НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ
Завідувач кафедри
Комп'ютерних наук
(назва кафедри)
_____ Голуб Б. Л.
(підпис) (ПІБ)
" 2 " червня 2025 р.

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА
на тему

" Програмне забезпечення мобільної системи служби таксі"

Спеціальність 121 – «Інженерія програмного забезпечення»

Гарант освітньої програми

к.н.т., доцент _____ Вайганг Г.О.
(наукова ступінь та вчене звання) (підпис) (ПІБ)

Керівник Бакалаврської кваліфікаційної роботи

ст. викладач _____ Міловідов Ю. О.
(наукова ступінь та вчене звання) (підпис) (ПІБ)

Виконав _____ Бондар Микола Юрійович
(підпис) (ПІБ студента)

КИЇВ – 2025

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

ЗАТВЕРДЖУЮ
Завідувач кафедри
Комп'ютерних наук
(назва кафедри)
_____ Голуб Б. Л.
(підпис) (ПІБ)
"16" грудня 2024 р.

З А В Д А Н Н Я
на виконання бакалаврської кваліфікаційної роботи студенту
Бондару Миколі Юрійовичу
(прізвище, ім'я, по батькові)
Спеціальність 121 – «Інженерія програмного забезпечення»
Тема бакалаврської кваліфікаційної оботи
Програмне забезпечення мобільної системи служби таксі
Затверджена наказом ректора НУБіП України "16" грудня 2024 р.
від № 2249 "С"
Термін подання завершеної роботи на кафедру 2025.06.02
(рік, місяць, число)
Вихідні дані до бакалаврської кваліфікаційної роботи
Опис предмету дослідження, опис програмного забезпечення
Перелік питань, які потрібно розробити:
Системний аналіз предметної області
Аналіз предметної області
Розробка програмного забезпечення
Тестування програмного забезпечення

Дата видачі завдання "16" грудня 2024 р.

Керівник Бакалаврської кваліфікаційної роботи
ст. викладач _____ Міловідов Ю. О.
(наукова ступінь та вчене звання) (підпис) (ПІБ)

Завдання прийняв до виконання
_____ Бондар Микола Юрійович
(підпис) (ПІБ студента)

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

ПК	–	персональний комп’ютер
БД	–	база даних
ТЗ	–	технічне завдання
ОС	–	операційна система
UI	–	Design (User Interface) перекладається як «призначений для користувача інтерфейс». Інтерфейс являє собою графічну структуру програми.
UX	–	(User Experience) перекладається як «користувацький досвід». Визначає чи є інтерфейс інтуїтивно зрозумілим.

ВСТУП

«Людина, яка нікуди не поспішає, нікуди не потрапляє.» Грегорі Девід Робертс. Усі ми іноді поспішаємо, на роботу, на навчання, на ділову зустріч чи на романтичне побачення. Саме в такі моменти ми розуміємо цінність та важливість, такого звичного та зручного, сервісу таксі.

За останні століття технології стрімко змінили наше життя, надаючи нам інструменти для зручності. Практично всі сервіси та послуги підалися різноманітним змінам під впливом цифровізації. Сфера таксі не виключення, на сьогоднішний день ми маємо можливість легко та зручно замовляти таксі прямо із застосунку в телефоні.

З розвитком міської інфраструктури та збільшенням трафіку, пересування великими містами стає дедалі складнішим. У таких умовах мобільні додатки для замовлення таксі стали важливим інструментом, який допомагає економити час, уникати стресу і забезпечувати комфортне пересування. Вони пропонують швидкий доступ до транспорту, можливість планувати маршрути та обирати оптимальні варіанти поїздок.

Актуальність теми дослідження. У сучасному світі, де мобільність і швидкість прийняття рішень мають вирішальне значення, зростає попит на ефективні, зручні та надійні сервіси для замовлення таксі. Розробка таких сервісів сприяє не лише підвищенню комфорту користувачів, але й оптимізації транспортних систем міст.

Розробленість теми. На сьогоднішній день на ринку є багато різноманітних служб таксі: Uklon, Opti, Bolt, OnTaxi, 838, Uber, які задають високі стандарти якості. Варто відзначити, не дивлячись на таку різноманітність сервісів, залишаються деякі невирішені проблеми зокрема

забезпечення доступності послуг у малих містах, підвищення рівня безпеки користувачів і зниження вартості поїздок.

Об’єкт дослідження: система міського транспорту.

Предмет дослідження: проектування та розробка мобільного додатку для служби таксі.

Мета дослідження: створення зручного, функціонального та доступного мобільного додатку для замовлення таксі, який задовольнить потреби користувачів і сприятиме розвитку бізнесу.

Завдання дослідження:

1. Аналіз ринку мобільних додатків для замовлення таксі.
2. Визначення потреб користувачів і бізнесу.
3. Розробка технічного завдання.
4. Проєктування архітектури додатку.
5. Реалізація основних функцій.
6. Тестування та оптимізація додатку.

У цьому дослідженні розробка мобільного додатку для таксі буде спрямована на вирішення актуальних проблем сучасного ринку, забезпечуючи користувачів зручним, безпечним і економічним інструментом для пересування.

Методи дослідження

У цій роботі застосовано такі підходи до дослідження:

Аналіз літературних джерел. Цей метод дав змогу вивчити теоретичні аспекти розробки мобільних додатків, а також ознайомитися з сучасними технологіями та дослідженнями в цій галузі. До аналізу залучено наукові статті, книги, технічну документацію та онлайн-ресурси, що охоплюють програмування, дизайн і користувацький досвід (UX/UI).

Аналіз ринку. Вивчення ринку мобільних додатків для таксі дозволило визначити актуальні потреби користувачів і бізнесу. Було проведено дослідження додатків-конкурентів, їхніх функцій, переваг і недоліків. Зібрано дані про користувацькі відгуки та рейтинги в App Store і Google Play.

Методи проєктування.

Проєктування використовувалося для створення архітектури та структури додатка, включаючи:

- Розробку технічного завдання (ТЗ): визначення функціональних вимог до додатка, його інтерфейсу та взаємодії з користувачем.
- Моделювання архітектури: створення діаграм, блок-схем і моделей, що відображають логічну та фізичну структуру додатка.
- Прототипування: розробку інтерактивних прототипів інтерфейсу для тестування користувацького досвіду до початку програмування.

Програмування.Розробка функціоналу додатка включає:

- Кодування: написання коду на вибраній мові програмування Java для Android.
- Інтеграцію з серверами та базами даних: налаштування обміну даними між додатком і серверною частиною.
- Інтеграцію з зовнішніми сервісами: підключення картографічних API для побудови маршрутів та інших необхідних інструментів.

Тестування. Тестування проводилося для виявлення та виправлення недоліків у роботі додатка, використовуючи:

- Функціональне тестування: перевірку відповідності функцій додатка технічному завданню.

- Тестування інтерфейсу (UX/UI): оцінку зручності та інтуїтивності дизайну.
- Тестування на різних пристроях: перевірку сумісності додатка з різними моделями смартфонів і версіями операційних систем.
- Тестування продуктивності: оцінку швидкості роботи додатка, часу завантаження та обробки запитів.

Застосування цих методів дозволило комплексно підійти до розробки мобільного додатка для служби таксі, забезпечивши його високу якість і відповідність потребам користувачів.

1. СИСТЕМНИЙ АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Опис предметної області

Сфера таксі вже давно стала невід'ємною частиною повсякденного життя. Історія її розвитку сягає кінця XIX століття, коли у 1897 році в німецькому місті Штутгарт з'явилися перші автомобілі, спеціально створені для таксі – Daimler Victoria з бензиновим двигуном. У тому ж році у Лондоні з'явилися перші електричні таксі — Bersey electric cab, які стали першою спробою створення екологічного транспорту.

З роками сфера таксі вдосконалювалась, водії ставали професійнішими, а самі автомобілі — комфортнішими. Проте справжній прорив відбувся у 2009 році з появою Uber — першого мобільного додатку для виклику таксі. Сервіси стали доступнішими, замовлення та оплата — зручнішими. В Україні розвиток мобільних застосунків значно пришвидшився через пандемію COVID-19 та війну, що створили попит на безпечний та автономний громадський транспорт. Таксі стало одним з небагатьох способів пересування під час комендантської години. Таким чином, на сучасному етапі сфера таксі охоплює цифрові технології, мобільні сервіси, картографічні рішення та електронні платежі, формуючи нову якість транспортного обслуговування.

1.2 Аналіз вимог до програмної системи

Після аналізу сучасних служб таксі виявлено ряд проблем, які стають причиною незадоволеності користувачів:

- Перевантажений інтерфейс деяких застосунків (наприклад, Bolt);
- Нестабільна робота у години пік (Uklon, Bolt);
- Обмежена доступність авто в невеликих містах (Opti, 838);
- Різнорівнева якість обслуговування.

З урахуванням цих недоліків, система, яку планується розробити, має задовольняти наступні вимоги:

- Інтуїтивно зрозумілий інтерфейс, без зайвих функцій;
- Висока стабільність роботи за будь-яких умов;
- Мінімальний час очікування авто навіть у години пік;
- Надійність та простота навігації для користувача;
- Підтримка основних картографічних сервісів;
- Інтеграція з платіжними системами;
- Можливість адаптації під користувачів у різних регіонах.

1.3 Моделювання предметної області

Моделювання предметної області передбачає створення узагальненої картини взаємодії між основними суб’єктами системи: пасажир – додаток – водій – сервер – аналітика.

Основні компоненти:

- Користувачі (пасажири, водії);
- Мобільний додаток (замовлення, відстеження, оплата);
- Серверна частина (обробка запитів, маршрутизація, аналітика);
- Картографічний модуль (визначення маршрутів, розрахунок часу);
- Модуль управління замовленнями (призначення авто, зворотний зв’язок);
- Модуль адміністратора (керування службою, аналітика).

