(startTime) ?: Date()

        for (i in 0 until steps.length()) {
            val step = steps.getJSONObject(i)
            val distance = step.getJSONObject("distance").getDouble("value")
            val polyline = step.getJSONObject("polyline").getString("points")
            val stepPath = decodePoly(polyline)

            for (j in 1 until stepPath.size) {
                val prevPoint = stepPath[j - 1]
                val currentPoint = stepPath[j]
                val segmentDistance = calculateDistance(prevPoint, currentPoint)
```



```
                accumulatedDistance += segmentDistance
                if (accumulatedDistance >= intervalMeters) {
                    val timeToTravelSeconds = (intervalMeters / 1000.0) /
averageSpeedKmh * 3600
                    calendar.add(Calendar.SECOND, timeToTravelSeconds.toInt())
                    val timeText = sdf.format(calendar.time).toString()
                    markers.add(Pair(currentPoint, timeText))
                    accumulatedDistance -= intervalMeters
                }
            }
        }

        if (accumulatedDistance > 0) {
            val timeToTravelSeconds = (accumulatedDistance / 1000.0) /
averageSpeedKmh * 3600
            calendar.add(Calendar.SECOND, timeToTravelSeconds.toInt())
            val endTimeText = sdf.format(calendar.time).toString()
            val endLocation = LatLng(
                leg.getJSONObject("end_location").getDouble("lat"),
                leg.getJSONObject("end_location").getDouble("lng")
            )

            markers.add(Pair(endLocation, endTimeText))
        }

        return markers
    }
}
class AnalyzeWeather {

    fun analyzeWeather(
        responses: List<WeatherResponseByTime>,
        marketWeather: List<WeatherMarker>
    ): WeatherAnalysis {

        //Температура
        val temperatureSummary = responses.flatMap { it.data.map { daum ->
daum.temp } }
        val averageTemp = temperatureSummary.average()
```

```
        val maxTemp = temperatureSummary.maxOrNull() ?: 0.0
        val minTemp = temperatureSummary.minOrNull() ?: 0.0

        //Хмарність, %
        val cloudinessSummary = responses.flatMap { it.data.map { daum ->
daum.clouds } }
        val averageCloudiness = cloudinessSummary.average().toLong()
        val maxCloudiness = cloudinessSummary.maxOrNull() ?: 0L
        val minCloudiness = cloudinessSummary.minOrNull() ?: 0L

        //Швидкість вітру
        val windSpeedSummary = responses.flatMap { it.data.map { daum ->
daum.wind_speed } }
        val averageWindSpeed = windSpeedSummary.average()
        val maxWindSpeed = windSpeedSummary.maxOrNull() ?: 0.0
        val minWindSpeed = windSpeedSummary.minOrNull() ?: 0.0

        //Вологість, %
        val humiditySummary = responses.flatMap { it.data.map { daum ->
daum.humidity } }
        val averageHumidity = humiditySummary.average().toLong()
```



```
        val maxHumidity = humiditySummary.maxOrNull() ?: 0L
        val minHumidity = humiditySummary.minOrNull() ?: 0L

        //Середня видимість, метри
        val visibilitySummary = responses.flatMap { it.data.map { daum ->
daum.visibility } }
        val averageVisibility = visibilitySummary.average().toLong()
        val maxVisibility = visibilitySummary.maxOrNull() ?: 10000L
        val minVisibility = visibilitySummary.minOrNull() ?: 0L

        //(якщо є) Порив вітру. Швидкість вітру
        val windGustSummary = responses.flatMap { it.data.map { daum ->
daum.wind_gust } }
        val averageWindGust = windGustSummary.average()
        val maxWindGust = windGustSummary.maxOrNull() ?: 0.0
        val minWindGust = windGustSummary.minOrNull() ?: 0.0

        //Напрямок вітру, градуси (метеорологічний)
        val windDegSummary = responses.flatMap { it.data.map { daum ->
daum.wind_deg } }
        val averageWindDeg = windDegSummary.average().toLong()
        val maxWindDeg = windDegSummary.maxOrNull() ?: 0L
        val minWindDeg = windDegSummary.minOrNull() ?: 0L

