МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему: **«РОЗРОБКА ІНТЕРАКТИВНОЇ СИСТЕМИ ПЛАНУВАННЯ МАРШРУТІВ»**

Виконав: студент 4 курсу, групи 6.1212-пі-3
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми програмна інженерія
(назва освітньої програми)
Я.М. Бучинський
(ініціали та прізвище)

Керівник доцент кафедри програмної інженерії, к.ф.-м.н. Мильцев О.М.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент доцент кафедри комп'ютерних наук, к.т.н. Добровольський Г.А.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2026

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти бакалавр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма програмна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2025 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Бучинському Ярославу Михайловичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка інтерактивної системи планування маршрутів

керівник роботи Мильцев Олександр Михайлович, к.ф.-м.н.
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 02 » грудня 2025 року № 1741-с

2. Строк подання студентом роботи 01.06.2026 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Технічна документація фреймворку Flutter Dart.
3. Специфікація картографічного сервісу TomTom.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Аналіз предметної області та специфікація вимог до програмного забезпечення.
2. Проєктування архітектури та моделювання системи.
3. Практична реалізація та тестування програмного продукту.

4. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою доповіді

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 09.12.2025 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	12.01.2025	
2.	Збір вихідних даних.	26.02.2026	
3.	Обробка методичних та теоретичних	06.03.2026	
	джерел.		
4.	Розробка першого та другого розділу.	06.04.2026	
5.	Розробка третього розділу.	18.05.2026	
6.	Оформлення та нормоконтроль	25.05.2026	
	кваліфікаційної роботи бакалавра.		
7.	Захист кваліфікаційної роботи.	18.06.2026	

Студент _____ Я.М. Бучинський
(підпис) (ініціали та прізвище)

Керівник роботи _____ О.М. Мильцев
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота бакалавра «Розробка інтерактивної системи планування маршрутів»: 53 с., 9 рис., 2 табл., 13 джерел, 1 додаток.

АВТОДОПОВНЕННЯ АДРЕС, ГЕОКОДУВАННЯ, КРОСПЛАТФОРМЕНА РОЗРОБКА, МОБІЛЬНА НАВІГАЦІЯ, ПЛАНУВАННЯ МАРШРУТІВ, DART, FLUTTER, JSON, REST API, TOMTOM API.

Об’єкт дослідження – процес побудови, оптимізації та інтерактивної візуалізації дорожніх маршрутів у середовищі мобільних операційних систем.

Мета роботи: проєктування, архітектурне обґрунтування та програмна реалізація кросплатформеного мобільного застосунку для розрахунку оптимальних шляхів переміщення та забезпечення покрокової навігації в реальному часі.

Методи дослідження – системний аналіз предметної області, об’єктно-орієнтоване моделювання програмних систем із застосуванням мови UML, класичні графові алгоритми маршрутизації, методи клієнт-серверної взаємодії за протоколом HTTP, асинхронна обробка геопросторових даних, а також методи ручного та інтеграційного тестування мобільних інтерфейсів.

У роботі розроблено мобільний навігатор «Zp Navigator Pro» на базі Flutter. Інтегровано сервіс TomTom API для розрахунку маршрутів і відстаней, а також реалізовано механізм динамічного відображення шляхів на інтерактивній карті.

SUMMARY

Bachelor’s qualifying paper “Development of an Interactive Route Planning System”: 53 pages, 9 figures, 2 tables, 13 references, 1 supplement.

AUTOCOMPLETE, CROSS-PLATFORM DEVELOPMENT, DART, FLUTTER, GEOCODING, JSON, MOBILE NAVIGATION, REST API, ROUTE PLANNING, TOMTOM API.

The object of the study is the process of building, optimizing, and interactively visualizing road routes within mobile operating systems.

The aim of the study is designing, architectural substantiation, and software implementation of a cross-platform mobile application for calculating optimal travel paths and providing real-time turn-by-turn navigation.

The methods of research include a systematic analysis of the subject domain, object-oriented software modeling using UML, classical graph routing algorithms, client-server interaction via HTTP, asynchronous processing of geospatial data, as well as manual and integration testing of mobile interfaces.

The paper develops the “Zp Navigator Pro” mobile application based on Flutter. The TomTom API service is integrated for calculating routes and distances, and a mechanism for dynamic path rendering on an interactive map is implemented.

ЗМІСТ

Завдання на кваліфікаційну роботу 2
Реферат 4
Summary 5
Вступ 8
1 Аналіз предметної області та специфікація вимог до програмного забезпечення 10
1.1 Дослідження концепцій сучасної мобільної навігації та геокодування 10
1.2 Порівняльний аналіз існуючих аналогів та програмних рішень 12
1.3 Аналіз апаратних та програмних обмежень мобільних систем 14
1.4 Специфікація функціональних та нефункціональних вимог 15
1.5 Аналіз проєктних ризиків та шляхи їх мінімізації 18
2 Проєктування архітектури та моделювання системи 20
2.1 Обґрунтування вибору технологічного стеку та інструментальних засобів 20
2.2 Архітектурні патерни та проєктування шарів системи 22
2.3 Об’єктно-орієнтоване моделювання системи за допомогою UML-діаграм 24
2.4 Проєктування інтерактивного користувацького інтерфейсу 26
2.5 Архітектурні патерни проєктування користувацького інтерфейсу 27
3 Практична реалізація та тестування програмного продукту 29
3.1 Розробка користувацького інтерфейсу та робота з картографічними даними 29
3.2 Інтеграція апаратних датчиків та TomTom REST API 32
3.3 Керування станом та відображення маршрутів 34
3.4 Тестування системи та компіляція релізної версії 36
3.5 Методологія та результати тестування системи 37

3.6 Компіляція та підготовка релізної версії застосунку 39
Висновки 41
Перелік посилань..... 43
Додаток А Лістинг вихідного коду мобільного застосунку 44

ВСТУП

Інтеграція готових навігаційних рішень (на кшталт Google Maps SDK) у власні проєкти часто супроводжується жорсткими лімітами API та перевантаженням пам’яті слабких смартфонів. Сучасна інженерна альтернатива – використання хмарних REST API, де важкі обчислення дорожніх графів делегуються серверу, а мобільний додаток працює як швидкий «тонкий клієнт». Проте це створює новий виклик: необхідність миттєво парсити великі JSON-масиви координат та плавно рендерити їх на екрані. Вирішенням цієї проблеми є використання кросплатформених фреймворків з прямим доступом до графічного процесора (GPU), таких як Flutter.

Актуальність теми зумовлена потребою у швидких мобільних навігаторах, які не перевантажені зайвим корпоративним функціоналом, стабільно обробляють апаратні GPS-дані та використовують оптимізовані мережеві запити для економії заряду акумулятора.

Мета роботи: проєктування та програмна реалізація кросплатформеного мобільного застосунку для розрахунку оптимальних шляхів переміщення та покрокової навігації.

Основні завдання:

- дослідити принципи геокодування та картографічних вебсервісів;
- спроєктувати архітектуру клієнта з розділенням UI та бізнес-логіки;
- реалізувати безпечну роботу з GPS на ОС Android та асинхронний парсинг геоданих;
- розробити енергоефективний інтерфейс (Dark Mode) з можливістю перемикання маршрутів;
- протестувати систему та зібрати інсталяційний пакет (APK).

Об’єкт дослідження – процес обчислення, оптимізації та інтерактивної візуалізації дорожніх маршрутів у мобільних системах.

Предмет дослідження – інструментальні засоби та методи клієнт-серверної взаємодії при розробці систем планування маршрутів.

Практичне значення. Створений додаток “Zp Navigator Pro” є повноцінним мінімально життєздатним продуктом (MVP), який можна використовувати як самостійний швидкий навігатор або масштабувати для інтеграції у великі логістичні B2B-системи.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА СПЕЦИФІКАЦІЯ ВИМОГ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Дослідження концепцій сучасної мобільної навігації та геокодування

Сьогодні розробка навігаторів відійшла від використання великих офлайн-карт. Сучасний мобільний пристрій працює переважно як клієнт. Смартфон збирає дані з модуля GPS і малює інтерфейс. Всі важкі математичні розрахунки маршрутів виконують зовнішні сервери через REST API.

Під час розробки доводиться стикатися з трьома базовими поняттями:

– *геокодування*: це перетворення текстової адреси у координати на карті (в Україні точність геокодування часто буває низькою, якщо провайдер не оновлює свої бази регулярно);
– *маршрутизація*: пошук найкоротшого шляху на графі доріг (якісний роутинг повинен обов’язково враховувати затори в реальному часі);
– *рендеринг карти*: замість старих растрових картинок зараз використовують векторні тайли (вони важать набагато менше, що дуже важливо при поганому мобільному інтернеті в дорозі).

Технологічний перехід від растрових до векторних тайлів (Vector Tiles) став революцією в мобільній картографії. Растрові тайли – це набір статичних зображень (PNG або JPEG), які сервер генерує для кожного рівня масштабування. Їх головний недолік полягає у фіксованій роздільній здатності: при спробі наблизити карту зображення втрачає чіткість (пікселізується) до моменту завантаження нового тайлу, а зміна стилю карти (наприклад, перемикання на Dark Mode) вимагає завантаження абсолютно нового набору картинок. Натомість векторні тайли передають на клієнт не готові картинки, а математичні дані у форматі JSON або Protocol Buffers (PBF) – координати точок, ліній та полігонів. Мобільний застосунок самостійно

рендерить ці дані за допомогою графічного процесора (GPU). Це дає три критичні переваги для розроблюваної системи “Zp Navigator Pro”:

– *економія трафіку*: векторні дані займають у 3-5 разів менше пропускної здатності мережі, ніж растрові зображення;
– *динамічна стилізація*: додаток може миттєво змінювати кольори вулиць або приховувати зайві об’єкти (наприклад, POI) на рівні коду, без повторних запитів до сервера;
– *плавність* (60 FPS): завдяки безперервному рендерингу на клієнті, масштабування карти (зум) відбувається абсолютно плавно, без візуальних розривів.

Крім того, для швидкого «холодного старту» (Cold Start) навігатора сучасні пристрої використовують технологію A-GPS (Assisted GPS). Вона дозволяє смартфону отримувати попередні дані про орбіти супутників (альманах та ефемериди) через інтернет або мобільні вишки, що скорочує час першого позиціонування з кількох хвилин до 2-3 секунд. Саме тому обов’язковою вимогою до системи є обробка дозволів не лише на точну локацію (ACCESS_FINE_LOCATION), але й на роботу в мережі.

Зберігання офлайн-картографічних даних у пам’яті мобільного пристрою є недоцільним для легковагових навігаційних застосунків, оскільки дорожній граф лише однієї країни може займати понад 8 ГБ оперативної пам’яті. Тому для даного проєкту було обрано онлайн-архітектуру з делегуванням обчислень на хмарний сервер та частковим кешуванням тайлів карти на стороні клієнта.

В контексті даної кваліфікаційної роботи, об’єктом розробки є не просто черговий картографічний сервіс, а спеціалізований застосунок “Zp Navigator Pro”. Його головна відмінність полягає в орієнтації на концепцію «тонкого клієнта» (Thin Client). Замість того, щоб інтегрувати важкі бази даних та зберігати великі обсяги офлайн-карт, які швидко застарівають і перевантажують пам’ять смартфона, розроблений застосунок фокусується на швидкій обробці мережевих запитів. Основна увага в роботі приділяється

оптимізації процесу взаємодії між мобільним користувацьким інтерфейсом та хмарним сервером маршрутизації. Такий підхід дозволяє розробленому застосунку миттєво реагувати на зміну дорожньої обстановки та відображати актуальні затори без навантаження на процесор мобільного пристрою.