1.4 Огляд інформаційних джерел та існуючих рішень

На ринку України функціонують п’ять основних служб таксі таб.1.

Таблиця 1

Сервіс	Переваги	Недоліки
Uklon	Місцева орієнтація, точний розрахунок вартості	Високі тарифи в години пік
Bolt	Доступність у багатьох містах, екоініціативи	Перевантажений інтерфейс, нестабільність
Opti	Найнижчі ціни	Обмежена кількість авто
838	Замовлення через дзвінок	Обмежений вибір, довге очікування
OnTaxi	Вибір авто, хороші ціни	Збої у малих містах

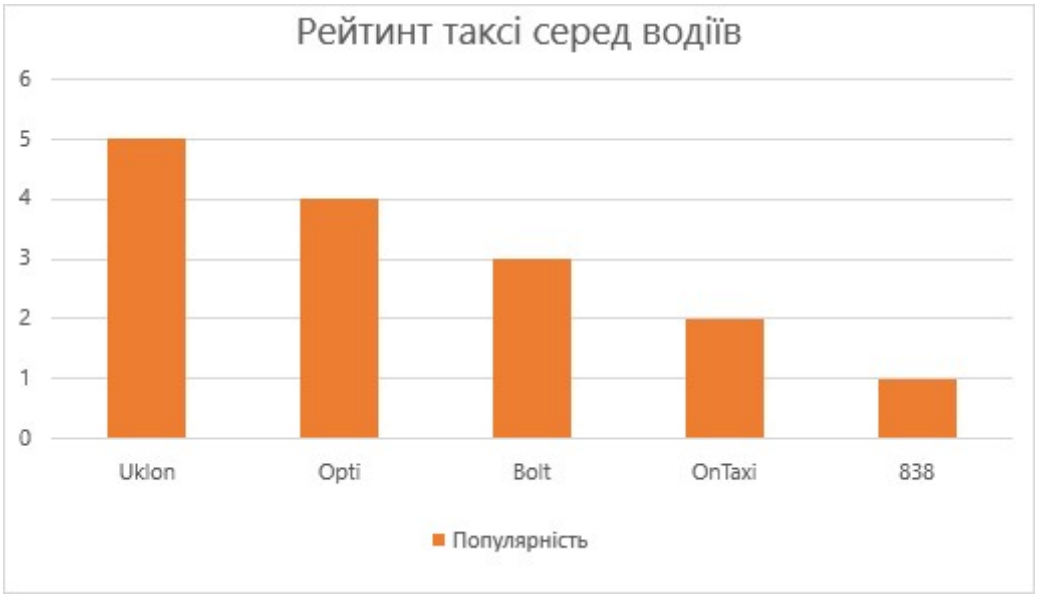


Рис.1 Рейтинг таксі серед водіїв

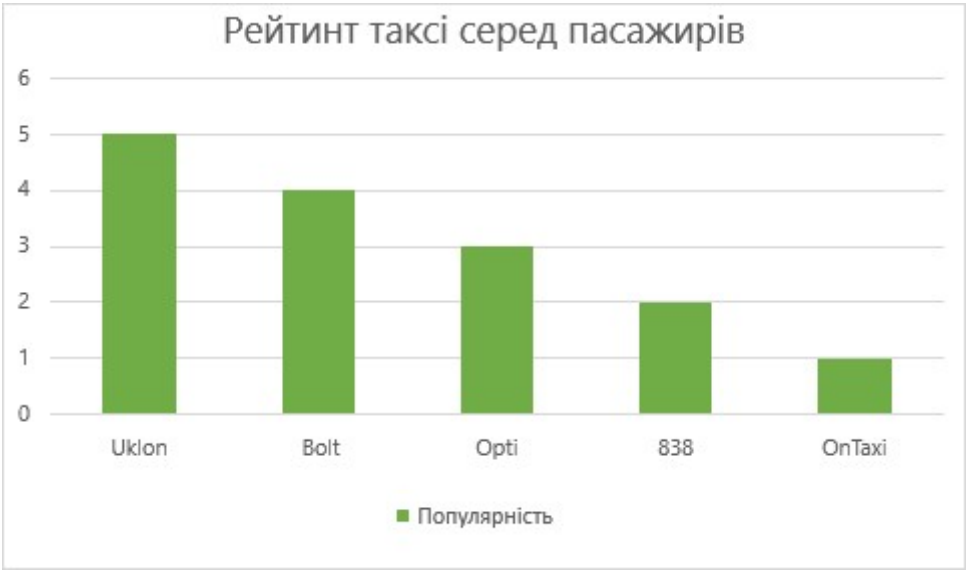


Рис.2 Рейтинг таксі серед пасажирів

1.5 Постановка завдання.

Сучасний ритм життя вимагає швидких і зручних рішень для пересування містом. Враховуючи зростаючу потребу в ефективних транспортних послугах, мобільний додаток "Таxі" покликаний забезпечити користувачам швидкий, безпечний і зручний спосіб замовлення таксі.

Мета розробки "Таxі" – створення інтуїтивно зрозумілого та надійного сервісу для взаємодії пасажирів і водіїв, що дозволить оптимізувати процес пошуку транспорту, зменшити час очікування та покращити загальний досвід користувачів. Додаток надасть можливість вибору типу автомобіля, відстеження маршрутута комунікації між водієм і пасажиром.

Найменування програмного продукту – "Таxі". Додаток розробляється для мобільних пристроїв на платформі Android. Основна область застосування – міські та міжміські перевезення.

Підставою для розробки є зростаючий попит на зручні сервіси виклику таксі, а також необхідність оптимізації роботи таксопарків.

Додаток призначений для швидкого виклику таксі, вибору типу автомобіля, відстеження маршруту в реальному часі та забезпечення безпечної комунікації між водієм і пасажиром.

Вимоги до функціональних характеристик:

- Реєстрація та авторизація користувачів (пасажирів та водіїв).
- Введення маршруту (вибір початкової та кінцевої точки через карту, введення вручну або автоматичне визначення геолокації).
- Вибір типу автомобіля (економ, універсал, мінівен, бізнес).
- Розрахунок вартості поїздки перед її підтвердженням.
- Відстеження водія на карті після підтвердження замовлення.

- Можливість зв’язку між пасажиром і водієм через внутрішній чат.

Вимоги до надійності:

- Авторизація через Google акаунт.
- Захист від некоректних дій користувачів (спам-замовлення, шахрайство тощо).
- Відновлення роботи після збою без втрати даних.

Умови експлуатації:

- Регулярне оновлення даних про доступність водіїв та тарифи.
- Безперервна робота серверної частини для обробки запитів.
- Сумісність із сучасними мобільними пристроями.

Вимоги до складу й параметрів технічних засобів:

- Об’єм оперативної пам’яті: від 1 ГБ.
- Вільне місце на пристрої: від 200 МБ.
- Підключення до Інтернету для роботи з картами та сервісами геолокації.

Вимоги до інформаційної й програмної сумісності:

- Підтримка ОС Android (версія 8.0 і вище).
- Інтеграція з картографічними сервісами (Google Maps, Apple Maps).

Контроль якості програмного продукту здійснюється на кожному етапі розробки, зокрема:

- Перевірка відповідності функціоналу початковим вимогам.
- Тестування на стабільність роботи та безпеку даних.
- Аналіз продуктивності та навантаження сервера.
- Оцінка зручності користування додатком (UI/UX-тестування).

- Приймання програмного забезпечення здійснюється після успішного проходження всіх тестів та відповідності початковим вимогам.

Перед випуском додатка буде проведено фінальне тестування, після чого "Таxі" буде доступний для завантаження користувачами.

2. РОЗРОБКА АРХІТЕКТУРИ ТА СТРУКТУРНОЇ ОРГАНІЗАЦІЇ ПРОГРАМНОГО ПРОДУКТУ

2.1 Проектування програмного продукту

В попередньому розділі ми визначили основні недоліки мобільних додатків наявних сервісів таксі, виходячи з цієї інформації ми можемо скласти перелік функцій та сценаріїв які будуть реалізовуватись в моєму додатку.

Варіанти використання ПЗ.

Для користувача.

- Користувач відкриває додаток "Таxі".
- Після реєстрації або входу в обліковий запис, користувач переходить до головного вікна.
- Користувач вказує адресу початку та завершення маршруту, обираючи зручний спосіб: через мітки на карті або введенням адреси вручну. Також є можливість встановити початкову точку маршруту за поточною геолокацією.
- Обирає певний тип авто(економ, універсал, мінівен, бізнес).
- Підтверджує замовлення.
- Користувач має змогу зв’язатися з водієм, через внутрішній чат.
- Користувач має змогу скасувати поїздку.

Для водія.

- Водій відкриває додаток "Таxі".
- Після реєстрації або входу в обліковий запис, переходить до головного вікна.

- Водій має можливість переглядати наявні замовлення для його типу автомобіля (економ, універсал, мінівен, бізнес), з інформацією про вартість поїздки та відстанню до клієнта.
- Приймає замовлення, та переходить на вікно з інформацією про замовлення, та картою з прокладеним маршрутом до клієнта.
- Водій має змогу зв’язатися з клієнтом, через внутрішний чат.
- Водій має змогу скасувати поїздку.

Отже мій додаток має виконувати наступні функції (Рис. 2):

a) Авторизація користувача.
b) Введення маршруту – вибір початкової та кінцевої точки поїздки.
c) Вибір типу авто – економ, універсал, мінівен, бізнес.
d) Замовлення таксі – підтвердження замовлення та очікування водія.
e) Зв’язок– внутрішній чат для уточнення деталей.
f) Перегляд доступних замовлень – список поїздок, що відповідають типу авто
g) Прийняття замовлення – підтвердження готовності виконати поїздку.
h) Маршрутизація – перегляд інформації про клієнта та маршруту на карті.
i) Скасування поїздки – можливість відмовитися від поїздки.