        //Weather
        val weatherData = responses.flatMap { it.data.flatMap { daum ->
daum.weather } }
        val mostFrequentWeatherId = weatherData.groupingBy { it.id
}.eachCount().maxByOrNull { it.value }?.key ?: 0L
        val mostFrequentMainCondition = weatherData.groupingBy { it.main
}.eachCount().maxByOrNull { it.value }?.key ?: "Unknown"
        val mostFrequentDescription = weatherData.groupingBy { it.description
}.eachCount().maxByOrNull { it.value }?.key ?: "Unknown"
        val mostFrequentIcon = weatherData.groupingBy { it.icon
}.eachCount().maxByOrNull { it.value }?.key ?: "Unknown"

        return WeatherAnalysis(
```

```
            averageTemp = averageTemp,
            maxTemp = maxTemp,
            minTemp = minTemp,
            averageCloudiness = averageCloudiness,
            maxCloudiness = maxCloudiness,
            minCloudiness = minCloudiness,
            averageWindSpeed = averageWindSpeed,
            maxWindSpeed = maxWindSpeed,
            minWindSpeed = minWindSpeed,
            averageHumidity = averageHumidity,
            maxHumidity = maxHumidity,
            minHumidity = minHumidity,
            averageVisibility = averageVisibility,
            maxVisibility = maxVisibility,
            minVisibility = minVisibility,
            averageWindGust = averageWindGust,
            maxWindGust = maxWindGust,
            minWindGust = minWindGust,
            averageWindDeg = averageWindDeg,
            maxWindDeg = maxWindDeg,
            minWindDeg = minWindDeg,
            mostFrequentWeatherId = mostFrequentWeatherId,
            mostFrequentMainCondition= mostFrequentMainCondition,
            mostFrequentDescription = mostFrequentDescription,
```



```
            mostFrequentIcon = mostFrequentIcon,
            marketWeather = marketWeather
        )
    }
}
interface WeatherApiService {

//https://api.openweathermap.org/data/3.0/onecall/timemachine?lat={lat}&lon={lon
}&dt={time}&appid={API key}
    @GET("data/3.0/onecall/timemachine")
    fun getWeatherByDate(
        @Query("lat") lat: Double,
        @Query("lon") lon: Double,
        @Query("appid") appId: String,
        @Query("lang") lang: String,
        @Query("units") units: String, // imperial - US || metric - All
        @Query("dt") dt: Long
    ): Call<WeatherResponseByTime>
}
interface CheckpointRepository:
JpaRepository<com.radchuk.bitreker.model.db.entity.Checkpoint, Long>
interface CheckpointWeatherRepository:
JpaRepository<com.radchuk.bitreker.model.db.entity.CheckpointWeather, Long>
interface JourneyWeatherSummaryRepository: JpaRepository<com.ra interface
RouteParamsRepository:
JpaRepository<com.radchuk.bitreker.model.db.entity.RouteParams, Long>
dchuk.bitreker.model.db.entity.JourneyWeatherSummary, Long>
interface RoutePointRepository:
JpaRepository<com.radchuk.bitreker.model.db.entity.RoutePoint, Long>
interface RouteStopsRepository: JpaRepository<RouteStops, Long>
interface StopRepository: JpaRepository<Stop, Long>
interface StopRepository: JpaRepository<Stop, Long>
interface WeatherRepository:
JpaRepository<com.radchuk.bitreker.model.db.entity.Weather, Long>
@Service
class BiService(
```

```
    private val client: OkHttpClient,
    private val weatherService: WeatherService
) {
    private val logger: Logger = LoggerFactory.getLogger(BiService::class.java)