1.2 Порівняльний аналіз існуючих аналогів та програмних рішень

Ринок навігаційних додатків жорстко поділений між B2C-монополістами та вузькопрофільними B2B-рішеннями. Для обґрунтування доцільності створення Zp Navigator Pro було проаналізовано три діаметрально різні підходи: Google Maps SDK, комерційну платформу Spoke Dispatch та зв'язку open-source інструментів OpenStreetMap + OSRM.

Google Maps залишається індустріальним стандартом. Проте їхня політика ліцензування API створює високий бар'єр для студентських та інді-проєктів. Для отримання ключа необхідно обов'язково прив'язати банківську картку, а ліміти безкоштовного рівня вичерпуються дуже швидко під час активної стадії дебагу коду.

Платформа Spoke Dispatch блискуче вирішує задачу логістики «останньої милі». Проте архітектура подібних систем є надлишковою для базових навігаційних сценаріїв, оскільки містить вбудований корпоративний функціонал (менеджмент автопарку, диспетчерська вебпанель, сканування штрихкодів). Вони абсолютно непридатні для базового сценарію, коли користувачу треба просто відкрити додаток і за 3 секунди прокласти маршрут додому, таблиця 1.1 наглядно демонструє аналіз та порівняння провайдерів.

Зв'язка OpenStreetMap з власним сервером OSRM (Open Source Routing Machine) – ідеальний варіант з точки зору незалежності від корпорацій. Але розгортання власного серверного модуля маршрутизації є технічно складним завданням, що виходить за межі ресурсів індивідуального розробника. Це потребує значних серверних ресурсів (мінімум 8 ГБ оперативної пам'яті лише

для завантаження дорожнього графа однієї країни), що повністю виходить за рамки бюджету бакалаврського проєкту.

Таблиця 1.1 – Порівняльний аналіз картографічних провайдерів

Провайдер API	**Наявність безкоштовних лімітів**	**Точність даних про трафік**	**Складність інтеграції у Flutter**
Google Maps SDK	Низька (потребує платіжну картку)	Дуже висока	Висока (важкі залежності)
OSRM (Open Source)	Абсолютна (через власний сервер)	Відсутня (лише статика)	Дуже висока (потребує DevOps)
TomTom API	Висока (2500 запитів/день)	Висока	Середня (через HTTP REST)
Spoke Dispatch	Відсутня (Enterprise підписка)	Висока	Висока

Як видно з таблиці 1.1, використання TomTom API є оптимальним рішенням, що забезпечує баланс між функціональністю, доступністю безкоштовного рівня та простотою інтеграції. Сервіс надає 2500 безкоштовних транзакцій щодня без вимоги верифікації фінансових даних, забезпечує точний traffic-flow і з коробки підтримує генерацію масиву точок для альтернативних маршрутів [3].

Проведений порівняльний аналіз існуючих рішень став безпосередньою основою для формування концепції “Zp Navigator Pro”. На відміну від проаналізованих громіздких корпоративних систем, які перевантажені функціями диспетчеризації чи рекламою, розроблена система вирішує конкретну інженерну проблему: забезпечення користувача інструментом для максимально швидкого прокладання маршруту. У роботі реалізовано принцип «мінімального тертя» (low friction) – застосунок дозволяє в два кліки по інтерактивній карті або за допомогою швидкого текстового пошуку отримати готовий маршрут. Вибір TomTom API у цьому контексті є не просто технічною альтернативою Google Maps, а фундаментальною основою розробленої системи, що дозволила повністю ізолювати складну бізнес-логіку обчислення

дорожніх графів на стороні сервера, залишивши мобільному додатку лише функцію швидкого рендерингу векторних тайлів.

1.3 Аналіз апаратних та програмних обмежень мобільних систем

Проєктування мобільного навігаційного програмного забезпечення суттєво відрізняється від розробки класичних десктопних або вебдодатків. Ключовим фактором, який визначає архітектуру системи, є жорсткі обмеження середовища експлуатації. Аналіз цих обмежень є обов’язковим кроком перед формуванням специфікації вимог до “Zp Navigator Pro”.

Енергоспоживання та температурний тротлінг (Thermal Throttling). Робота з картографією є однією з найбільш ресурсомістких задач для смартфона. Одночасна активність екрана (підсвічування), модуля GPS (прийом супутникових сигналів) та графічного процесора (рендеринг карти) призводить до швидкого розряджання акумулятора та нагрівання пристрою. При досягненні критичної температури мобільна ОС примусово знижує тактову частоту процесора (тротлінг), що викликає зависання інтерфейсу. Для уникнення цього явища додаток має бути максимально легковисним, а інтерфейс – оптимізованим (зокрема, через використання Dark Mode для AMOLED-матриць).

Життєвий цикл фонових процесів (Doze Mode). Починаючи з версії Android 6.0, операційна система агресивно керує фоновими процесами для економії заряду. Якщо додаток згорнуто, система може призупинити його доступ до мережі та GPS. Тому архітектура системи повинна передбачати швидке відновлення стану (State Restoration) при поверненні додатка на передній план, а мережеві запити мають виконуватися асинхронно і максимально швидко.

Мережева латентність (Network Latency). Під час переміщення автомобіля якість інтернет-з’єднання постійно змінюється від

високошвидкісного 4G/LTE до нестабільного EDGE або повної втрати сигналу в тунелях. Отже, програмний продукт не може розраховувати на стабільне з'єднання. Це вимагає впровадження жорстких таймаутів (Timeout) для HTTP-запитів до TomTom API. Якщо сервер не відповідає протягом заданого часу, система повинна миттєво перервати запит і повідомити користувача, не допускаючи блокування потоку виконання (UI Thread).

Окремим аспектом відмовостійкості є обробка HTTP-статусу 429 (Too Many Requests), який сервер TomTom повертає у разі вичерпання денного ліміту безкоштовного рівня у 2500 транзакцій. У поточній реалізації основним превентивним механізмом захисту від перевищення квоти є мінімізація кількості мережевих викликів. Зокрема, запит до модуля предиктивного пошуку _fetchSuggestions ініціюється лише після введення щонайменше двох символів, що скорочує кількість звернень до Search API [4]. у порівнянні з реакцією на кожен окремий символ. Запит до Routing API викликається рівно один раз при встановленні двох точок маршруту, а перемикання між основним та альтернативним шляхами відбувається без повторного звернення до сервера – виключно через операції з масивами в оперативній пам'яті пристрою. Таким чином, архітектура системи мінімізує ризик досягнення ліміту через раціональне використання доступних транзакцій.

Розуміння цих обмежень підтверджує правильність обраної концепції «тонкого клієнта»: чим менше математичних обчислень (наприклад, графових алгоритмів Дейкстри чи A* [8]) виконується на самому процесорі смартфона, тим стабільніше і довше працюватиме навігатор у реальних дорожніх умовах.

1.4 Специфікація функціональних та нефункціональних вимог

Чітке формування вимог до програмного забезпечення дозволило уникнути проблеми "feature creep" (безконтрольного розростання функціоналу) на етапі написання коду. Основний фокус проєкту було зміщено на швидкість відклику UI та мінімалізм.

Розглянемо детальніше **функціональні вимоги**, які висуваються до розроблюваної системи.

Збір та обробка GPS-координат пристрою. Під час розробки цього модуля виникла специфічна проблема безпеки. Починаючи з версії Android 11, політики конфіденційності ОС жорстко обмежують доступ до геолокації. Додаток повинен не просто відправити запит на ACCESS_FINE_LOCATION, а й коректно відпрацювати відмову користувача, видавши інформаційний SnackBar, не допускаючи крашу системи.

Інтерактивна взаємодія з картою. Відмальовування базових шарів TileLayer та здатність динамічно будувати полілінії маршруту (PolylineLayer) на основі розпарсеного масиву точок з JSON-відповіді TomTom.

Розрахунок часу (ETA) та дистанції з можливістю миттєвого перемикання між основним маршрутом та альтернативним (fallback-варіантом). Реалізація цього механізму вимагає використання інструментів state-менеджменту для перемальовування віджетів.

Окрім функціональних, розроблена система має задовольняти низку критичних **нефункціональних вимог**.

Продуктивність (Performance). Рендеринг карти має відбуватися без відчутних затримок, з частотою кадрів не менше 60 FPS. Саме тому було обрано кросплатформений фреймворк Flutter, який не використовує повільні системні WebView, а рендерить UI напряму через GPU (графічний процесор).

Енергоефективність. Інтерфейс Zp Navigator Pro спроєктовано виключно в режимі Dark Mode. Практика показує, що для пристроїв з OLED та AMOLED матрицями чорні пікселі фізично вимикаються, суттєво знижує енергоспоживання екрану, що підтверджується технічними специфікаціями виробників AMOLED-дисплеїв під час тривалої навігації з увімкненим екраном.

Портованість (Portability). Кодова база, написана мовою Dart, має без конфліктів компілюватися під ОС Android, а за потреби – під iOS, без глибокого рефакторингу платформно-залежної бізнес-логіки.

Додатково, специфікація вимог до застосунку “Zp Navigator Pro” включає розширену та безпечну взаємодію з користувачем, що безпосередньо реалізовано у коді проєкту:

Зокрема, система включає *модуль інтелектуального пошуку*. Розроблюваний додаток повинен не лише обробляти кліки по карті, але й надавати предиктивний пошук адрес (Autocomplete) з використанням української локалізації, що є критично важливим для зручного використання смартфона водієм.

Наступним важливим аспектом є *стійкість до втрати сигналу та обмежень ОС*. Система має демонструвати коректну поведінку при відсутності дозволів на геолокацію (на рівні політик Android) або тимчасовій втраті інтернет-з’єднання. Програмний продукт спроєктовано таким чином, щоб перехоплювати ці винятки і не допускати аварійного завершення (crash) програми.

Окрім цього, передбачено *адаптивний рендеринг маршрутів в пам’яті*. Застосунок повинен підтримувати механізм динамічного перемикання між основним та альтернативним шляхами без необхідності повторного надсилання HTTP-запиту на сервер, що вимагає ефективного керування станом (State Management) всередині оперативної пам’яті пристрою.

Значна увага приділяється *безпеці даних та комунікацій (Security)*. Оскільки застосунок взаємодіє із зовнішніми хмарними сервісами через API-ключі, а вихідний код мобільних додатків може бути підданий зворотному інжинірингу (Reverse Engineering), система не передає конфіденційні дані користувача у відкритому вигляді. Усі мережеві запити до ендпоінтів TomTom інкапсулюються та виконуються виключно через криптографічний протокол HTTPS (TLS/SSL). Крім того, додаток не зберігає історію переміщень у локальній базі даних (наприклад, SQLite або SharedPreferences) без механізмів шифрування для уникнення витоку геоданих у разі компрометації пристрою.

Останньою ключовою вимогою є *масштабованість розробки (Scalability)*. Кодова база структурована таким чином, щоб додавання нових

модулів (наприклад, голосового помічника для озвучування маневрів або інтеграції з сервісами виклику таксі) не вимагало переписування існуючого ядра. Використання об’єктно-орієнтованого підходу мови Dart та розділення бізнес-логіки від інтерфейсу користувача гарантує відповідність принципам SOLID, що є необхідною умовою для подальшої підтримки та розширення продукту.

1.5 Аналіз проєктних ризиків та шляхи їх мінімізації

Розробка інтерактивної системи планування маршрутів, яка критично залежить від зовнішніх хмарних сервісів та апаратних датчиків, супроводжується низкою технічних та експлуатаційних ризиків. Для забезпечення стабільності програмного продукту на етапі специфікації вимог було ідентифіковано ключові ризики та закладено архітектурні рішення для їх мінімізації.