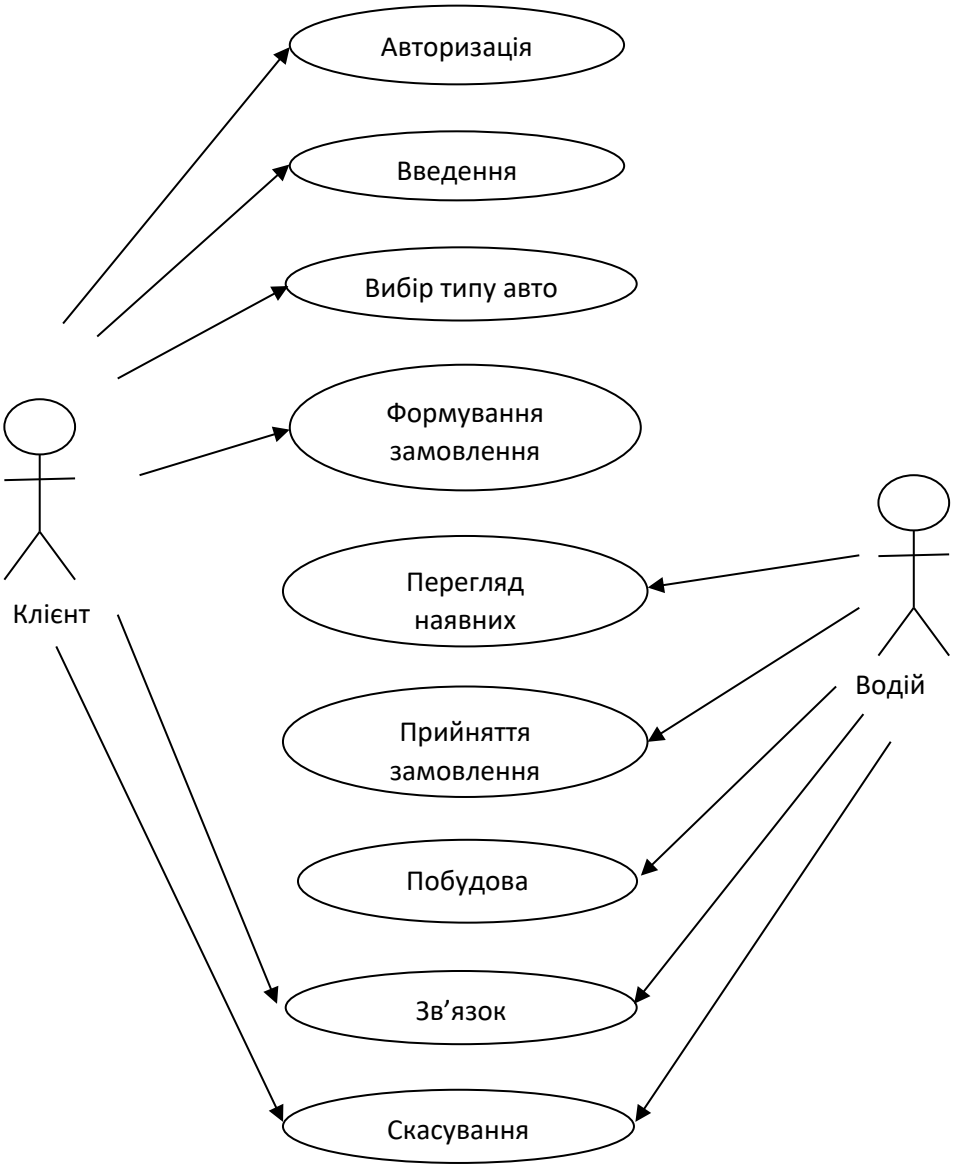


Рис. 3 Діаграма прецедентів додатку «Таxі»

Створення моделі розробки програмного продукту

V-подібна модель (Рис. 4) розробки програмного продукту є розширенням водоспадної моделі, що поєднує етапи розробки і тестування.Вона отримала свою назву через форму букви "V", де кожен етап розробки має відповідний етап тестування, що дозволяє виявляти помилки на ранніх стадіях.

Процес починається з аналізу вимог, збираються дані про потреби користувачів та визначаються основні функціональні вимоги.

Наступним етапом є проектування системи, на якому розробляється загальна архітектура продукту.

Після цього детально проектується програмне забезпечення, де визначаються алгоритми та структури даних.

Після завершення етапу проектування розпочинається написання коду, під час якого розробники реалізують функціональність продукту. Паралельно з цим проводиться юніт-тестування, яке перевіряє окремі компоненти.

Коли всі модулі інтегровані, проводиться інтеграційне тестування, щоб перевірити взаємодію компонентів.

Далі продукт проходить валідацію, коли тестується його відповідність вимогам користувачів. Після успішного тестування продукт готовий до впровадження в реальне середовище. В кінці процесу здійснюється підтримка продукту, що включає виправлення помилок і оновлення.

V-подібна модель дозволяє знижувати ризики, оскільки помилки виявляються на кожному етапі. Вона особливо корисна для проектів з чіткими вимогами та високими вимогами до якості.

Деталізований опис роботи V-подібної моделі

V-подібна модель передбачає дві основні частини: ліву сторону, яка охоплює етапи аналізу та проектування, і праву сторону, яка зосереджена на тестуванні та валідації системи.

Ліва сторона V-моделі (Етапи планування та розробки)

- *Планування проекту та визначення вимог.* На цьому етапі команда розробників спільно із зацікавленими сторонами проводить зустрічі, інтерв'ю та опитування для збору інформації про очікування від майбутнього продукту. Визначаються основні функціональні та нефункціональні вимоги, бізнес-цілі, а також обмеження, що можуть впливати на процес розробки. Результатом цього етапу є створення документації з вимогами, яка стане основою для подальших етапів розробки.
- *Аналіз вимог та створення специфікацій.* Отримані вимоги аналізуються, уточнюються та документуються у вигляді специфікації вимог до програмного забезпечення. Визначаються критерії прийнятності, які використовуватимуться на етапі тестування. На цьому етапі також оцінюється можливість реалізації вимог, враховуючи технічні, часові та ресурсні обмеження.
- *Розробка архітектури системи.* Визначається загальна структура програмного продукту, розробляється архітектурний дизайн. Створюються високорівневі діаграми, що описують взаємодію основних компонентів системи. Визначаються технології, які будуть використані для реалізації продукту. Основна увага приділяється забезпеченню масштабованості, продуктивності та безпеки системи.
- *Деталізоване проектування компонентів.* Кожен компонент системи розробляється окремо, включаючи його внутрішню логіку,

алгоритми, інтерфейси та взаємодію з іншими модулями. Визначається структура баз даних, створюються специфікації для модулів, API та інтерфейси користувача. Готується документація, що містить детальні описи програмних компонентів, що стане основою для подальшого етапу кодування.

Права сторона V-моделі (Етапи тестування та перевірки)

- *Модульне тестування.* Кожен компонент або модуль тестується окремо, щоб перевірити його коректну роботу відповідно до специфікацій. Використовуються юніт-тести, що дозволяють перевірити правильність функціонування окремих функцій та методів. Основна мета – виявити та виправити помилки на рівні окремих модулів ще до інтеграції в загальну систему.
- *Інтеграційне тестування.* На даному етапі тестується взаємодія між різними модулями системи. Перевіряється коректність обміну даними між компонентами, відповідність архітектурним вимогам та узгодженість між підсистемами. Використовуються різні підходи, такі як тестування зверху вниз, знизу вгору.
- *Системне тестування.* Повністю інтегрована система тестується на відповідність усім специфікованим вимогам. Виконується функціональне тестування (перевірка роботи системи відповідно до вимог) та нефункціональне тестування (продуктивність, безпека, навантаження тощо). Основна мета – переконатися, що система працює стабільно та виконує всі необхідні функції.
- *Кінцеве тестування та приймання системи.* Проводиться тестування продукту кінцевими користувачами або замовниками. Оцінюється відповідність системи бізнес-вимогам, зручність використання та стабільність роботи. Якщо система проходить всі

перевірки, вона отримує схвалення для впровадження та переходить у стадію експлуатації.

Рис. 4 V-подібна модель

2.2 Графічне моделювання архітектури та структури програмного продукту.

Створення структурної схеми програмного продукту.

Структурна схема (Рис. 5) є важливим інструментом для візуалізації компонентів системи та їх взаємозв'язків. Для додатку "Таxі", призначеного для виклику таксі, структурна схема може бути поділена на кілька основних модулів: підсистема візуалізації, підсистема замовлень, підсистема зберігання даних, підсистема управління поїздками.

Підсистема візуалізації. Цей компонент відповідає за взаємодію користувача з додатком. Вона має окремі функціональні можливості для пасажира та водія.

Для пасажира:

- Реєстрація та Вхід – форми для створення облікового запису та входу до системи.
- Вибір маршруту – можливість вказати адресу початку та завершення поїздки, вибрати тип автомобіля (економ, бізнес, мінівен тощо).
- Оформлення замовлення – формування замовлення за вказаними параметрами адреси та типу авто.
- Зв’язок з водієм – внутрішній чат.
- Відстеження автомобіля – перегляд розташування водія на карті в реальному часі.

Для водія:

- Реєстрація та Вхід – вхід у систему.
- Список доступних замовлень – перегляд активних замовлень, відстані до клієнта та орієнтовної вартості поїздки.
- Прийом замовлення – можливість прийняти або відхилити замовлення.
- Маршрут до клієнта – автоматичне прокладання маршруту до місця посадки.
- Зв’язок із пасажиром – внутрішній чат.
- Завершення поїздки – підтвердження виконання замовлення, отримання оплати.