    @Value("\${keyAPI.maps}")
    private lateinit var apiKey: String

    fun getRouteResponse(
        startCity: String,
        endCity: String,
        startTime: String,
        stepKM: Double,
        speed: Double,
        metricWeather: String
    ): ResponseRoute {
        val mapsJson = fetchRouteData(startCity, endCity)
        return parseRoute(mapsJson, startTime, stepKM, speed, metricWeather)
    }

    private fun fetchRouteData(startCity: String, endCity: String): String {
        val url = "https://maps.googleapis.com/maps/api/directions/json?" +
                "origin=$startCity&destination=$endCity&key=$apiKey"

        val request = Request.Builder().url(url).build()
```



```
        return client.newCall(request).execute().use { response ->
            response.body?.string().orEmpty()
        }
    }

    private fun parseRoute(
        json: String,
        startTime: String,
        stepKM: Double,
        speed: Double,
        metricWeather: String
    ): ResponseRoute {
        val jsonObject = JSONObject(json)
        val route = jsonObject.getJSONArray("routes").getJSONObject(0)
        val encodedPath =
route.getJSONObject("overview_polyline").getString("points")
        val path = RouteUtils.decodePoly(encodedPath)

        val legs = route.getJSONArray("legs").getJSONObject(0)
        val markers = RouteUtils.getMarkersEveryDistance(legs, stepKM, speed,
startTime)
        val weatherMarkers = getWeatherMarkers(markers, metricWeather)



        val distance = legs.getJSONObject("distance").getString("text")
        val duration = RouteUtils.betweenTime(startTime, markers.last().second)
        val analysis = AnalyzeWeather().analyzeWeather(weatherMarkers.map {
it.weatherInfo }, weatherMarkers)

        logger.info("Markers: $markers")
        logger.info("Weather Markers: $weatherMarkers")
        logger.info("Analysis: $analysis")

        return ResponseRoute(
```

```
            mapResponse = MapResponse(
                path = path,
                distance = distance,
                duration = duration,
                markers = markers
            ),
            analysis = analysis
        )
    }

    private fun getWeatherMarkers(
        markers: List<Pair<LatLng, String>>,
        metricWeather: String
    ): List<WeatherMarker> {
        return markers.mapNotNull { (location, timestamp) ->
            weatherService.getWeather(location.latitude, location.longitude,
stringToUnixTime(timestamp), metricWeather)
                ?.let { WeatherMarker(location, timestamp, it) }
        }
    }
}
@Service
class RiskCalculator {

    fun calculateRisk(response: JourneyResponse, units: String): RiskAssessment
{
```



```
        val weatherId = response.mostFrequentWeatherId ?: 0L

        val dangerRisk = when (weatherId) {
            in 200L..232L, in 502L..504L, 522L, 602L, 622L, 781L -> 100
            511L, 601L, 621L, 771L, 731L, 761L -> 80
            501L, 600L, 741L, 721L -> 60
            500L, in 300L..321L, in 800L..804L -> 40
            else -> 20
        }

        val temperature = if (units == "imperial") (response.averageTemp - 32) *
5 / 9 else response.averageTemp
        val windSpeed = if (units == "imperial") response.averageWindSpeed *
0.44704 else response.averageWindSpeed

        val comfortRisk = when {
            temperature < -20 || temperature > 45 -> 100
            temperature < -15 || temperature > 40 -> 90
            temperature < -10 || temperature > 35 -> 80
            response.averageHumidity > 90 || windSpeed > 25 -> 75
            response.averageHumidity > 85 || windSpeed > 20 -> 70
            response.averageHumidity > 80 || windSpeed > 15 -> 60
            weatherId in 500L..531L || weatherId in 600L..622L || weatherId in
700L..799L || weatherId == 804L -> 50
            weatherId in 300L..321L || weatherId in 800L..803L -> 30
            else -> 20
        }

        return RiskAssessment(dangerRisk, comfortRisk)
    }
}
@Service
class RouteCalculator(
    private val biService: BiService,
```