Першим суттєвим ризиком є *вичерпання квот стороннього API (Rate Limiting).* Оскільки система покладається на безкоштовний рівень (Free Tier) TomTom REST API, існує ймовірність перевищення денного ліміту у 2500 транзакцій. У разі вичерпання ліміту сервер почне повертати помилку HTTP 429 (Too Many Requests), що призведе до повної непрацездатності функції побудови маршрутів. Для мінімізації цього ризику в коді застосунку впроваджено оптимізацію мережевих викликів. Зокрема, модуль предиктивного пошуку адрес (Autocomplete) використовує техніку Debouncing, завдяки якій запит на сервер відправляється не при кожному натисканні клавіші, а лише після того, як користувач припинить введення тексту на кілька сотень мілісекунд. Це дозволяє зменшити кількість запитів до API у 3-4 рази.

Другим викликом виступає *апаратна похибка GPS-позиціонування*. В умовах щільної міської забудови (так звані «кам’яні джунглі») або під час руху

в тунелях сигнал від супутників може відбиватися, що призводить до стрибків координат на відстань від 50 до 200 метрів та може викликати хибне перерахування маршруту. Щоб уникнути цієї проблеми, на рівні конфігурації модуля geolocator задано фільтри просторової точності (distanceFilter). Таким чином, система ігнорує дрібні флуктуації координат, якщо зміщення становить менше 5-10 метрів, що ефективно згладжує візуальне переміщення маркера користувача по карті.

Третім проєктним ризиком є *залежність від сторонніх бібліотек (Vendor Lock-in)*. Використання специфічних пакетів екосистеми Flutter (наприклад, flutter_map або http) створює небезпеку того, що при оновленні мови Dart до мажорної версії ці пакети можуть втратити підтримку розробниками та стати застарілими (deprecated), що унеможливить компіляцію проєкту. З метою мінімізації цього фактору архітектура проєкту побудована за принципом слабкої зв’язності (Loose Coupling). Логіка парсингу та геокодування винесена в окремі незалежні функції, які не прив’язані до конкретного UI-віджета [9]. Завдяки цьому підходу, у разі необхідності зміни картографічного рушія, основна бізнес-логіка додатку залишиться повністю незмінною.

2 ПРОЄКТУВАННЯ АРХІТЕКТУРИ ТА МОДЕЛЮВАННЯ СИСТЕМИ

Якість мобільного програмного забезпечення безпосередньо залежить від закладеного архітектурного фундаменту. Ефективне моделювання процесів взаємодії компонентів дозволяє мінімізувати зв’язність коду (coupling) та забезпечити високу розширюваність системи [11]. У цьому розділі наведено інженерне обґрунтування обраного технологічного стеку, описано архітектурні шари застосунку “Zp Navigator Pro”, виконано об’єктно-орієнтоване моделювання за допомогою мови UML та розроблено концепцію користувацького інтерфейсу.

2.1 Обґрунтування вибору технологічного стеку та інструментальних засобів

Створення сучасного мобільного геосервісу вимагає вирішення дилеми між нативним підходом (розробка окремих кодових баз під Android та iOS) та кросплатформеним. Нативна розробка (Kotlin/Swift) забезпечує максимальну продуктивність, проте подвоює витрати на підтримку та тестування.

Для реалізації інтерактивної системи планування маршрутів було обрано фреймворк Flutter (мова Dart) [1]. Цей вибір обґрунтовано декількома ключовими інженерними факторами.

Першим із них є *продуктивність рендерингу*. На відміну від кросплатформених інструментів типу React Native, які використовують повільний міст (bridge) до нативних віджетів, Flutter рендерить інтерфейс напряму через власний графічний рушій (Skia/Impeller) безпосередньо на GPU [10]. Ця особливість є критично важливою для відмальовування тисяч точок дорожнього графа на карті без втрати плавності та забезпечення стабільних 60 FPS.

Другим вагомим фактором виступає *асинхронне програмування*. Модель виконання мови Dart базується на концепції Event Loop. Використання асинхронних потоків Future та ключових слів async/await дозволяє виконувати важкі мережеві HTTP-запити до картографічного сервера у фоновому режимі, що запобігає блокуванню головного UI-потоку застосунку. Як основне джерело картографічних даних було обрано сервіс TomTom API. Його вибір у порівнянні з Google Maps та OpenStreetMap зумовлений гнучкістю інтеграції. Обчислення геокодування та оптимізації траєкторій відбувається на стороні серверів TomTom за допомогою архітектурного стилю REST. Мобільний додаток обмінюється даними через компактний формат JSON, що суттєво заощаджує мережевий трафік користувача.

В межах розробки “Zp Navigator Pro” вибір фреймворку Flutter обґрунтовується не лише його базовою продуктивністю, але й наявністю потужної екосистеми пакетів, цілеспрямовано інтегрованих у проєкт. Зокрема, для роботи з картографічними даними використано пакет flutter_map у зв’язці з математичною бібліотекою latlong2 [5]. Це дозволило відмовитися від важких нативних залежностей (таких як Google Maps SDK) і реалізувати швидкий рендеринг карти виключно засобами Dart. Для забезпечення безпечного доступу до апаратного модуля геолокації впроваджено пакет geolocator, який бере на себе складну логіку узгодження системних дозволів (permissions) у нових версіях Android [2]. Крім того, асинхронна модель Dart (через класи Future та ключові слова async/await) стала стала критично важливою для реалізації методів взаємодії з API: десеріалізація великих JSON-масивів з сотнями координат від TomTom відбувається без блокування головного потоку (UI-thread), зберігаючи абсолютну плавність та інтерактивність карти під час очікування відповіді від сервера.

Невід’ємною частиною технологічного стеку є файл конфігурації залежностей pubspec.yaml, який визначає усі зовнішні пакети проєкту. У таблиці 2.1 наведено перелік підключених бібліотек із зазначенням їх функціонального призначення.

Таблиця 2.1 – Залежності проєкту “Zp Navigator Pro”

Пакет	Версія	Призначення
flutter_map	7.0.2	Рендеринг інтерактивної карти та шарів поліліній і маркерів безпосередньо через графічний рушій Flutter без використання WebView
latlong2	0.9.1	Математична бібліотека для роботи з географічними координатами у форматі широта/довгота (LatLng), використовується спільно з flutter_map
geolocator	13.0.2	Крос-платформений доступ до апаратного GPS-модуля пристрою з підтримкою системи дозволів Android 11+
http	1.2.2	Легковисний HTTP-клієнт для асинхронних GET-запитів до REST API TomTom

Обрані пакети є активно підтримуваними бібліотеками екосистеми pub.dev з відкритим вихідним кодом. Відсутність важких нативних залежностей (таких як Google Maps SDK) дозволяє зберегти мінімальний розмір фінального APK-пакета та уникнути конфліктів при компіляції під різні версії Android SDK.

2.2 Архітектурні патерни та проєктування шарів системи

У проєкті застосовано шарову архітектуру (Layered Architecture), що концептуально узгоджується з патерном MVVM (Model-View-ViewModel). Однак, оскільки розроблений продукт є мінімально життєздатним прототипом (MVP), було прийнято архітектурне рішення відмовитися від повноцінної реалізації окремого шару ViewModel з використанням складних інструментів керування станом (BLoC, Provider, Riverpod). Реактивність інтерфейсу забезпечується вбудованим механізмом Flutter – функцією setState(), що є обґрунтованим компромісом між архітектурною чистотою та швидкістю розробки прототипу. Головний принцип архітектури – повна ізоляція користувацького інтерфейсу від бізнес-логіки та мережевих запитів.

Систему розділено на три автономні рівні.

Першим із них є *шар представлення (Presentation Layer / View)*, який складається з дерева віджетів (Widget Tree). Він відповідає виключно за відображення карти, маркерів та інформаційних панелей. Цей шар є пасивним, тобто він не містить логіки обчислення маршрутів, а лише реактивно реагує на зміну стану системи.

Другим рівнем виступає *шар логіки та керування станом (Application/State Layer)*, що відповідає за реактивну поведінку додатку. У поточній реалізації (MVP-рішенні) стан інкапсульовано всередині об'єкта _MapScreenState за допомогою базового механізму setState, що забезпечує миттєве перемальовування поліліній на екрані при зміні активного маршруту.

Третім архітектурним компонентом є *шар даних (Data/Service Layer)*. Він містить спеціалізовані сервіси для взаємодії з апаратним GPS-модулем (Geolocator) та мережевий клієнт (http). Цей рівень формує URL-запити, обробляє HTTP-відповіді та виконує десеріалізацію (парсинг) сирих JSON-даних у строго типізовані Dart-моделі координат (LatLng).Такий підхід дозволяє за необхідності повністю замінити провайдера карт (наприклад, перейти з TomTom на Mapbox), змінивши лише код у шарі даних, при цьому інтерфейс додатку залишиться абсолютно незмінним.

Застосовуючи шарову архітектуру до реалій мого застосунку, систему спроєктовано таким чином, щоб мінімізувати залежність візуальної частини від джерела даних. У розробленому MVP-концепті роль проміжного шару логіки та керування станом бере на себе клас MapScreenState, у ньому інкапсульовано ключові змінні стану: масиви координат (routePoints, primaryRoute, alternativeRoute) та розраховані метрики поїздки. Віджети шару представлення (наприклад, кастомна інформаційна панель з деталями маршруту або PolylineLayer) є повністю «реактивними» – вони лише підписуються на ці змінні і автоматично перебудовуються при виклику функції setState(). Такий підхід дозволив зосередити всю логіку мережевих HTTP-запитів та парсингу JSON (Data Layer) в окремих асинхронних методах (_fetchRoute). У перспективі це означає, що за необхідності зміни провайдера

маршрутизації (наприклад, з TomTom на Mapbox) достатньо буде модифікувати лише один метод обробки даних, не зачіпаючи жодного рядка коду, що відповідає за відмальовування інтерфейсу.

2.3 Об’єктно-орієнтоване моделювання системи за допомогою UML-діаграм

Для детального аналізу логіки роботи системи та взаємодії користувача з додатком було проведено об’єктно-орієнтоване моделювання із застосуванням мови UML [7].

Першим етапом стало проєктування діаграми прецедентів (Use Case Diagram) (див. рис. 2.1), яка визначає межі системи та базові сценарії взаємодії користувача (актора) з мобільним застосунком.

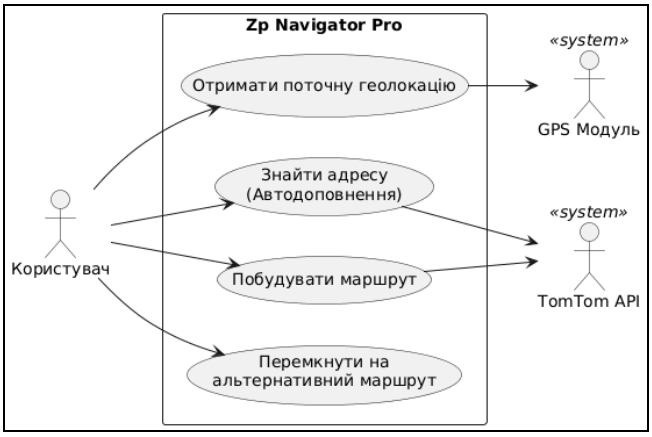


Рисунок 2.1 – Діаграма прецедентів інтерактивної системи

Для опису динаміки системи та взаємодії об’єктів у часі під час розрахунку маршруту побудовано діаграму послідовності (див. рис. 2.2). Вона відображає життєвий цикл запиту: від кліку користувача по екрану до виклику методу оновлення інтерфейсу.