Підсистема замовлень. Цей компонент відповідає за обробку та управління всіма поїздками.

Для пасажира:

- Створення замовлення.
- Розрахунок вартості поїздки.

- Можливість скасування замовлення.

Для водія:

- Перегляд доступних замовлень.
- Отримання інформації про клієнта.
- Прийом та виконання замовлення.

Структурна схема додатку "Taxi" дозволяє чітко визначити взаємодію між його компонентами та користувачами. Завдяки розподілу функціональності між пасажирами та водіями, додаток забезпечує зручний та ефективний сервіс для всіх учасників процесу.

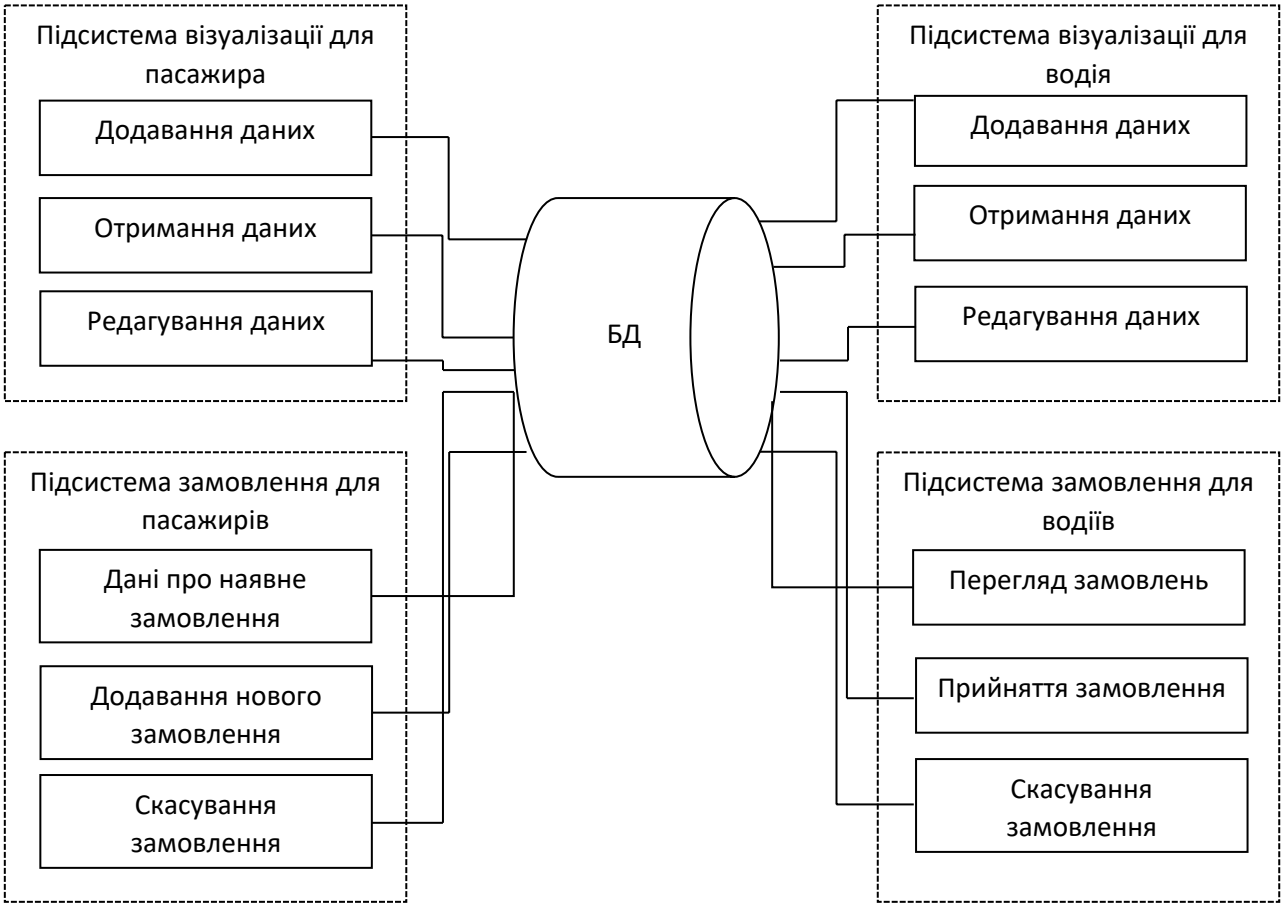


Рис. 5 Структурна схема додатку

Моделювання структури бази даних

У процесі розробки інформаційної системи для мобільного застосунку таксі надзвичайно важливим етапом є проєктування структури бази даних.

Одним із найкращих способів візуалізації логічної моделі даних є використання ER-діаграм (діаграм «сутність–зв’язок»), які дозволяють чітко окреслити основні об’єкти системи, їх характеристики та взаємозв’язки.

На рис. 6 представлено ER-діаграму, яка моделює ключові компоненти системи, зокрема користувачів (пасажирів і водіїв), замовлення поїздок, транспортні засоби та маршрути. Побудова такої діаграми дозволяє глибше зрозуміти структуру даних, що будуть використовуватися під час функціонування додатку, а також слугує основою для подальшого створення фізичної моделі бази даних.

Метою створення ER-діаграми є забезпечення логічної узгодженості між усіма елементами системи, оптимізація зберігання даних та полегшення майбутньої реалізації функціоналу мобільного застосунку.

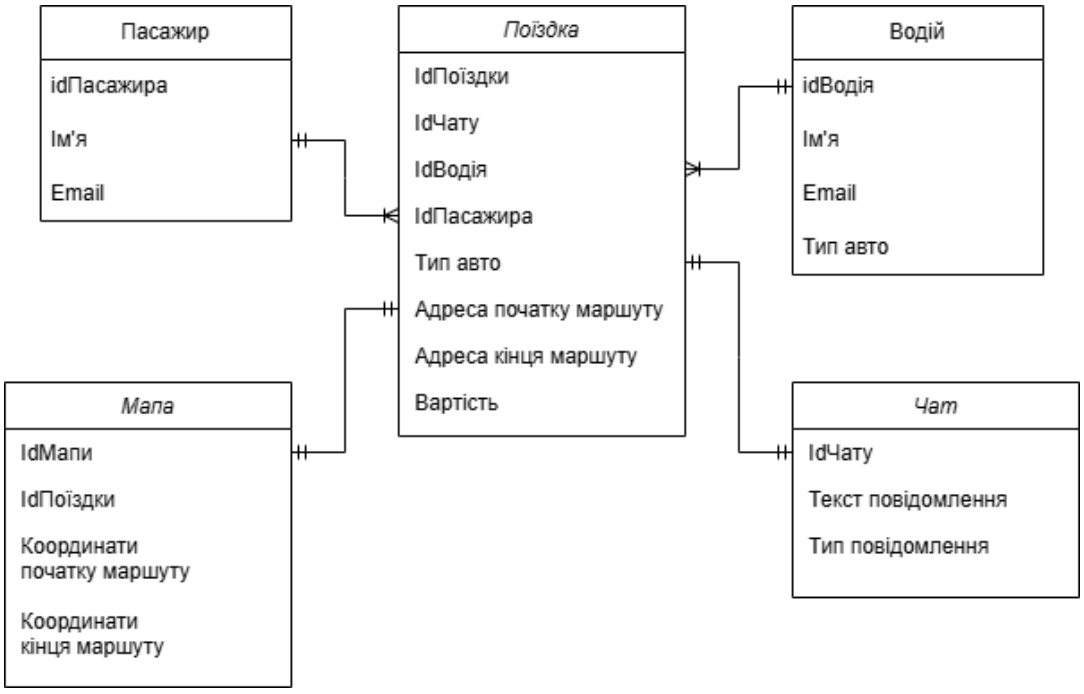


Рис. 6 Er-діаграма

Створення діаграми класів.

Після визначення логічної структури даних за допомогою ER-діаграми наступним кроком у процесі розробки мобільного застосунку таксі є створення діаграми класів.

Діаграма класів є одним із ключових інструментів, який дозволяє моделювати програмну архітектуру системи шляхом визначення основних класів, їх атрибутів, методів та взаємозв’язків між ними.

На рис. 7 представлено діаграму класів, що відображає структуру об'єктів мобільного застосунку таксі. Зокрема, описано класи, що відповідають за користувачів (водіїв та пасажирів), обробку замовлень, транспортні засоби, маршрути, оплату тощо.

Розробка діаграми класів дозволяє формалізувати структуру коду, спростити процес реалізації та забезпечити зрозумілу й підтримувану архітектуру додатку. Вона також є важливою основою для подальшої імплементації бізнес-логіки та взаємодії між компонентами системи.