```
    private val riskCalculator: RiskCalculator
) {

    fun calculateJourney(routeRequest: RouteRequest): JourneyResponse {
        val cities = routeRequest.cities
        val journeyStats = JourneyStats()

        var startTimeCity = routeRequest.departureTime

        for (i in 0 until cities.size - 1) {
            val stop = cities[i].stopDuration
            logger.info("startTime for segment $i =
${addTimeToDateTime(startTimeCity, stop)}")
            val response = biService.getRouteResponse(
                cities[i].cityName, cities[i + 1].cityName,
addTimeToDateTime(startTimeCity, stop),
                routeRequest.stepDistance, routeRequest.averageSpeed,
routeRequest.metricWeather
            )
            startTimeCity = response.mapResponse.markers.last().second

            journeyStats.accumulate(response)
        }

        val journeyResponse = journeyStats.toJourneyResponse(0,0, cities.size -
1)
```



```


        val riskAssessment = riskCalculator.calculateRisk(journeyResponse,
routeRequest.metricWeather)

        return journeyResponse.copy(
            dangerRisk = riskAssessment.dangerRisk,
            comfortRisk = riskAssessment.comfortRisk
        )
    }

    private fun addTimeToDateTime(dateTimeStr: String, timeToAdd: String):
String {
        val dateTimeFormatter = DateTimeFormatter.ofPattern("HH:mm dd.MM.yyyy")
        var dateTime = LocalDateTime.parse(dateTimeStr, dateTimeFormatter)
        val timeParts = timeToAdd.split(":")
        dateTime =
dateTime.plusHours(timeParts[0].toLong()).plusMinutes(timeParts[1].toLong())
        return dateTime.format(dateTimeFormatter)
    }
}
@Service
class RouteSaveService(
    private val tripRepository: TripRepository,
    private val routeParamsRepository: RouteParamsRepository,
    private val stopRepository: StopRepository,
    private val routeStopsRepository: RouteStopsRepository,
    private val checkpointRepository: CheckpointRepository,
    private val weatherRepository: WeatherRepository,
    private val checkpointWeatherRepository: CheckpointWeatherRepository,
    private val journeyWeatherSummaryRepository:
JourneyWeatherSummaryRepository,
    private val routePointRepository: RoutePointRepository
```

```
) {
    fun saveRoute(routeRequest: RouteRequest, journeyResponse: JourneyResponse)
{
        val trip = com.radchuk.bitreker.model.db.entity.Trip(
            startPoint = routeRequest.cities.first().cityName,
            endPoint = routeRequest.cities.last().cityName,
            dangerRisk = journeyResponse.dangerRisk,
            comfortRisk = journeyResponse.comfortRisk
        )
        val savedTrip = tripRepository.save(trip)

        val formatter = DateTimeFormatter.ofPattern("HH:mm dd.MM.yyyy")
        val localDateTime = LocalDateTime.parse(routeRequest.departureTime,
formatter)

        val routeParams = com.radchuk.bitreker.model.db.entity.RouteParams(
            avgSpeed = routeRequest.averageSpeed,
            checkInterval = routeRequest.stepDistance.toInt(),
            startDatetime = localDateTime,
            metricWeather = routeRequest.metricWeather,
            trip = savedTrip
        )
        routeParamsRepository.save(routeParams)

        routeRequest.cities.forEach { city ->
            val stop = Stop(
                address = city.cityName,
                waitingTime = city.stopDuration.toIntOrNull() ?: 0,
```



```
                lat = 0.0,
                lng = 0.0
            )
            val savedStop = stopRepository.save(stop)

            val routeStop = RouteStops(
                stop = savedStop,
                route = savedTrip
            )
            routeStopsRepository.save(routeStop)
        }

        journeyResponse.allTravel.forEachIndexed { index, latLng ->
            val point = com.radchuk.bitreker.model.db.entity.RoutePoint(
                trip = savedTrip,
                latitude = latLng.latitude,
                longitude = latLng.longitude,
                pointOrder = index
            )
            routePointRepository.save(point)
        }

        val journeySummary =
com.radchuk.bitreker.model.db.entity.JourneyWeatherSummary(
            trip = savedTrip,
            averageTemp = journeyResponse.averageTemp,
            maxTemp = journeyResponse.maxTemp,
            minTemp = journeyResponse.minTemp,
            averageCloudiness = journeyResponse.averageCloudiness,
            maxCloudiness = journeyResponse.maxCloudiness,
```