Статичну структуру програмного коду, типи даних та зв’язки між класами віджетів і контролерів описує діаграма класів (див. рис. 2.3). Вона демонструє архітектурні взаємозв’язки всередині кодової бази.

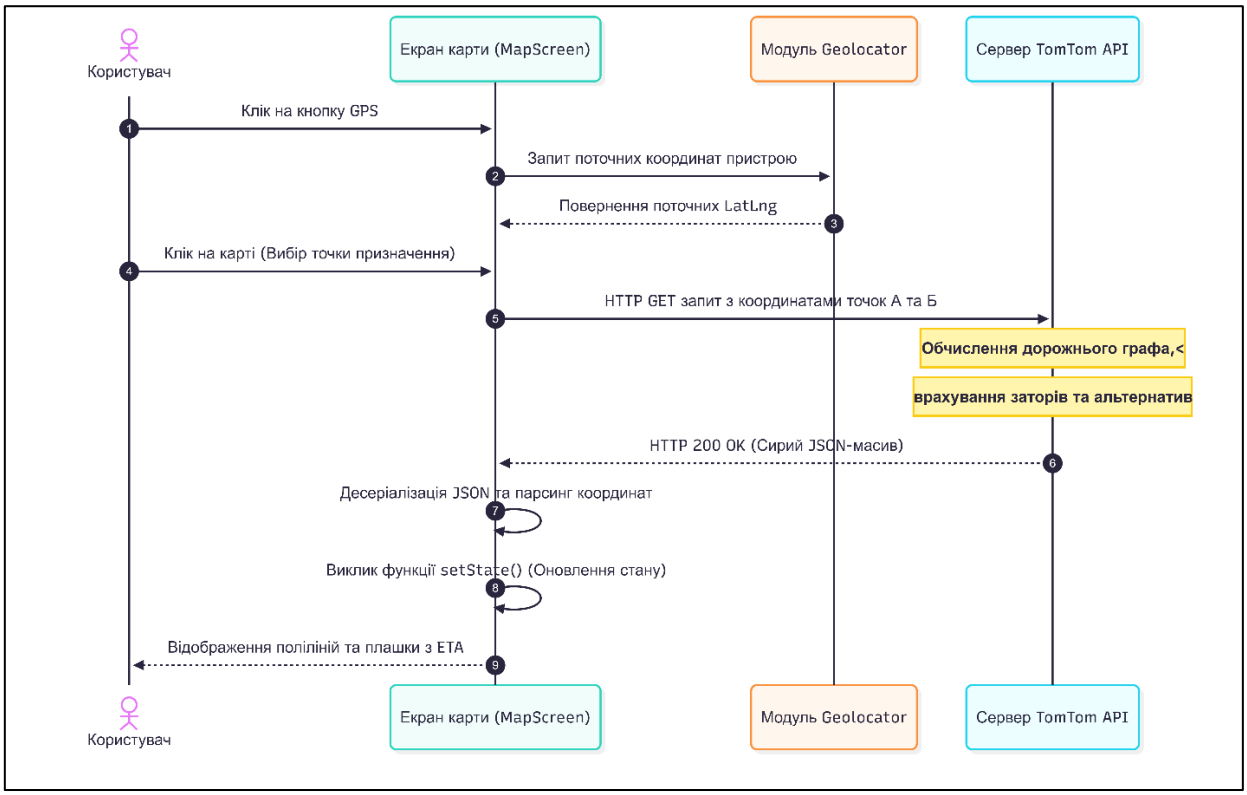


Рисунок 2.2 – Діаграма послідовності процесу планування маршруту

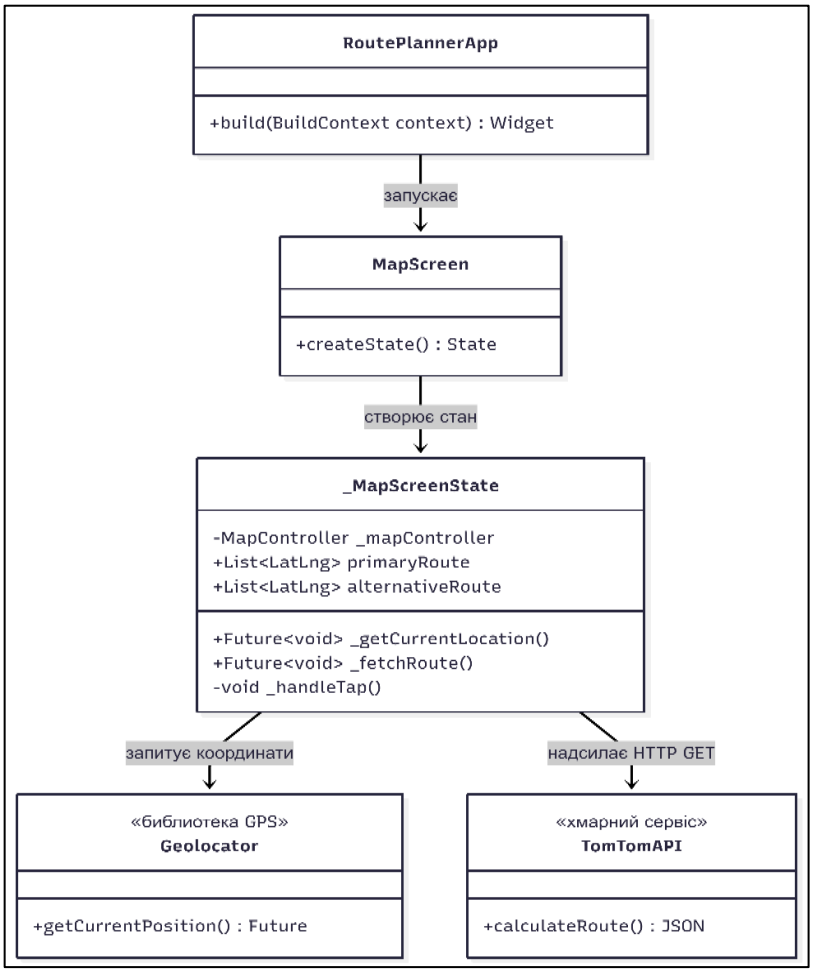


Рисунок 2.3 – Діаграма класів мобільного застосунку

2.4 Проєктування інтерактивного користувацького інтерфейсу

Розробка користувацького інтерфейсу Zp Navigator Pro базувалася на гайдлайнах Material Design від компанії Google [6]. Основний фокус UX-проєктування було спрямовано на ергономічність використання додатку під час руху, коли увага водія обмежена.

Інженерні рішення, впроваджені в UI/UX розроблюваного застосунку, включають декілька ключових аспектів.

Першим важливим рішенням є *безальтернативний Dark Mode*. Використання темної палітри (ThemeData.dark()) та нічної теми картографічних тайлів TomTom суттєво знижує навантаження на зір користувача у темну пору доби. Для мобільних пристроїв з AMOLED-дисплеями це забезпечує фізичне відключення чорних пікселів екрана, ефективно оптимізуючи загальне енергоспоживання системи.

Другим аспектом виступає *мінімалістичний оверлей*. Інформаційна панель з очікуваним часом прибуття (ETA) та дистанцією маршруту спроєктована у вигляді плаваючого контейнера (Positioned), який з’являється на екрані лише після успішної десеріалізації даних маршруту. Такий підхід дозволяє тримати карту максимально відкритою та не перевантажувати інтерфейс зайвими візуальними елементами.

Третім рішенням стало *інтерактивне керування жестами*. Інтегрована карта підтримує стандартні події onTap для швидкого встановлення маркерів початку та кінця шляху безпосередньо по кліку на дорожнє полотно. Це значно пришвидшує процес введення координат та спрощує взаємодію користувача з інтерфейсом.

При проєктуванні користувацького інтерфейсу “Zp Navigator Pro” за основу взято специфіку використання навігаційних систем у реальних умовах – за кермом автомобіля або під час активного руху. Тому було прийнято рішення відмовитися від складних багаторівневих меню на користь концепції «одного екрана» (Single-Page Application). Головний віджет карти займає весь

доступний простір дисплея, а елементи керування та пошуку реалізовані як плаваючі напівпрозорі контейнери, що не перекривають огляд топології вулиць. Нижня інформаційна панель з деталями маршруту (час прибуття, дистанція) з'являється динамічно лише тоді, коли дані успішно отримані від сервера, залишаючи карту максимально відкритою в режимі очікування.

2.5 Архітектурні патерни проєктування користувацького інтерфейсу

Окремим завданням на етапі проєктування системи стала розробка ефективної та масштабованої архітектури користувацького інтерфейсу (UI). Оскільки фреймворк Flutter використовує декларативний підхід до побудови UI, де все є «віджетом» (Widget), неконтрольоване розростання дерева віджетів (Widget Tree) могло б призвести до погіршення читабельності коду та зниження продуктивності під час перемальовування екрана. Для вирішення цієї проблеми у застосунку "Zp Navigator Pro" було застосовано принцип «композиції замість успадкування» (Composition over inheritance) та патерн винесення UI-компонентів у незалежні функції-білдери.

Базова структура головного екрана побудована на основі віджета Stack, який дозволяє накладати шари один на один у координатному просторі осі Z. На нижньому (нульовому) рівні розташовано важкий віджет FlutterMap, який відповідає за рендеринг картографічних тайлів та поліліній. Поверх нього, за допомогою позиціонування (Positioned), розміщуються легкі інтерактивні елементи: плаваюча кнопка виклику геолокації (FloatingActionButton) та інформаційні панелі маршрутів.

Щоб уникнути антипатерну "God Object" (коли метод build головного класу займає сотні рядків коду), логіку побудови повторюваних елементів інтерфейсу було декомпоновано. Яскравим прикладом цього інженерного рішення є створення кастомного компонента генерації карток маршрутів

зображений на рисунку 2.4. Замість дублювання коду для основного та альтернативного шляхів, розроблено універсальний метод-білдер.

```
Widget _buildRouteOption(String label, String time, String dist, bool active, VoidCallback tap) {
  return GestureDetector(
    onTap: tap,
    child: Container(
      padding: const EdgeInsets.all(12),
      decoration: BoxDecoration(
        color: active ? Colors.blueAccent.withOpacity(0.1) : Colors.transparent,
        borderRadius: BorderRadius.circular(12),
        border: Border.all(color: active ? Colors.blueAccent : Colors.transparent, width: 2),
      ), // BoxDecoration
      child: Row(
        mainAxisAlignment: MainAxisAlignment.spaceBetween,
        children: [
          Text(label, style: const TextStyle(fontWeight: FontWeight.bold)),
          Text('$time хв', style: TextStyle(color: active ? Colors.greenAccent : Colors.white, fontSize: 18, fontWeight:
          Text('$dist км', style: const TextStyle(color: Colors.grey)),
```

Рисунок 2.4 – Реалізація методу-білдера для генерації інтерактивної панелі

Такий архітектурний підхід повністю відповідає принципу DRY (Don’t Repeat Yourself). Метод інкапсулює в собі логіку стилізації: він приймає на вхід булевий прапорець active, на основі якого динамічно розраховує кольори рамок, рівень прозорості (opacity) та товщину ліній (width).