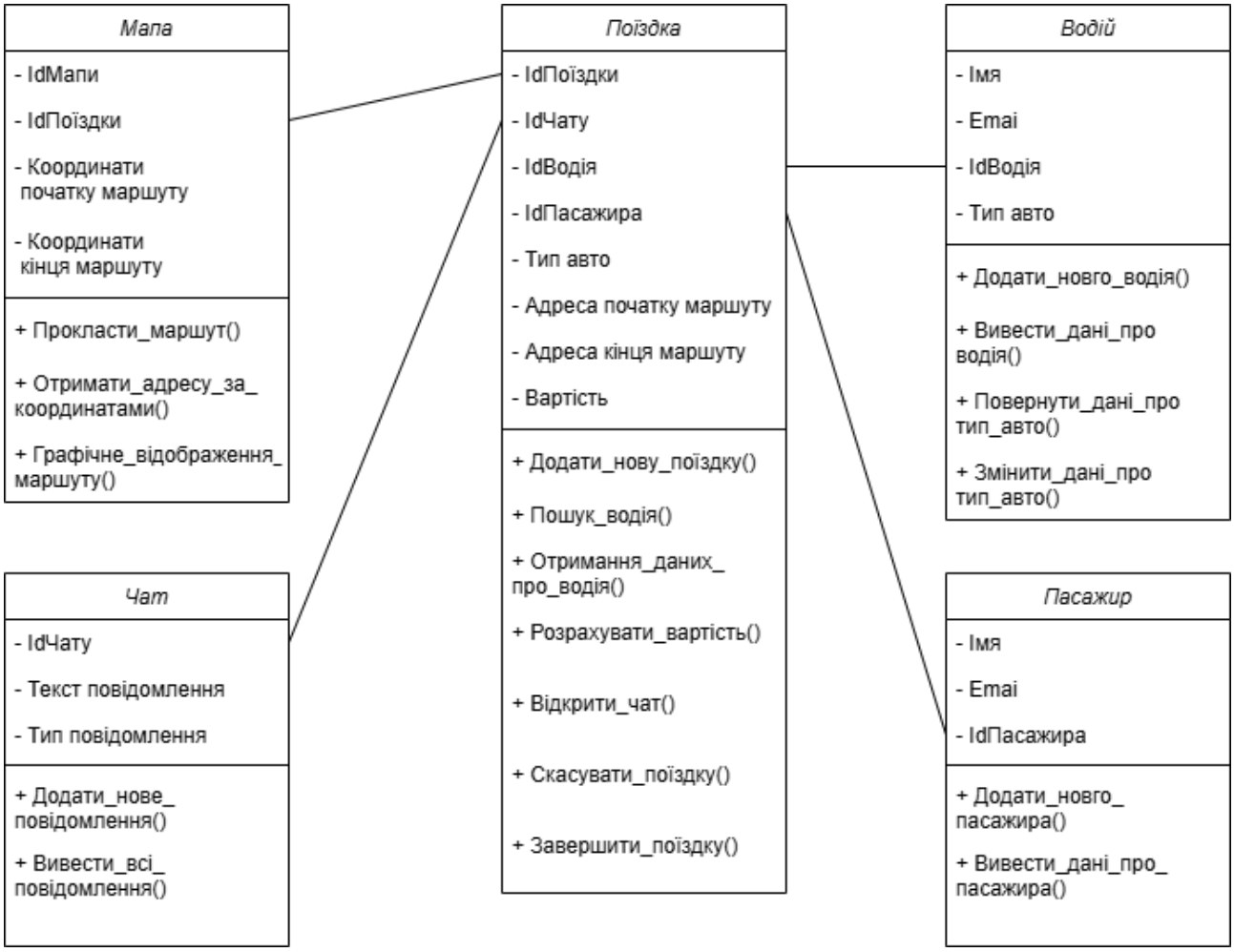


Рис. 7 Діаграма класів

Створення функціональної схеми програмного продукту.

Функціональна схема є графічним представленням роботи системи, що демонструє її основні компоненти та їх взаємодію. Вона відображає ключові функції додатку, їхню послідовність та взаємозв’язки між різними модулями. Блок-схема (Рис. 8.1, Рис. 8.2) наочно ілюструє алгоритм роботи додатку, що допомагає краще зрозуміти його функціонування. Це сприяє виявленню можливих проблем і вдосконаленню системи ще на етапі проектування.

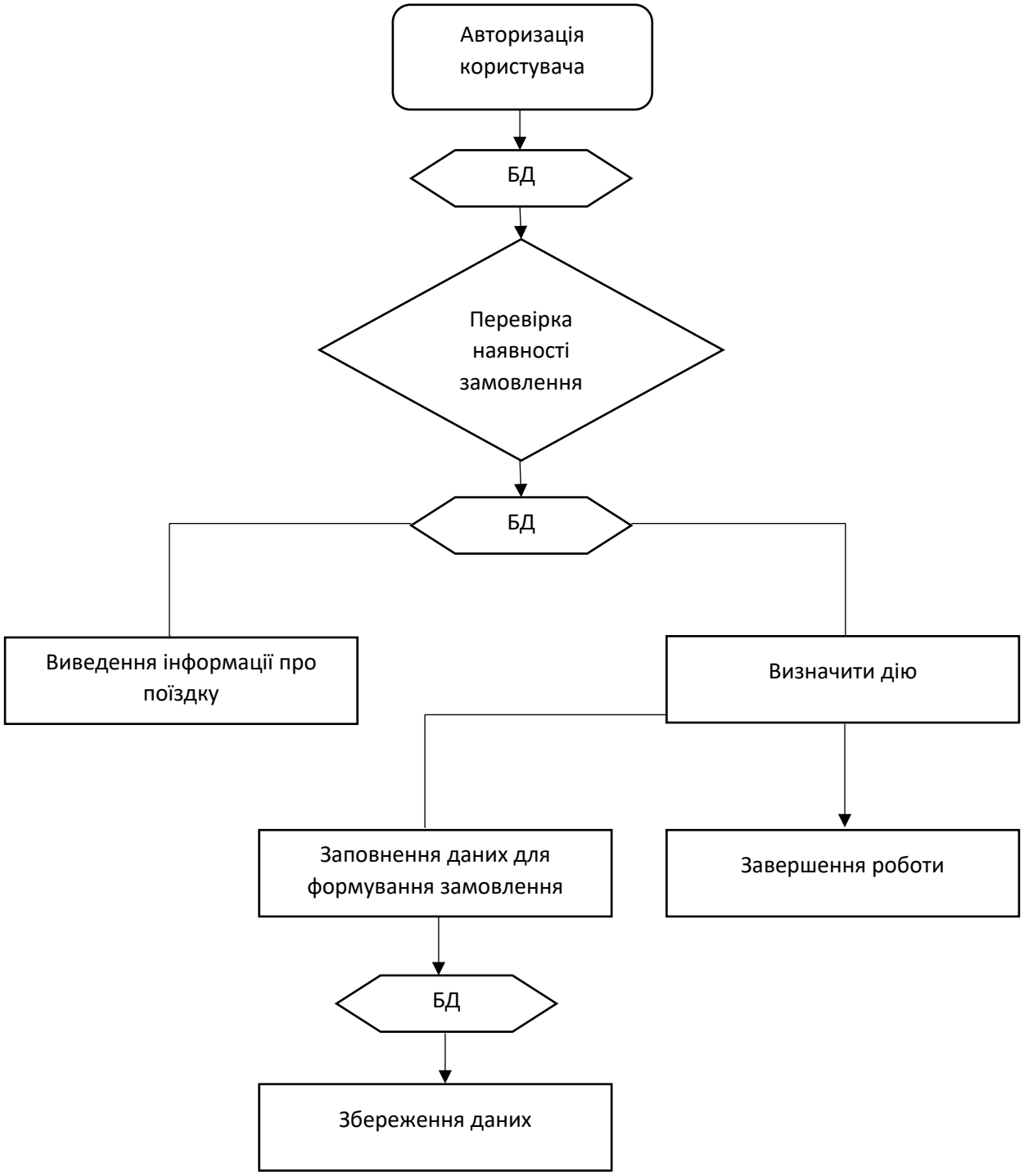


Рис. 8.1 Функціональна блок-схема для пасажира

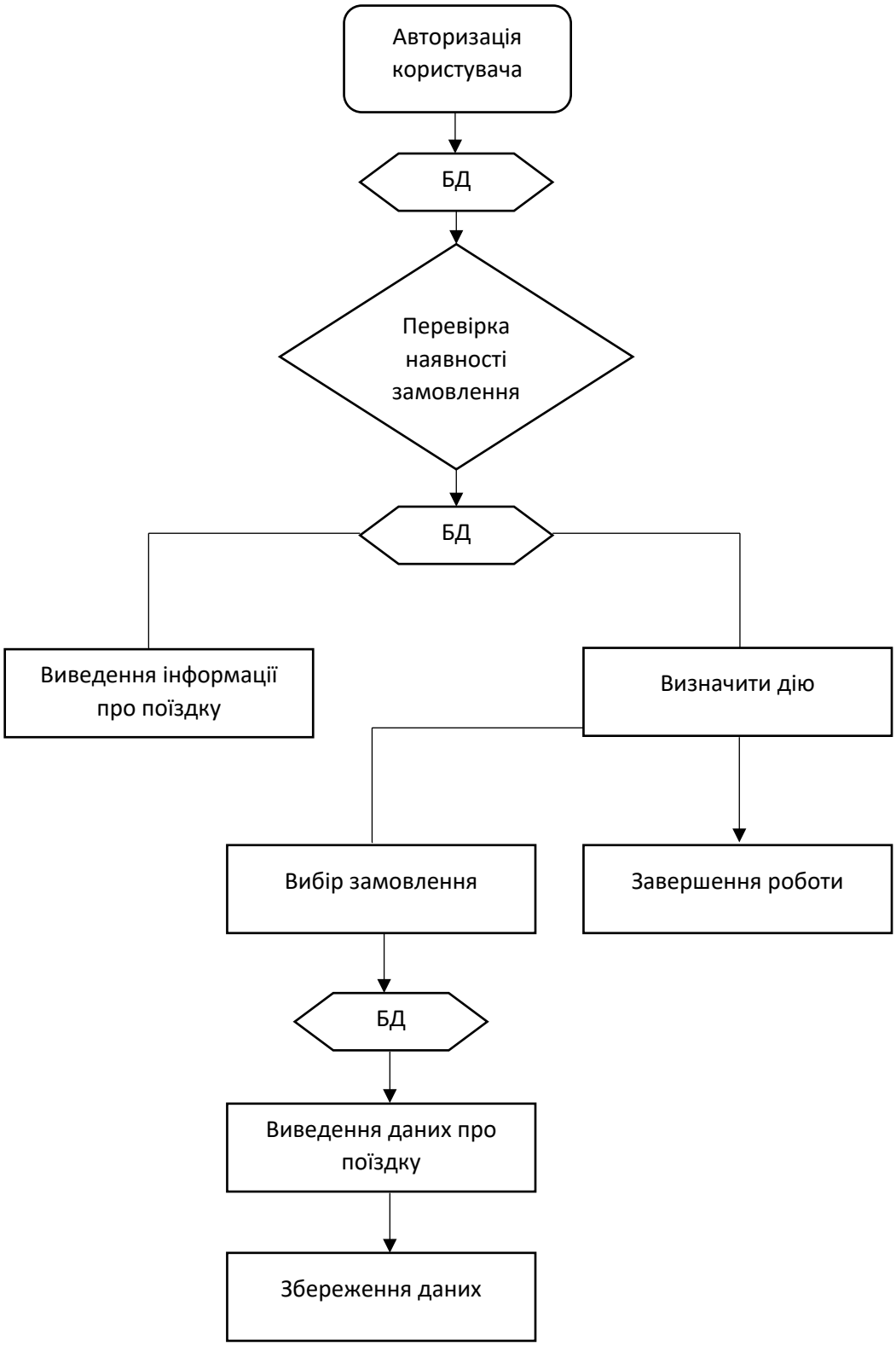


Рис. 8.2 Функціональна блок-схема для водія

Створення алгоритму функціонування програмного продукту та його підпрограм.