```
            minCloudiness = journeyResponse.minCloudiness,
            averageWindSpeed = journeyResponse.averageWindSpeed,
            maxWindSpeed = journeyResponse.maxWindSpeed,
            minWindSpeed = journeyResponse.minWindSpeed,
            averageHumidity = journeyResponse.averageHumidity,
            maxHumidity = journeyResponse.maxHumidity,
            minHumidity = journeyResponse.minHumidity,
            averageVisibility = journeyResponse.averageVisibility,
            maxVisibility = journeyResponse.maxVisibility,
            minVisibility = journeyResponse.minVisibility,
            averageWindGust = journeyResponse.averageWindGust,
            maxWindGust = journeyResponse.maxWindGust,
            minWindGust = journeyResponse.minWindGust,
            averageWindDeg = journeyResponse.averageWindDeg,
            maxWindDeg = journeyResponse.maxWindDeg,
            minWindDeg = journeyResponse.minWindDeg,
            mostFrequentWeatherId = journeyResponse.mostFrequentWeatherId,
            mostFrequentMainCondition =
journeyResponse.mostFrequentMainCondition,
            mostFrequentDescription = journeyResponse.mostFrequentDescription,
            mostFrequentIcon = journeyResponse.mostFrequentIcon
        )
        journeyWeatherSummaryRepository.save(journeySummary)

        journeyResponse.totalWeatherMarkers.forEach { marker ->
            val localMarkerTime = LocalDateTime.parse(marker.timestamp,
formatter)
            val checkpoint = com.radchuk.bitreker.model.db.entity.Checkpoint(
                arrivalTime = localMarkerTime,
                trip = savedTrip
            )
```



```
            val savedCheckpoint = checkpointRepository.save(checkpoint)

            marker.weatherInfo.data.firstOrNull()?.let { daum ->
                val weather = com.radchuk.bitreker.model.db.entity.Weather(
                    state = daum.weather.firstOrNull()?.main ?: "Unknown",
                    temperature = daum.temp,
                    feelsLike = daum.feels_like,
                    pressure = daum.pressure,
                    humidity = daum.humidity,
                    dewPoint = daum.dew_point,
                    uvi = daum.uvi,
                    clouds = daum.clouds,
                    visibility = daum.visibility,
                    windSpeed = daum.wind_speed,
                    windGust = daum.wind_gust,
                    windDeg = daum.wind_deg,
                    description = daum.weather.firstOrNull()?.description,
                    icon = daum.weather.firstOrNull()?.icon
                )
                val savedWeather = weatherRepository.save(weather)

                val checkpointWeather =
com.radchuk.bitreker.model.db.entity.CheckpointWeather(
                    checkpoint = savedCheckpoint,
                    weather = savedWeather
                )
                checkpointWeatherRepository.save(checkpointWeather)
            }
```

```
        }
    }
}
@Service
class WeatherService(
    private val weatherApiService: WeatherApiService
) {

    @Value("\${keyAPI.openweather}")
    lateinit var openWeatherApiKey: String

    fun getWeather(lat: Double, lon: Double, time: Long, metric : String):
WeatherResponseByTime? {
        val call = weatherApiService.getWeatherByDate(
            lat,
            lon,
            openWeatherApiKey,
            "en",
            metric,
            time
        )
        val response = call.execute()
        return if (response.isSuccessful) response.body() else null
    }
}
// Функція для перетворення рядка в Unix час
fun stringToUnixTime(dateString: String): Long {
    val dateFormat = SimpleDateFormat("HH:mm dd.MM.yyyy", Locale.getDefault())
    val date = dateFormat.parse(dateString)
    return date?.time?.div(1000) ?: 0L
}
```