Використання віджета GestureDetector забезпечує перехоплення подій дотику (onTap), перетворюючи звичайний інформаційний контейнер на повноцінний елемент керування станом системи. Крім того, виділення статичних та динамічних частин UI у строгу ієрархію гарантує, що під час виклику функції setState() (наприклад, при зміні активного маршруту) фреймворк Flutter здійснює «брудну перевірку» (dirty check) лише змінених вузлів дерева, не витрачаючи ресурси процесора на перерахунок незмінних елементів (таких як поле пошуку або базовий шар карти).

3 ПРАКТИЧНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ПРОГРАМНОГО ПРОДУКТУ

Після завершення етапу проєктування архітектури та моделювання взаємодій було розпочато фазу практичної реалізації мобільного застосунку “Zp Navigator Pro”. Цей етап включав розробку користувацького інтерфейсу, налаштування модулів геолокації, написання мережевого шару для роботи з TomTom REST API та фінальне тестування продукту на віртуальних і фізичних пристроях.

3.1 Розробка користувацького інтерфейсу та робота з картографічними даними

Користувацький інтерфейс побудовано з використанням декларативної парадигми фреймворку Flutter. Головним екраном системи є віджет MapScreen, який успадковується від класу StatefulWidget, який можна побачити на рисунку 3.1, оскільки додаток вимагає постійного оновлення екрана під час взаємодії з картою.

Основою візуалізації є пакет flutter_map. Замість використання стандартного системного компонента WebView, який споживає багато оперативної пам’яті, цей пакет рендерить карту безпосередньо через графічний рушій за допомогою віджета TileLayer. Для забезпечення вимог енергоефективності (Dark Mode) у urlTemplate було передано спеціальний URL-шаблон нічної карти TomTom (basic/night).

Інтерактивна взаємодія реалізована через обробник подій onTap. При кліку користувача по полотну карти координати дотику (об’єкт типу LatLng) автоматично додаються до масиву routePoints. Відображення маркерів реалізовано через MarkerLayer, де початкова точка (індекс 0) підсвічується зеленим кольором, а кінцева точка пункту призначення – червоним.

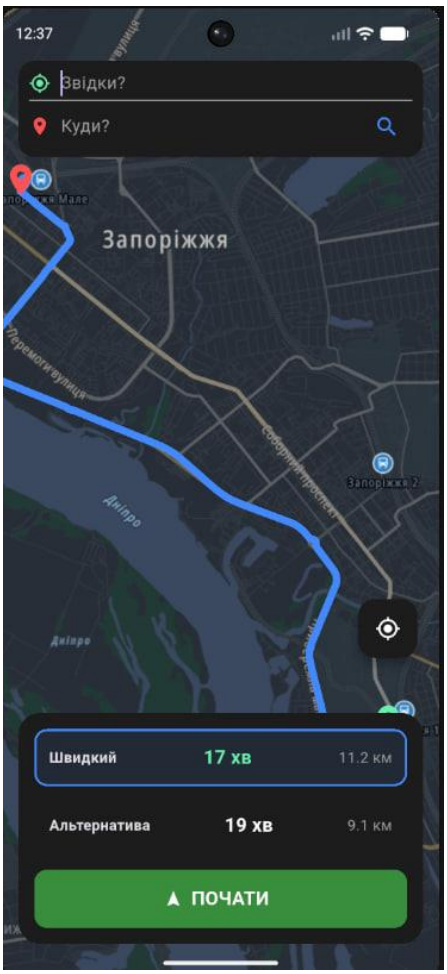


Рисунок 3.1 – Головний екран застосунку “Zp Navigator Pro” (мобільна версія)

Важливим компонентом архітектури користувацького інтерфейсу є об’єкт MapController, який інстанціюється як поле класу _MapScreenState і передається безпосередньо у віджет FlutterMap. На відміну від механізму setState(), який ініціює повне перемальовування дерева віджетів, MapController дозволяє керувати станом карти точково – без зайвого навантаження на рушій рендерингу.

У розробленому застосунку контролер виконує три ключові операції. По-перше, після отримання GPS-координат від модуля geolocator викликається метод _mapController.move(myPos, 15.0), який плавно переміщує камеру до поточного місцезнаходження користувача із рівнем масштабування 15. По-друге, після успішної геокодифікації текстового запиту камера автоматично центрується на знайденій точці викликом

_mapController.move(startPoint, 13.0). По-третє, у момент початку покрокової навігації рівень масштабування збільшується до 17 для забезпечення детального огляду прилеглої дорожньої мережі.

Такий підхід є архітектурно правильним: логіка керування камерою повністю інкапсульована в методах бізнес-логіки (_getCurrentLocation, _buildRouteFromInputs, обробник кнопки «Почати»), а не розпорошена по дереву віджетів, що відповідає принципу єдиної відповідальності (Single Responsibility Principle).

Під час холодного старту застосунку карта відображає географічний центр міста Запоріжжя – координати LatLng(47.8388, 35.1396), жорстко задані у конфігурації MapOptions. Це рішення зумовлене цільовою аудиторією та назвою застосунку “Zp Navigator Pro”: до моменту отримання актуальних GPS-даних від апаратного модуля користувач бачить знайому міську топологію, що усуває дезорієнтацію при першому запуску. Щойно модуль geolocator повертає реальні координати, MapController миттєво переміщує камеру до фактичного місцезнаходження.

Окремої уваги в практичній реалізації заслуговує створення інтерактивного режиму покрокової навігації (Turn-by-Turn Navigation). На відміну від статичного відображення ліній, розроблено логіку ведення користувача за маршрутом. У коді реалізовано метод getTurnIcon, який аналізує текстові інструкції від API (наприклад, «поверніть ліворуч», «кільцевий рух») та динамічно зіставляє їх з відповідними графічними векторами з бібліотеки Icons. Коли користувач натискає кнопку «Почати», система змінює стан (isNavigating = true): камера автоматично фокусується на стартовій локації, а на екрані з’являється спеціальний оверлей, який покроково виводить підказки маневрів. Це перетворює застосунок із простого планувальника у повноцінного дорожнього асистента.

Окремої уваги заслуговує реалізація модуля інтелектуального пошуку адрес, який зображений на рисунку 3.2. Віджет _buildAutocompleteField побудований на основі стандартного компонента Flutter Autocomplete<String>.

Його ключова перевага полягає в асинхронному виклику optionsBuilder – фреймворк самостійно керує частотою звернень до методу _fetchSuggestions, не дублюючи запити при швидкому введенні тексту. Додатковий рівень фільтрації реалізовано безпосередньо в коді: запит до сервера ініціюється лише після введення щонайменше двох символів.

```
Future<List<String>> _fetchSuggestions(String query) async {
  if (query.length < 2) return [];

  final center = _mapController.camera.center;
  final url = 'https://api.tomtom.com/search/2/search/${Uri.encodeComponent(query)}.json?key=$_apiKey&typeahead=true&countrySet=
```

Рисунок 3.2 – Фрагмент вихідного коду: реалізація предиктивного пошуку адрес з фільтрацією запитів

Параметр typeahead=true активує на стороні сервера TomTom режим предиктивного пошуку, що дозволяє отримувати релевантні підказки ще до завершення введення повної адреси. Параметри countrySet=UA та language=uk-UA обмежують область пошуку територією України та забезпечують повернення результатів виключно українською мовою, що є критично важливим для зручності використання застосунку вітчизняними користувачами.

3.2 Інтеграція апаратних датчиків та TomTom REST API

Критично важливим етапом стала реалізація доступу до апаратного GPS-модуля пристрою. У зв’язку з жорсткими політиками безпеки сучасних версій Android (API 30+), у файлі маніфесту AndroidManifest.xml було прописано дозволи ACCESS_FINE_LOCATION та ACCESS_COARSE_LOCATION. Асинхронна функція _getCurrentLocation() перевіряє статус дозволів за допомогою пакета geolocator і, у разі успіху, плавно анімує камеру до поточних координат користувача.

Логіка маршрутизації інкапсульована в асинхронному методі _fetchRoute(), яка зображена на рисунку 3.3. Коли масив routePoints досягає розміру двох елементів (старт і фініш), система ініціює мережевий запит до TomTom API.

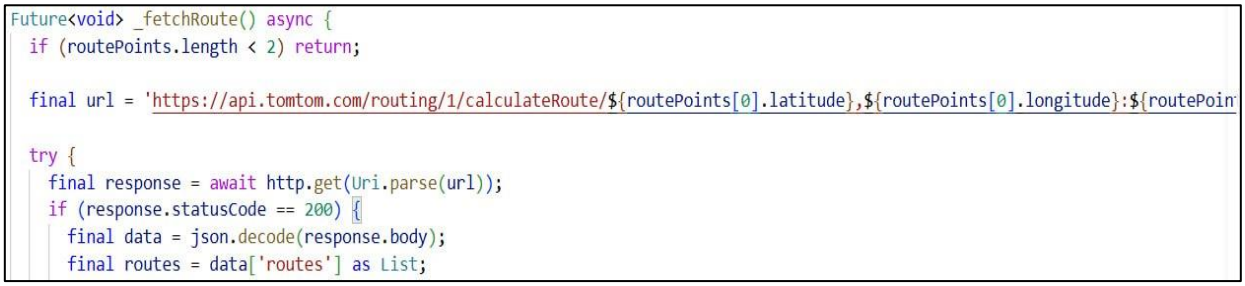


Рисунок 3.3 – Фрагмент вихідного коду: метод асинхронного запиту до TomTom API

Як видно з фрагмента коду на рисунку 3.3, мережевий запит формується за допомогою пакета http. До URL-адреси динамічно підставляються координати точок та персональний API-ключ. Сервер повертає відповідь у форматі JSON. Десеріалізація даних відбувається у блоці try-catch для запобігання аварійному завершенню програми у випадку розриву інтернет-з'єднання. Система парсить отриманий масив routes та витягує з нього вкладені об'єкти legs і points, перетворюючи їх на типізовані списки List<LatLng>.

Важливим етапом практичної реалізації стала розробка ефективного механізму десеріалізації (парсингу) даних. Відповідь від сервера TomTom містить надлишковий обсяг інформації. Щоб не перевантажувати оперативну пам'ять мобільного пристрою, реалізовано власні локальні функції-хелпери (наприклад, extract та extractSteps), які проходять по вкладених масивах JSON-об'єкта (legs та points) і витягують виключно координати latitude та longitude. Ці сирі дані одразу трансформуються у строго типізовані списки List<LatLng>. Завдяки цьому підходу, навіть при побудові довгих маршрутів із тисячами вузлів, процес мапінгу даних відбувається миттєво і не викликає візуальних «фрізів» (зависань) інтерфейсу користувача.

3.3 Керування станом та відображення маршрутів

Оскільки розроблений програмний продукт є концептом (MVP), для керування станом (State Management) було вирішено відмовитися від складних архітектурних патернів типу BLoC або Provider. Реактивність інтерфейсу забезпечується базовим інструментом Flutter – функцією setState().

Після успішного парсингу геоданих система заповнює два масиви: primaryRoute (головний маршрут) та alternativeRoute (найшвидша альтернатива, розрахована алгоритмами TomTom). Виклик setState() повідомляє фреймворк про необхідність перемалювати дерево віджетів. Віджет PolylineLayer миттєво відмальовує ці масиви у вигляді поліліній: синьою для основного шляху та сірою напівпрозорою для альтернативного (рис. 3.4).