Алгоритм роботи додатку "Таxі" для пасажира.

1) Користувач відкриває мобільний додаток "Таxі". Завантажуються початкові дані (попередньо збережені сесії).

2) Якщо це перший запуск програми користувач проходить реєстрацію за допомогою облікового запису Google. Додаток надсилає ці дані на бекенд-сервер. Сервер перевіряє унікальність даних. Успішна реєстрація зберігає дані користувача в базі даних.

3) Далі відбувається аутентифікація користувача додаток надсилає дані на бекенд-сервер для перевірки. Сервер аутентифікує користувача, перевірає чи є не завершені поїзди, якщо є то відкриває вікно з наявною поїздкою.

4) Користувач має можливість переглядати карту та вибирати початкову точку маршруту або місце призначення за допомогою мітки.

5) Користувач може визначити початок маршруту, використовуючи свою поточну геолокацію.

6) Користувач може ввести адресу точок початку і закінчення маршуту.

7) Користувач може обрати тип авто.

8) Після формування замовлення, додаток надсилає дані у БД. Сервер зберігає інформацію про замовлення в базі даних.

9) Користувач може зв’язатися з водієм.

10) Користувач має можливість скасувати поїздку, після чого форма буде очищена для нового заповнення.

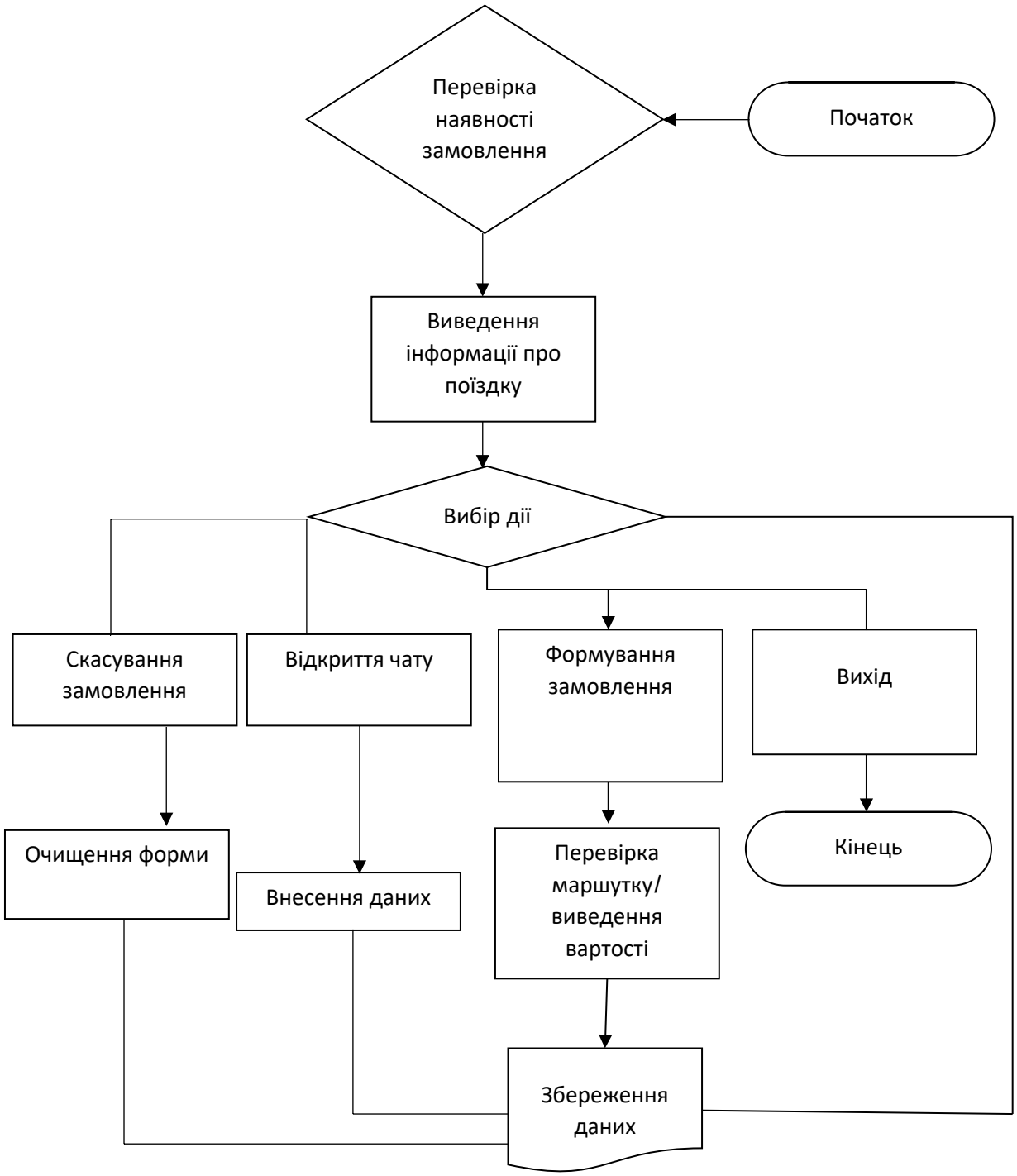


Рис. 9.1 Алгоритм роботи додатку для пасажира

Алгоритм роботи додатку "Таxі" для водія.

1) Користувач відкриває мобільний додаток "Таxі". Завантажуються початкові дані (попередньо збережені сесії).

2) Якщо це перший запуск програми користувач проходить реєстрацію за допомогою облікового запису Google. Додаток надсилає ці дані на бекенд-сервер. Сервер перевіряє унікальність даних. Успішна реєстрація зберігає дані користувача в базі даних.

3) Далі відбувається аутентифікація користувача додаток надсилає дані на бекенд-сервер для перевірки.

4) Додаток завантажує дані з БД про поїздки для певного типу авто, та виводить дані для кожного замовлення про його вартість та відстань до клієнта.

5) Користувач може вибарти найраще для себе замовлення

6) Після підтвердження замовлення, додаток надсилає дані у БД. Сервер зберігає інформацію про зміну типу замовлення (на водій в дорозі) в базі даних.

7) Користувач може зв’язатися з пасажиром.

8) Користувач має можливість скасувати поїздку, після чого додаток відриє вікно з списком наявних поїздок.