```
// Функція для перетворення Unix часу в рядок
fun unixTimeToString(unixTime: Long): String {
    val dateFormat = SimpleDateFormat("HH:mm dd.MM.yyyy", Locale.getDefault())
    val date = java.util.Date(unixTime * 1000)
    return dateFormat.format(date)
}
class ResponseRoute (
    val mapResponse: MapResponse,
    val analysis: WeatherAnalysis
)
data class WeatherResponseByTime (
    val lat: Double,
    val lon: Double,
    val timezone: String,
    val timezone_offset: Long,
    val data: List<Daum>,
)
data class Weather(
    val id: Long,
    val main: String,
    val description: String,
    val icon: String,
)
data class Daum(
    val dt: Long,
    val sunrise: Long,
    val sunset: Long,
    val temp: Double,
```

```
    val feels_like: Double,
    val pressure: Long,
    val humidity: Long,
    val dew_point: Double,
    val uvi: Double,
    val clouds: Long,
    val visibility: Long,
    val wind_speed: Double,
    val wind_gust: Double,
    val wind_deg: Long,
    val weather: List<Weather>,
)
data class WeatherAnalysis(

    val averageTemp: Double,
    val maxTemp: Double,
    val minTemp: Double,

    val averageCloudiness: Long,
    val maxCloudiness: Long,
    val minCloudiness: Long,

    val averageWindSpeed: Double,
    val maxWindSpeed: Double,
    val minWindSpeed: Double,

    val averageHumidity: Long,
    val maxHumidity: Long,
    val minHumidity: Long,

    val averageVisibility: Long,
    val maxVisibility: Long,
```



```
    val minVisibility: Long,

    val averageWindGust: Double,
    val maxWindGust: Double,
    val minWindGust: Double,

    val averageWindDeg: Long,
    val maxWindDeg: Long,
    val minWindDeg: Long,

    val mostFrequentWeatherId: Long,
    val mostFrequentMainCondition: String,
    val mostFrequentDescription: String,
    val mostFrequentIcon: String,

    val marketWeather: List<WeatherMarker>
)
data class WeatherMarker(
    val location: LatLng,
    val timestamp: String,
    val weatherInfo: WeatherResponseByTime
)
data class JourneyResponse (
    val allTravel: List<LatLng>,
    val allTime: String,
    val allDistance: String,
```

```
    val averageTemp: Double,
    val maxTemp: Double,
    val minTemp: Double,
    val averageCloudiness: Double,
    val maxCloudiness: Long,
    val minCloudiness: Long,
    val averageWindSpeed: Double,
    val maxWindSpeed: Double,
    val minWindSpeed: Double,
    val averageHumidity: Double,
    val maxHumidity: Long,
    val minHumidity: Long,
    val averageVisibility: Double,
    val maxVisibility: Long,
    val minVisibility: Long,
    val averageWindGust: Double,
    val maxWindGust: Double,
    val minWindGust: Double,
    val averageWindDeg: Double,
    val maxWindDeg: Long,
    val minWindDeg: Long,
    val mostFrequentWeatherId: Long?,
    val mostFrequentMainCondition: String?,
    val mostFrequentDescription: String?,
    val mostFrequentIcon: String?,
    val dangerRisk: Int,
    val comfortRisk: Int,
    val totalWeatherMarkers: List<WeatherMarker>
)
data class RouteRequest(
    val cities: List<CityStop>,
    val departureTime: String,
    val stepDistance: Double,
```



```
    val averageSpeed: Double,
    val metricWeather: String
)
data class CityStop(
    val cityName: String,
    val stopDuration: String
)
data class MapResponse(
    val path: List<LatLng>,
    val distance: String,
    val duration: String,
    val markers: List<Pair<LatLng, String>>
)
data class LatLng(
    val latitude: Double,
    val longitude: Double
)
@Entity
@Table(name = "Checkpoint")
data class Checkpoint(
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    val markerId: Long = 0,