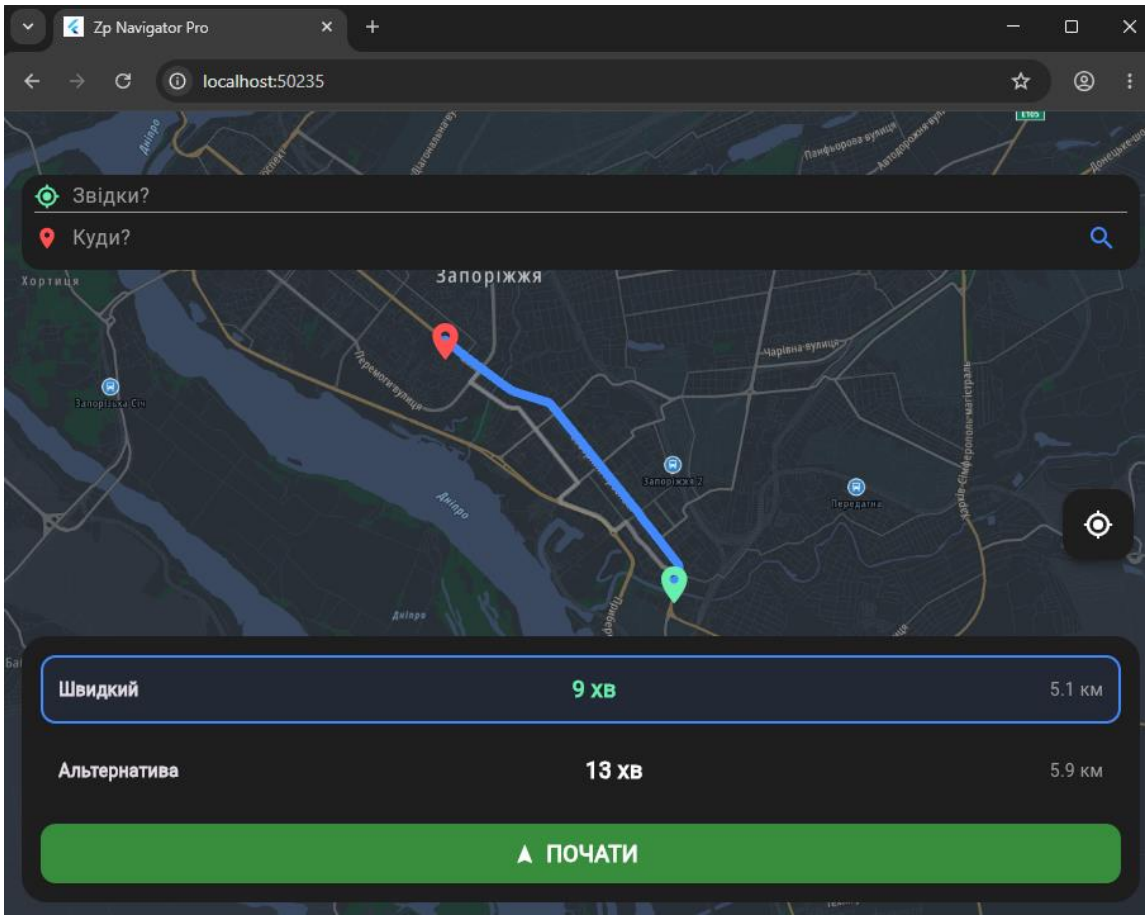


Рисунок 3.4 – Візуалізація основного та альтернативного маршрутів (запуск у режимі Flutter Web для демонстрації)

Одночасно з появою ліній на екрані динамічно генерується нижня інформаційна панель. Користувач має змогу натиснути кнопку перемикання маршрутів (swap_horiz). Ця подія міняє масиви місцями в пам’яті програми, що автоматично призводить до зміни кольорів ліній на карті та перерахунку тексту в полях ETA (час у дорозі) і дистанції.

Ядром бізнес-логіки застосунку “Zp Navigator Pro” є комунікація з картографічним провайдером TomTom REST API. Оскільки розрахунок дорожнього графа є ресурсомісткою операцією, вона повністю делегована хмарному серверу. Мобільний клієнт відповідає виключно за формування правильного HTTP-запиту та десеріалізацію отриманої відповіді. Фрагмент коду, що відповідає за цю логіку, представлено на рисунку 3.5.

```
Future<void> _fetchRoute() async {
  if (routePoints.length < 2) return;

  final url = 'https://api.tomtom.com/routing/1/calculateRoute/${routePoints[0].latitude},${routePoints[0].longitude}:${routePoints

  try {
    final response = await http.get(Uri.parse(url));
    if (response.statusCode == 200) {
      final data = json.decode(response.body);
      final routes = data['routes'] as List;

      setState(() {
        List<LatLng> extract(Map r) => (r['legs'][0]['points'] as List).map((p) => LatLng(p['latitude'], p['longitude'])).toList();

        List<String> extractSteps(Map r) {
          if (r['guidance'] != null && r['guidance']['instructions'] != null) {
            return (r['guidance']['instructions'] as List).map((i) => i['message'].toString()).toList();
          }
          return ["Прямуйте за маршрутом"];
        }

        primaryRoute = extract(routes[0]);
        primaryInstructions = extractSteps(routes[0]);
        routeDistance = (routes[0]['summary']['lengthInMeters'] / 1000).toStringAsFixed(1);
        routeTime = (routes[0]['summary']['travelTimeInSeconds'] / 60).round().toString();

        if (routes.length > 1) {
          alternativeRoute = extract(routes[1]);
          altInstructions = extractSteps(routes[1]);
          altRouteDistance = (routes[1]['summary']['lengthInMeters'] / 1000).toStringAsFixed(1);
          altRouteTime = (routes[1]['summary']['travelTimeInSeconds'] / 60).round().toString();
        } else {
          alternativeRoute = [];
          altInstructions = [];
        }
        isRouteFound = true;
        isAlternativeActive = false;
```

Рисунок 3.5 – Програмна реалізація клієнт-серверної взаємодії з TomTom API

Розроблена функція спрацьовує автоматично, як тільки масив routePoints досягає довжини у два елементи (початкова та кінцева точки маршруту). Формування URI-рядка запиту відбувається динамічно: до базової адреси ендпоінту шляхом інтерполяції рядків вбудовуються поточні координати масиву.

Важливим архітектурним рішенням стала передача додаткових GET-параметрів. Зокрема, прапорець traffic=true змушує серверний алгоритм враховувати актуальні затори у реальному часі при розрахунку полілінії. Параметр maxAlternatives=1 вказує серверу на необхідність повернення не лише головного, але й одного резервного шляху розв’язку. Весь процес мережевої взаємодії обгорнуто в блок try-catch.

Використання ключового слова await перед методом http.get гарантує, що головний потік інтерфейсу не буде заблоковано на час очікування відповіді від сервера. Після отримання статусу відповіді 200 OK, система переходить до локальної функції extractPoints, яка ітерується по JSON-вузлах legs та points, витягуючи сирі координати та наповнюючи масиви primaryRoute та alternativeRoute для подальшого рендерингу.

3.4 Тестування системи та компіляція релізної версії

Процес верифікації програмного продукту проходив у два етапи. На першому етапі використовувався віртуальний емулятор з API 34. Це дозволило перевірити верстку інтерфейсу на різних розмірах екранів та протестувати поведінку додатка при імітації відсутності GPS-сигналу.

На другому етапі застосунок було розгорнуто на фізичному пристрої під управлінням ОС Android через міст налагодження по USB. Тестування на реальному апаратному забезпеченні підтвердило виконання нефункціональних вимог: додаток стабільно видає 60 FPS під час скролінгу карти, апаратний GPS коректно позиціонує користувача, а політика Dark Mode працює без інверсії кольорів.

Під час практичного програмування особливу увагу приділено питанням відмовостійкості розробленої системи. Мобільні пристрої часто працюють в умовах нестабільного покриття мережі. Тому всі мережеві виклики до TomTom API концептуально огорнуті в блоки безпечної обробки винятків try-catch. У разі таймауту або обриву з'єднання, код коректно перехоплює помилку, не допускаючи аварійного закриття застосунку.

Аналогічний підхід застосовано і до апаратних датчиків: якщо користувач відмовляє у доступі до GPS-модуля на рівні налаштувань ОС, програма не блокує свою роботу, а плавно переходить у режим ручного керування, дозволяючи побудувати маршрут шляхом текстового пошуку через поле Autocomplete.

3.5 Методологія та результати тестування системи

Завершальним етапом розробки мобільного застосунку "Zp Navigator Pro" стало проведення функціонального тестування (Functional Testing). Метою цього етапу була перевірка коректності роботи всіх модулів системи, валідація обробки виняткових ситуацій (Edge Cases) та перевірка відмовостійкості програми при зміні станів апаратного забезпечення смартфона.

Оскільки система спроєктована за архітектурою «тонкого клієнта» та критично залежить від зовнішніх джерел даних, тестування проводилося переважно методом «чорного ящика» [12,13]. Основна увага приділялася сценаріям взаємодії з апаратним модулем GPS, поведінці користувацького інтерфейсу та коректності десеріалізації відповідей від TomTom REST API.

Для документування процесу перевірки було розроблено та виконано набір ключових тестових сценаріїв (Test Cases), які детально описані нижче.

Перший сценарій тестування передбачав ініціалізацію системи без дозволів на геолокацію. Метою цього етапу була перевірка здатності застосунку коректно обробляти жорсткі політики безпеки сучасних

операційних систем (Android 11 та вище), коли користувач свідомо забороняє доступ до апаратного GPS-модуля. Під час тестування в налаштуваннях операційної системи було повністю вимкнено службу передачі геоданих, після чого відбувся холодний запуск (Cold Start) застосунку. Відповідно до очікувань, відсутність доступу до ключового датчика не призвела до фатальної помилки програми. Виняток безпеки був успішно перехоплений пакетом geolocator, застосунок не завершив роботу аварійно (Crash), а замість цього вивів інформаційне повідомлення (SnackBar) з проханням надати дозвіл. При цьому головний екран з картою завантажився коректно, залишаючи користувачу можливість скористатися ручним текстовим пошуком маршруту.

Другий сценарій був спрямований на дослідження поведінки системи при втраті інтернет-з’єднання. Головною метою виступала оцінка відмовостійкості системи (Fault Tolerance) в умовах нестабільного мобільного покриття, що є типовою ситуацією під час руху автомобіля за містом. Для відтворення ситуації користувач обирав початкову та кінцеву точки маршруту, а в момент ініціалізації функції _fetchRoute (до отримання відповіді від сервера) примусово вимикалися модулі Wi-Fi та мобільної передачі даних (LTE). Запит до REST API передбачувано завершувався помилкою таймауту. Однак, завдяки архітектурному рішенню обгорнути всі мережеві виклики у блоки try-catch, виняток типу SocketException успішно перехоплювався на рівні бізнес-логіки. Інтерфейс користувача не «зависав» в очікуванні відповіді, а миттєво повертав спливаюче повідомлення про відсутність з’єднання з сервером.

Третій сценарій охоплював взаємодію з альтернативними маршрутами в оперативній пам’яті. Це дозволило перевірити ефективність управління станом (State Management) та швидкодію перемальовування користувацького інтерфейсу без здійснення зайвих HTTP-запитів. Після успішної побудови маршруту, який складався з основного та альтернативного шляхів, користувач натискав на неактивну (сіру) полілінію або відповідну кнопку на інформаційній панелі. Логіка програми передбачала перехресне присвоєння

масивів координат primaryRoute та alternativeRoute. Тестування показало, що виклик методу setState() відпрацьовує коректно: колір ліній на карті миттєво змінюється, а час прибуття (ETA) та дистанція на панелі оновлюються синхронно з обраним шляхом. Жодних візуальних затримок або дублювання мережевих запитів до TomTom API не зафіксовано.