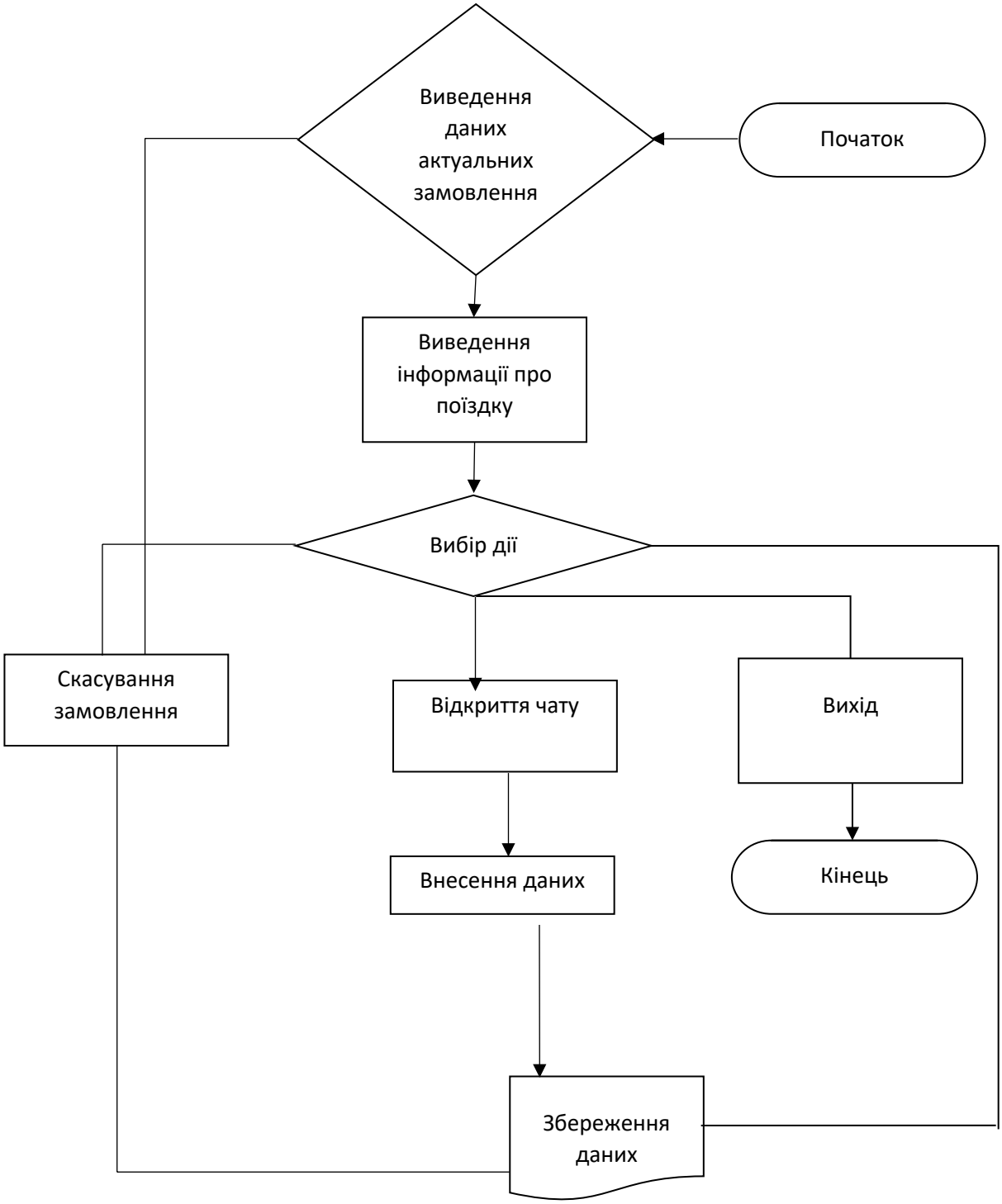


Рис. 9.2 Алгоритм роботи додатку для водія

2.3 Оцінка інструментів для розробки

Обґрунтування вибору апаратних засобів.

Вибір хмарної платформи.

Основою серверної частини застосунку є Firebase — потужна платформа розробки мобільних та веб-додатків від Google. Вона забезпечує ефективну взаємодію клієнтів із сервером, збереження даних та їхню синхронізацію в реальному часі.

Основні сервіси Firebase, що використовуються у додатку:

- Realtime Database – хмарна NoSQL-база даних, що дозволяє зберігати дані у форматі JSON та синхронізувати їх між клієнтами в реальному часі.
- Firebase Authentication – система автентифікації користувачів через email, телефонний номер або акаунти Google/Facebook.

Основні переваги Firebase Realtime Database для застосунку "Taxi"

- Синхронізація в реальному часі – всі зміни в даних миттєво відображаються у клієнтських додатках без потреби перезавантаження.
- Гнучка структура NoSQL – зберігання даних у JSON-форматі забезпечує швидкий доступ та масштабованість.
- Безпека та контроль доступу – налаштування дозволів.
- Масштабованість – Firebase автоматично адаптується до збільшення кількості користувачів без додаткових серверних налаштувань.

Використання картографічних сервісів.

Однією з ключових функцій мобільного застосунку є відображення карт, пошук маршрутів та визначення місцезнаходження користувачів. Для цього інтегровано Google Maps API та Google Geocoding API.

Google Maps API забезпечує:

- Відображення інтерактивної карти з можливістю масштабування та навігації.
- Позначення точок початку та кінця маршруту.
- Відображення розташування автомобіля водія в реальному часі.

Google Geocoding API використовується для:

- Перетворення введеної адреси в географічні координати (широту та довготу).
- Визначення адреси за координатами GPS (зокрема, автоматичне визначення поточного місцезнаходження користувача).

Завдяки цим сервісам система автоматично розраховує оптимальний маршрут для водія, що значно покращує швидкість обслуговування замовлень та знижує час очікування клієнтів.

Системні вимоги для мобільних пристроїв

Для коректної роботи застосунку необхідний мобільний пристрій з такими характеристиками:

- Операційна система: Android 8.0 або вище.
- Оперативна пам’ять: мінімум 1 ГБ (рекомендовано 2 ГБ і більше).
- Вільне місце на пристрої: не менше 128 МБ.
- Наявність модуля GPS: для визначення геолокації користувача.
- Інтернет-з’єднання: для взаємодії з сервером та обміну даними в режимі реального часу.

Висновок

Розробка мобільного додатку для служби таксі є складним, але вкрай важливим процесом, що вимагає врахування безлічі факторів – від зручності інтерфейсу користувача до ефективності алгоритмів розрахунку маршрутів і вартості поїздки.

У цьому розділі було розглянуто основні функціональні вимоги до системи, її архітектуру, а також механізми взаємодії між водіями та пасажирами.

Однією з ключових особливостей додатку є його зручність та інтуїтивність. Реалізація зрозумілого інтерфейсу, можливість швидкого виклику таксі, відстеження автомобіля в реальному часі роблять додаток ефективним інструментом як для пасажирів, так і для водіїв.

Використання сучасних технологій і фреймворків дозволяє оптимізувати роботу алгоритмів та забезпечити швидкий обмін даними між усіма учасниками сервісу.

Важливу роль відіграє і система геолокації, яка дозволяє точно визначати місцезнаходження користувачів та прокладати оптимальні маршрути.

Не менш вадливим є забезпечення безперебійної роботи серверів та оптимізація навантаження, щоб уникнути збоїв при великій кількості запитів.

Мій мобільний додаток для служби таксі – це не просто програмний продукт, а комплексне рішення, яке поєднує в собі інноваційні технології, аналіз користувацького досвіду та ефективні алгоритми управління перевезеннями. Його успішна реалізація дозволяє покращити якість транспортних послуг, підвищити рівень комфорту для користувачів та забезпечити ефективну роботу служби таксі в умовах сучасного цифрового середовища.

3. РОЗРОБКА МОБІЛЬНОГО ЗАСТОСУНКУ

3.1 Вибір середовища розробки

Після детального аналізу сфери послуг таксі та вивчення сценаріїв використання мобільного додатку, я визначив основні критерії та вимоги до середовища розробки. Головним пріоритетом було забезпечення підтримки мобільної ОС Android, оскільки саме для неї розроблявся додаток. Крім того, важливою вимогою стала можливість легкої інтеграції з хмарною БД, що забезпечує збереження та обробку інформації про замовлення, водіїв і клієнтів. Найкращим вибором для розробки став «Android Studio».

Основні переваги цього середовища:

- **Інтеграція з IntelliJ IDEA**. «Android Studio» базується на IntelliJ IDEA, що надає широкий набір інструментів для написання, тестування та налагодження коду.
- **Інтелектуальний редактор коду**. Завдяки автозавершенню, рефакторингу та аналізу коду, процес розробки стає більш ефективним і швидким.
- **Вбудований емулятор**. Це дозволяє тестувати додаток на різних версіях Android та пристроях без фізичного доступу до них.
- **Розширені інструменти налагодження**. Наявність профайлера продуктивності, засобів для аналізу використання пам’яті та CPU дозволяє оптимізувати роботу додатку.
- **Інтеграція з системами контролю версій**. Підтримка Git спрощує управління версіями коду та співпрацю з іншими розробниками.

- **Інтеграція з Firebase**. Легка інтеграція з «Firebase» дозволяє додати БД, аутентифікацію користувачів, що є важливими функціями для додатку таксі.
- **Jetpack Compose**. Сучасний підхід до створення інтерфейсу користувача, який значно спрощує розробку та дозволяє швидко оновлювати UI.
- **Visual Layout Editor.** Інструмент, що дозволяє створювати інтерфейси за допомогою drag-and-drop без необхідності вручну писати XML-код.
- **Gradle Build System.** Потужний механізм збірки, який дозволяє легко керувати залежностями, конфігураціями та автоматизувати процеси розробки.
- **Інтеграція з Google Maps API**, що дозволяє легко додавати карти, маршрути та визначати місцеперебування користувача.
- **Підтримка GPS та інших сенсорів**, що критично для роботи додатків таксі.

3.2 Авторизація користувача

Під час відкриття додатка користувач спочатку потрапляє на початковий екран (рис. 10). Після цього автоматично з’являється вікно авторизації через Google (рис. 11), де користувач може вибрати акаунт для входу в систему. Якщо додаток уже було відкрито раніше, то автентифікація виконується автоматично, використовуючи останню збережену електронну адресу. Це дозволяє зменшити час входу та забезпечує зручність користування, адже користувачеві не потрібно щоразу вводити свої облікові дані вручну. Такий підхід також підвищує рівень безпеки, оскільки використовує надійний механізм автентифікації від Google.

Рис. 10 Початковий екран

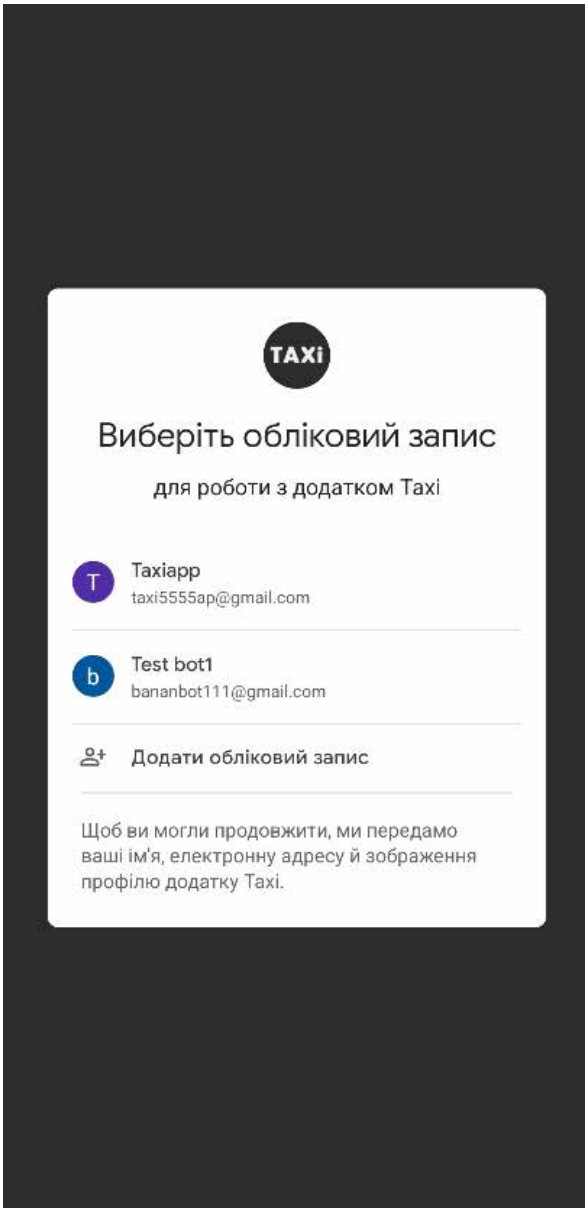


Рис. 11 Вікно авторизації Google

Для кращого усвідомлення процесів які відбуваються при авторизації користувача варто поглянути на алгоритм роботи, що представлений схемою рис. 12.