    @Column(nullable = false)
```

```
    val arrivalTime: LocalDateTime,

    @ManyToOne
    @JoinColumn(name = "route_id")
    val trip: com.radchuk.bitreker.model.db.entity.Trip
)
@Entity
@Table(name = "CheckpointWeather")
data class CheckpointWeather(
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    val id: Long = 0,

    @ManyToOne
    @JoinColumn(name = "marker_id")
    val checkpoint: com.radchuk.bitreker.model.db.entity.Checkpoint,

    @ManyToOne
    @JoinColumn(name = "weather_id")
    val weather: com.radchuk.bitreker.model.db.entity.Weather
)
@Entity
@Table(name = "journey_weather_summary")
data class JourneyWeatherSummary(
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    val summaryId: Long = 0,

    @OneToOne
    @JoinColumn(name = "trip_id")
    val trip: com.radchuk.bitreker.model.db.entity.Trip,

    val averageTemp: Double?,
    val maxTemp: Double?,
    val minTemp: Double?,
    val averageCloudiness: Double?,
    val maxCloudiness: Long?,
```



```
    val minCloudiness: Long?,
    val averageWindSpeed: Double?,
    val maxWindSpeed: Double?,
    val minWindSpeed: Double?,
    val averageHumidity: Double?,
    val maxHumidity: Long?,
    val minHumidity: Long?,
    val averageVisibility: Double?,
    val maxVisibility: Long?,
    val minVisibility: Long?,
    val averageWindGust: Double?,
    val maxWindGust: Double?,
    val minWindGust: Double?,
    val averageWindDeg: Double?,
    val maxWindDeg: Long?,
    val minWindDeg: Long?,
    val mostFrequentWeatherId: Long?,
    val mostFrequentMainCondition: String?,
    val mostFrequentDescription: String?,
    val mostFrequentIcon: String?
)
@Entity
@Table(name = "route_params")
data class RouteParams(
```

```
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    val paramsId: Long = 0,

    val avgSpeed: Double,
    val checkInterval: Int,
    val startDatetime: LocalDateTime,
    val metricWeather: String? = null,

    @OneToOne
    @JoinColumn(name = "route_id")
    val trip: com.radchuk.bitreker.model.db.entity.Trip
)
@Entity
@Table(name = "route_points")
data class RoutePoint(
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    val pointId: Long = 0,

    @ManyToOne
    @JoinColumn(name = "trip_id")
    val trip: com.radchuk.bitreker.model.db.entity.Trip,

    val latitude: Double,
    val longitude: Double,
    val pointOrder: Int
)
@Entity
@Table(name = "RouteStops")
data class RouteStops(
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    val id: Long = 0,

    @ManyToOne
    @JoinColumn(name = "stop_id")
```



```
    val stop: Stop,

    @ManyToOne
    @JoinColumn(name = "route_id")
    val route: com.radchuk.bitreker.model.db.entity.Trip
)
@Entity
@Table(name = "Stop")
data class Stop(
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    val stopId: Long = 0,

    @Column(nullable = false)
    val address: String,

    @Column
    val waitingTime: Int,

    @Column(nullable = false)
    val lat: Double,

    @Column(nullable = false)
```

```
    val lng: Double
)
@Entity
@Table(name = "trip")
data class Trip(
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    val routeId: Long = 0,

    @Column(nullable = false)
    val startPoint: String,

    @Column(nullable = false)
    val endPoint: String,

    val dangerRisk: Int? = null,
    val comfortRisk: Int? = null
)
@Entity
@Table(name = "weather")
data class Weather(
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    val weatherId: Long = 0,

    val state: String,
    val temperature: Double,
    val feelsLike: Double?,
    val pressure: Long?,
    val humidity: Long?,
    val dewPoint: Double?,
    val uvi: Double?,
    val clouds: Long?,
    val visibility: Long?,
    val windSpeed: Double?,
    val windGust: Double?,
    val windDeg: Long?,
    val description: String?,
    val icon: String?
)
```