Четвертий сценарій полягав у перевірці предиктивного пошуку адрес із застосуванням локалізації. Основною метою була валідація роботи модуля Autocomplete та коректності обробки текстових запитів українською мовою. Для цього у рядок пошуку вводився частковий запит (наприклад, «Запоріжжя, Собор...»). Відповідно до заданого у коді параметра language=uk-UA, сервер повертав масив пропозицій виключно українською мовою. Випадаючий список динамічно оновлювався під час введення кожного нового символу, демонструючи високу швидкість обробки подій onChange текстового контролера.

Загальний висновок за результатами проведених тестів підтверджує високий рівень стабільності та надійності розробленого програмного забезпечення. Особливо ефективно відпрацювали механізми захисту на рівні мережевої взаємодії та роботи з апаратними датчиками. Перехоплення критичних винятків реалізовано згідно з сучасними стандартами розробки мобільних інтерфейсів. Це гарантує, що у випадку непередбачуваних ситуацій застосунок забезпечить користувачу інформативний зворотний зв’язок і збереже повну працездатність доступних офлайн-модулів.

3.6 Компіляція та підготовка релізної версії застосунку

Завершальним технічним етапом розробки стала підготовка інсталяційного пакета для розгортання на кінцевих пристроях користувачів. Інструментарій Flutter CLI розрізняє два принципово різних режими компіляції: debug та release.

У режимі debug застосунок компілюється з використанням JIT-компілятора (Just-in-Time), який перетворює Dart-байткод у машинний код безпосередньо під час виконання програми. Цей режим містить вбудовані інструменти профілювання та налагодження, проте суттєво збільшує розмір пакета і знижує продуктивність – він призначений виключно для стадії розробки.

При компіляції у release-режимі активується AOT-компілятор (Ahead-of-Time), який перетворює увесь Dart-код у нативний машинний код ARM заздалегідь – до першого запуску програми на пристрої. Це забезпечує значно швидший старт застосунку та вищу продуктивність під час роботи. Додатково під час збірки виконуються три оптимізації: tree shaking – видалення невикористаного коду з підключених бібліотек; мінімізація ресурсів – стиснення графічних та шрифтових активів; обфускація Java-оболонки за правилами ProGuard, що ускладнює зворотний інжиніринг пакета.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи успішно вирішено інженерну задачу – спроєктовано та розроблено кросплатформений мобільний застосунок “Zp Navigator Pro”. Створений програмний продукт є повноцінним мінімально життєздатним продуктом (MVP) для інтерактивного планування маршрутів, який відповідає всім заявленим вимогам специфікації.

Під час дослідження предметної області та аналізу наявних рішень було виявлено головну проблему сучасних навігаторів – надмірну залежність від важких закритих SDK. Для уникнення цієї проблеми розроблено легковисну архітектуру, що базується на використанні хмарного TomTom REST API. Такий підхід дозволив повністю делегувати складні обчислення дорожніх графів на сторону сервера, звільнивши апаратні ресурси мобільного пристрою, та забезпечив доступ до високоточних даних про трафік без комерційних обмежень.

Вибір фреймворку Flutter та мови програмування Dart виявився технічно виправданим. Спроєктована шарова архітектура додатку чітко розділяє користувацький інтерфейс та бізнес-логіку. Відмова від повільних системних WebView на користь прямого рендерингу тайлів карти за допомогою пакета flutter_map вирішила проблему просідання фреймрейту. Навіть при динамічному відмальовуванні складних поліліній інтерфейс працює стабільно і плавно.

Практична реалізація проєкту супроводжувалася вирішенням кількох критичних технічних викликів. По-перше, інтеграція апаратного модуля GPS вимагала суворого дотримання нових політик безпеки ОС Android (API 30+), що зумовило розробку надійної системи обробки дозволів. По-друге, асинхронний парсинг масивних JSON-відповідей від сервера був оптимізований таким чином, щоб десеріалізація сотень координат не блокувала головний UI-потік програми.

Окремий акцент у розробці було зроблено на ергономіці та енергоефективності. Застосунок функціонує виключно в режимі Dark Mode. Практика показує, що для пристроїв з AMOLED-матрицями це не лише зменшує навантаження на зір водія, але й суттєво економить заряд акумулятора під час тривалих поїздок.

Проведене тестування на віртуальних емуляторах та фізичному пристрої підтвердило надійність системи. За результатами функціонального тестування усі чотири сценарії отримали статус «Пройдено». Рендеринг карти стабільно забезпечує частоту кадрів 60 FPS під час скролінгу та масштабування. Безкоштовний рівень TomTom API надає 2500 транзакцій на добу, що є достатнім для повноцінного використання застосунку в умовах реального трафіку. Фінальний інсталяційний пакет app-release.apk, зібраний у режимі AOT-компіляції, готовий до розгортання на пристроях з Android 5.0 і вище.

Інтерактивний механізм перемикання станів (State Management) коректно оновлює метрики часу та дистанції при виборі користувачем альтернативних шляхів. Фінальний програмний продукт успішно скомпільовано за допомогою інструментарію Flutter CLI у готовий до розгортання інсталяційний пакет (APK).

Закладена в “Zp Navigator Pro” архітектура є повністю масштабованою. За необхідності поточну кодову базу можна використовувати як незалежний клієнтський навігатор, або безболісно розширити до масштабної логістичної платформи B2B-сегменту.

ПЕРЕЛІК ПОСИЛАНЬ

1. Flutter documentation. *Official website*. URL: https://docs.flutter.dev/ (дата звернення: 04.02.2026).

2. Geolocator plugin for Flutter. *Dart packages*. URL: https://pub.dev/packages/geolocator (дата звернення: 04.02.2026).

3. TomTom Routing API Documentation. *TomTom Developer Portal*. URL: https://developer.tomtom.com/routing-api/documentation/ (дата звернення: 13.02.2026).

4. TomTom Search API Documentation. *TomTom Developer Portal*. URL: https://developer.tomtom.com/search-api/documentation/ (дата звернення: 16.02.2026).

5. Flutter_map package documentation. *Dart packages*. URL: https://pub.dev/packages/flutter_map (дата звернення: 15.05.2026).

6. Material Design 3 Guidelines. *Official system specifications*. URL: https://m3.material.io/ (дата звернення: 02.03.2026).

7. Лавріщева К. М. Програмна інженерія : підручник. Київ : Академперіодика, 2016. 314 с.

8. Ткачов В. М., Коваленко О. В. Алгоритми та структури даних : навчальний посібник. Харків : ХНУРЕ, 2021. 156 с.

9. Блінов І. О., Романюк О. В. Програмування мобільних пристроїв та систем : навчальний посібник. Вінниця : ВНТУ, 2022. 120 с.

10. Windmill E. Flutter in Action : Application development with Clean Architecture. Shelter Island : Manning Publications, 2020. 325 p.

11. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Reading : Addison-Wesley, 1994. 395 p.

12. Крепич С. Я. Тестування програмного забезпечення : навчальний посібник. Тернопіль : ТНЕУ, 2019. 144 с.

13. Баранюк О. В. Тестування програмного забезпечення та верифікація мобільних інтерфейсів : конспект лекцій. Запоріжжя : ЗНУ, 2023. 88 с.

ДОДАТОК А

Лістинг вихідного коду мобільного застосунку

```
import 'package:flutter/material.dart';
import 'package:flutter_map/flutter_map.dart';
import 'package:latlong2/latlong.dart';
import 'package:http/http.dart' as http;
import 'package:geolocator/geolocator.dart';
import 'dart:convert';

void main() {
  runApp(const RoutePlannerApp());
}

class RoutePlannerApp extends StatelessWidget {
  const RoutePlannerApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Zp Navigator Pro',
      theme: ThemeData.dark().copyWith(
        scaffoldBackgroundColor: const Color(0xFF121212),
      ),
      home: const MapScreen(),
      debugShowCheckedModeBanner: false,
    );
  }
}

class MapScreen extends StatefulWidget {
  const MapScreen({super.key});

  @override
  State<MapScreen> createState() => _MapScreenState();
}

class _MapScreenState extends State<MapScreen> {
  final MapController _mapController = MapController();

  String _startQuery = '';
  String _endQuery = '';

  List<LatLng> routePoints = [];
  List<LatLng> primaryRoute = [];
  List<LatLng> alternativeRoute = [];

  List<String> primaryInstructions = [];
  List<String> altInstructions = [];

  String routeTime = '';
```

```
        String routeDistance = '';
        String altRouteTime = '';
        String altRouteDistance = '';

        bool isRouteFound = false;
        bool isAlternativeActive = false;

        bool isNavigating = false;
        int currentStep = 0;

        final String _apiKey = 'OBXtWeCr8vwn5NPSS4Djo6MgNgdEXc1J';

        Future<void> _getCurrentLocation() async {
          bool serviceEnabled = await Geolocator.isLocationServiceEnabled();
          if (!serviceEnabled) return;

          LocationPermission permission = await Geolocator.checkPermission();
          if (permission == LocationPermission.denied) {
            permission = await Geolocator.requestPermission();
            if (permission == LocationPermission.denied) return;
          }

          Position position = await Geolocator.getCurrentPosition();
          LatLng myPos = LatLng(position.latitude, position.longitude);
          _mapController.move(myPos, 15.0);

          setState(() {
            _startQuery = "Моє місцезнаходження";
            if (routePoints.isEmpty) {
              routePoints.add(myPos);
            } else {
              routePoints[0] = myPos;
            }
          });
        }

        Future<List<String>> _fetchSuggestions(String query) async {
          if (query.length < 2) return [];

          final center = _mapController.camera.center;
          final url =
'https://api.tomtom.com/search/2/search/${Uri.encodeComponent(query)}.json?ke
y=$_apiKey&typeahead=true&countrySet=UA&language=uk-
UA&lat=${center.latitude}&lon=${center.longitude}&limit=10';

          try {
            final response = await http.get(Uri.parse(url));
            if (response.statusCode == 200) {
              final data = json.decode(response.body);
              final results = data['results'] as List;
              return results.map((e) =>
e['address']['freeformAddress'].toString()).toList();
            }
          } catch (e) {
            debugPrint("Помилка підказок: $e");
          }
```

```
          return [];
        }

        Future<LatLng?> _searchCoordinates(String query) async {
          if (query.isEmpty) return null;
          if (query == "Моє місцезнаходження" && routePoints.isNotEmpty)
return routePoints[0];

          final center = _mapController.camera.center;
          final url =
'https://api.tomtom.com/search/2/search/${Uri.encodeComponent(query)}.json?ke
y=$_apiKey&typeahead=true&countrySet=UA&language=uk-
UA&lat=${center.latitude}&lon=${center.longitude}&limit=1';

          try {
            final response = await http.get(Uri.parse(url));
            if (response.statusCode == 200) {
              final data = json.decode(response.body);
              final results = data['results'] as List;
              if (results.isNotEmpty) {
                final pos = results[0]['position'];
                return LatLng(pos['lat'], pos['lon']);
              }
            }
          } catch (e) {
            debugPrint("Помилка координат: $e");
          }
          return null;
        }

        Future<void> _buildRouteFromInputs() async {
          if (_startQuery.isEmpty || _endQuery.isEmpty) return;

          final startPoint = await _searchCoordinates(_startQuery);
          final endPoint = await _searchCoordinates(_endQuery);

          if (startPoint != null && endPoint != null) {
            setState(() {
              routePoints = [startPoint, endPoint];
              _mapController.move(startPoint, 13.0);
              isNavigating = false;
              currentStep = 0;
            });
            _fetchRoute();
            FocusManager.instance.primaryFocus?.unfocus();
          }
        }

        Future<void> _fetchRoute() async {
          if (routePoints.length < 2) return;

          final url =
'https://api.tomtom.com/routing/1/calculateRoute/${routePoints[0].latitude},$
{routePoints[0].longitude}:${routePoints[1].latitude},${routePoints[1].longit
ude}/json?key=$_apiKey&traffic=true&maxAlternatives=1&instructionsType=text&l
anguage=uk-UA';
```

```
          try {
            final response = await http.get(Uri.parse(url));
            if (response.statusCode == 200) {
              final data = json.decode(response.body);
              final routes = data['routes'] as List;

              setState(() {
                List<LatLng> extract(Map r) => (r['legs'][0]['points'] as
List).map((p) => LatLng(p['latitude'], p['longitude'])).toList();

                List<String> extractSteps(Map r) {
                  if (r['guidance'] != null && r['guidance']['instructions']
!= null) {
                    return (r['guidance']['instructions'] as List).map((i) =>
i['message'].toString()).toList();
                  }
                  return ["Прямуйте за маршрутом"];
                }

                primaryRoute = extract(routes[0]);
                primaryInstructions = extractSteps(routes[0]);
                routeDistance = (routes[0]['summary']['lengthInMeters'] /
1000).toStringAsFixed(1);
                routeTime = (routes[0]['summary']['travelTimeInSeconds'] /
60).round().toString();

                if (routes.length > 1) {
                  alternativeRoute = extract(routes[1]);
                  altInstructions = extractSteps(routes[1]);
                  altRouteDistance = (routes[1]['summary']['lengthInMeters'] /
1000).toStringAsFixed(1);
                  altRouteTime = (routes[1]['summary']['travelTimeInSeconds']
/ 60).round().toString();
                } else {
                  alternativeRoute = [];
                  altInstructions = [];
                }
                isRouteFound = true;
                isAlternativeActive = false;
              });
            }
          } catch (e) { debugPrint("$e"); }
        }

        IconData _getTurnIcon(String instruction) {
          final lower = instruction.toLowerCase();
          if (lower.contains('ліворуч')) return Icons.turn_left;
          if (lower.contains('праворуч')) return Icons.turn_right;
          if (lower.contains('кільц')) return Icons.roundabout_right;
          if (lower.contains('прибули') || lower.contains('фініш')) return
Icons.flag;
          return Icons.straight;
        }

        @override
        Widget build(BuildContext context) {
```

```
          final activeInstructions = isAlternativeActive ? altInstructions :
primaryInstructions;

          return Scaffold(
            body: Stack(
              children: [
                FlutterMap(
                  mapController: _mapController,
                  options: MapOptions(
                    initialCenter: const LatLng(47.8388, 35.1396),
                    initialZoom: 13.0,
                    onTap: (_, p) {
                      if(isNavigating) return;
                      setState(() {
                        if (routePoints.length >= 2) routePoints.clear();
                        routePoints.add(p);
                        if (routePoints.length == 2) _fetchRoute();
                      });
                    },
                  ),
                  children: [
                    TileLayer(
                      urlTemplate:
'https://api.tomtom.com/map/1/tile/basic/night/{z}/{x}/{y}.png?key=$_apiKey',
                      userAgentPackageName: 'com.example.route_planner',
                    ),
                    PolylineLayer(
                      polylines: [
                        if (alternativeRoute.isNotEmpty && !isNavigating)
                          Polyline(
                            points: alternativeRoute,
                            strokeWidth: isAlternativeActive ? 7.0 : 4.0,
                            color: isAlternativeActive ? Colors.blueAccent :
Colors.grey.withOpacity(0.5),
                          ),
                        if (primaryRoute.isNotEmpty)
                          Polyline(
                            points: primaryRoute,
                            strokeWidth: isAlternativeActive && !isNavigating
? 4.0 : 7.0,
                            color: isAlternativeActive && !isNavigating ?
Colors.grey.withOpacity(0.5) : Colors.blueAccent,
                          ),
                      ],
                    ),
                    MarkerLayer(
                      markers: routePoints.asMap().entries.map((e) => Marker(
                        point: e.value,
                        width: 40, height: 40,
                        child: Icon(Icons.location_on, color: e.key == 0 ?
Colors.greenAccent : Colors.redAccent, size: 35),
                      )).toList(),
                    ),
                  ],
                ),
```

```
                if (!isNavigating)
                  Positioned(
                    top: 50, left: 15, right: 15,
                    child: Container(
                      padding: const EdgeInsets.symmetric(horizontal: 10,
vertical: 5),
                      decoration: BoxDecoration(color: const
Color(0xFF1E1E1E), borderRadius: BorderRadius.circular(15)),
                      child: Column(
                        children: [
                          Row(
                            children: [
                              const Icon(Icons.my_location, size: 20, color:
Colors.greenAccent),
                              const SizedBox(width: 10),
                              Expanded(child:
_buildAutocompleteField('Звідки?', (val) => _startQuery = val, _startQuery)),
                            ],
                          ),
                          const Divider(color: Colors.grey, height: 1),
                          Row(
                            children: [
                              const Icon(Icons.location_on, size: 20, color:
Colors.redAccent),
                              const SizedBox(width: 10),
                              Expanded(child: _buildAutocompleteField('Куди?',
(val) => _endQuery = val, _endQuery)),
                              IconButton(icon: const Icon(Icons.search, color:
Colors.blueAccent), onPressed: _buildRouteFromInputs),
                            ],
                          ),
                        ],
                      ),
                    ),
                  )
                else if (activeInstructions.isNotEmpty)
                  Positioned(
                    top: 50, left: 15, right: 15,
                    child: GestureDetector(
                      onTap: () {
                        setState(() {
                          if (currentStep < activeInstructions.length - 1) {
                            currentStep++;
                          } else {
                            isNavigating = false;
                          }
                        });
                      },
                      child: Container(
                        padding: const EdgeInsets.all(20),
                        decoration: BoxDecoration(
                          color: Colors.green.shade800,
                          borderRadius: BorderRadius.circular(15),
                          boxShadow: const [BoxShadow(color: Colors.black54,
blurRadius: 10, offset: Offset(0, 5))],
                        ),
```

```
                        child: Row(
                          children: [

Icon(_getTurnIcon(activeInstructions[currentStep]), size: 40, color:
Colors.white),
                            const SizedBox(width: 15),
                            Expanded(
                              child: Text(
                                activeInstructions[currentStep],
                                style: const TextStyle(color: Colors.white,
fontSize: 18, fontWeight: FontWeight.bold),
                              ),
                            ),
                          ],
                        ),
                      ),
                    ),
                  ),

                Positioned(
                  right: 20, bottom: isRouteFound && !isNavigating ? 300 : 30,
                  child: FloatingActionButton(onPressed: _getCurrentLocation,
backgroundColor: const Color(0xFF1E1E1E), child: const Icon(Icons.gps_fixed,
color: Colors.white)),
                ),

                if (isRouteFound && !isNavigating)
                  Positioned(
                    bottom: 30, left: 15, right: 15,
                    child: Container(
                      padding: const EdgeInsets.all(15),
                      decoration: BoxDecoration(color: const
Color(0xFF1E1E1E), borderRadius: BorderRadius.circular(20)),
                      child: Column(
                        children: [
                          _buildRouteOption('Швидкий', routeTime,
routeDistance, !isAlternativeActive, () => setState(() => isAlternativeActive
= false)),
                          if (alternativeRoute.isNotEmpty) ...[
                            const SizedBox(height: 10),
                            _buildRouteOption('Альтернатива', altRouteTime,
altRouteDistance, isAlternativeActive, () => setState(() =>
isAlternativeActive = true)),
                          ],
                          const SizedBox(height: 15),
                          ElevatedButton.icon(
                            onPressed: () {
                              setState(() {
                                isNavigating = true;
                                currentStep = 0;
                                _mapController.move(routePoints[0], 17.0);
                              });
                            },
                            icon: const Icon(Icons.navigation, color:
Colors.white),
                            label: const Text('ПОЧАТИ', style:
```

```
TextStyle(color: Colors.white, fontSize: 18, fontWeight: FontWeight.bold)),
                            style: ElevatedButton.styleFrom(
                              backgroundColor: Colors.green.shade700,
                              minimumSize: const Size(double.infinity, 55),
                              shape: RoundedRectangleBorder(borderRadius:
BorderRadius.circular(12)),
                            ),
                          )
                        ],
                      ),
                    ),
                  ),

                if (isNavigating)
                  Positioned(
                    bottom: 30, left: 20,
                    child: FloatingActionButton(
                      onPressed: () => setState(() => isNavigating = false),
                      backgroundColor: Colors.redAccent,
                      child: const Icon(Icons.close, color: Colors.white),
                    ),
                  ),
              ],
            ),
          );
        }

        Widget _buildAutocompleteField(String hint, Function(String)
onChanged, String initialValue) {
          return Autocomplete<String>(
            optionsBuilder: (TextEditingValue textEditingValue) async {
              onChanged(textEditingValue.text);
              if (textEditingValue.text.length < 2) return const
Iterable<String>.empty();
              return await _fetchSuggestions(textEditingValue.text);
            },
            onSelected: (String selection) {
              onChanged(selection);
              FocusManager.instance.primaryFocus?.unfocus();
            },
            fieldViewBuilder: (context, controller, focusNode,
onEditingComplete) {
              if (initialValue == "Моє місцезнаходження" &&
controller.text.isEmpty) {
                 controller.text = initialValue;
              }
              return TextField(
                controller: controller,
                focusNode: focusNode,
                style: const TextStyle(color: Colors.white),
                decoration: InputDecoration(hintText: hint, hintStyle: const
TextStyle(color: Colors.grey), border: InputBorder.none, isDense: true),
                onChanged: onChanged,
                onSubmitted: (_) {
                  onEditingComplete();
                  _buildRouteFromInputs();
```

```
              },
            );
          },
          optionsViewBuilder: (context, onSelected, options) {
            return Align(
              alignment: Alignment.topLeft,
              child: Material(
                color: Colors.transparent,
                child: Container(
                  width: MediaQuery.of(context).size.width - 80,
                  margin: const EdgeInsets.only(top: 5),
                  decoration: BoxDecoration(color: const Color(0xFF2C2C2C),
borderRadius: BorderRadius.circular(10)),
                  child: ListView.builder(
                    padding: EdgeInsets.zero,
                    shrinkWrap: true,
                    itemCount: options.length,
                    itemBuilder: (context, index) {
                      final String option = options.elementAt(index);
                      return ListTile(
                        leading: const Icon(Icons.location_city, color:
Colors.grey),
                        title: Text(option, style: const TextStyle(color:
Colors.white, fontSize: 14)),
                        onTap: () => onSelected(option),
                      );
                    },
                  ),
                ),
              ),
            );
          },
        );
      }

      Widget _buildRouteOption(String label, String time, String dist, bool
active, VoidCallback tap) {
        return GestureDetector(
          onTap: tap,
          child: Container(
            padding: const EdgeInsets.all(12),
            decoration: BoxDecoration(
              color: active ? Colors.blueAccent.withOpacity(0.1) :
Colors.transparent,
              borderRadius: BorderRadius.circular(12),
              border: Border.all(color: active ? Colors.blueAccent :
Colors.transparent, width: 2),
            ),
            child: Row(
              mainAxisAlignment: MainAxisAlignment.spaceBetween,
              children: [
                Text(label, style: const TextStyle(fontWeight:
FontWeight.bold)),
                Text('$time хв', style: TextStyle(color: active ?
Colors.greenAccent : Colors.white, fontSize: 18, fontWeight:
FontWeight.bold)),
```

```
                  Text('$dist км', style: const TextStyle(color:
Colors.grey)),
                ],
              ),
            ),
          );
        }
      }
```