Рис. 12 Схема авторизації користувача

Для кращого розуміння роботи додатку розглянемо методи що забезпечують процес авторизації користувача. Скріншоти з кодом даного вікна подані в додатку А рис. 13.1-13.5.

Перший фрагмент коду рис. 13.1, створений для перевірки чи користувач увійшов до свого акаунту раніше. Якщо користувач не увійшов, виконується функція singIn(). Якщо користувач вже заходив до свого акаунту раніше, даний метод отримує його e-mail, для виконання запиту chekUserInDatabase(email).

Наступний фрагмент коду рис. 13.2 відповідає за процес аутентифікації користувача через Google. Процес починається з виклику методу signIn(), який запускає вікно входу Google. Далі, після взаємодії користувача, спрацьовує метод onActivityResult(), який отримує результат авторизації. Якщо вхід успішний, програма зчитує електронну адресу користувача та перевіряє її у базі даних за допомогою функції checkUserInDatabase(). Якщо ж авторизація не вдалася, програма намагається повторити вхід.

Наступний фрагмент коду рис. 13.3 відповідає за перевірку наявності користувача в базі даних Firebase при вході в додаток для таксі.

Його основне завдання – визначити, чи є дані в БД про користувача. Спочатку метод checkUserInDatabase(String email) звертається до бази даних Firebase і виконує запит, шукаючи запис із відповідною електронною адресою. Якщо такий запис існує, здійснюється додатковий запит для отримання типу користувача (type). Після отримання цього параметра викликається функція navigateToSecond(typeValue), яка визначає, куди саме слід направити користувача в додатку (відкрити вікно користувача чи водія). Якщо ж електронна адреса не знайдена в базі, викликаються методи register(email) та createUserInDatabase(email), які реєструють нового користувача і додають його до бази даних.

Даний фрагмент коду рис. 13.4, відповідає за реєстрацію нового користувача, топто сворення ногово запису з електроною адресою та типу користувача (водій, пасажир). Кожен новий користувач отримує тип пасажир.

Фрагмент коду рис. 13.5, відповідає за вікриття наступного вікна додатку в залежності від типу облікогово запису. Якщо тип акаунту відповідає водієві на наступне вікно передаються дані про тип його авто intent.putExtra("typeCar", userType).

3.3 Головний екран пасажира

Після успішної авторизації користувача, відкривається головне вікно рис.14 на якому розміщено зручну інтерактивну карту з можливістю маштабування. На даному екрані користувач має змогу обрати тип авто, вказати адресо початку та закінчення маршуту та замовити таксі. Після того як користувач зробить замовлення вигляд головного екрау зміниться рис. 15, редагувати адресу маршуту на даному етапі неможливо, можна лише відмінити поїздку. Коли водій прийме замовлення вигляд головного екрану знову зміниться рис. 16, на карті відображатиметься місцезнаходження водія.

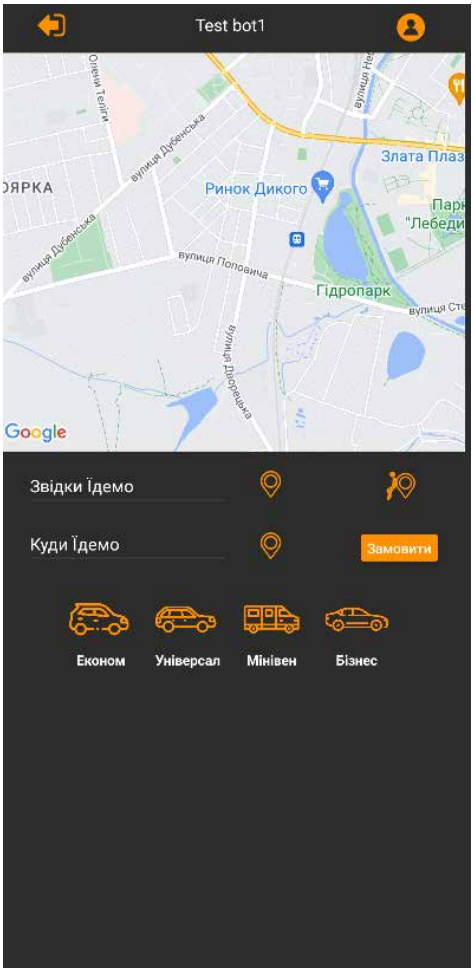


Рис. 14 Головне вікно користувача

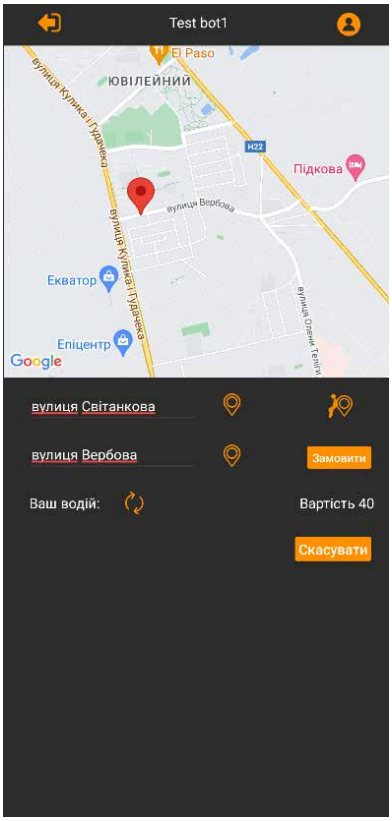


Рис. 15 Головне вікно користувача пошук водія

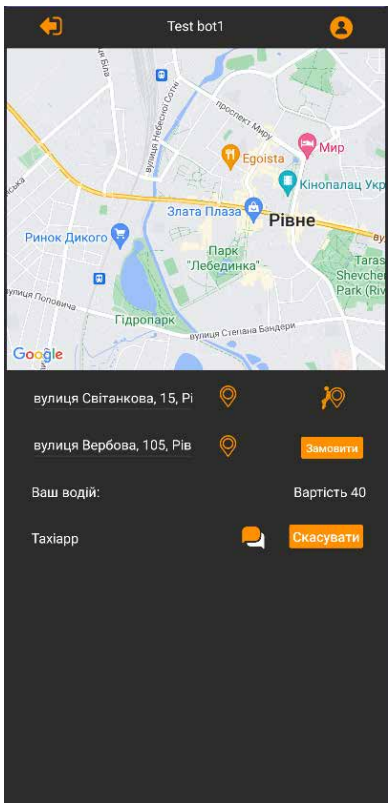


Рис. 16 Головне вікно користувача інформація про поїздку

Для роботи даного вікна використовуються методи код яких представлений в додатку Б рис. 17.1-17.16.

Перший метод рис. 17.1 відповідає за перевірку наявності активних поїздок користувача в системі таксі.

Метод checkOnTrip(String email) виконує запит до бази даних Firebase, шукаючи записи про активні поїздки, пов’язані з електронною адресою користувача. Якщо знайдено відповідний запис, перевіряється параметр type, який вказує на статус поїздки. Якщо type дорівнює 1 або 3, це означає, що поїздка активна, і користувач вже перебуває в процесі замовлення. У цьому випадку отримується ідентифікатор поїздки (idTrip), змінна aBoolean встановлюється в true, і викликаються методи getInfo(idTrip), startListenerFirebase(idTrip) і getNewMessage(), які відповідають за оновлення даних про поїздку, та перевіряють наявність нових повідомлень в додатку. Якщо type дорівнює 0, це означає, що замовлення очікує призначення водія. У такому випадку також отримується idTrip, встановлюється aBoolean=true, викликається getInfo(idTrip), та додатково запускається метод getDriver(), який відповідає за графічне відображення пошуку водія.

Наступний фрагмент коду рис. 17.2 надсилає запит до БД для отримання координат маршуту та певних даних про поїздку.

Спочатку метод створює запит до бази даних, фільтруючи поїздки за унікальним ідентифікатором idtrip. Якщо запис існує, дані поїздки витягуються та аналізуються.

Одним із перших параметрів, що отримуються, є coust – вартість поїздки, яка одразу відображається у coustText, доповнюючи інтерфейс інформацією про ціну.

Далі метод перевіряє наявність координат початкової (latlngStart) і кінцевої (latlngEnd) точок маршруту. Якщо координати збережені у базі, вони зчитуються як широта (latitude) і довгота (longitude), після чого створюються відповідні об’єкти LatLng (latLngStart та latLngEnd).

Насамкінець викликається метод setText(), який оновлює інтерфейс додатка, та обробляє отримані дані.

Фрагмент коду рис 17.3 використовується для перетворення отриманих координат в адресу, та для оновлення інтерфейсу.

Першим кроком створюється об'єкт Geocoder, який використовується для перетворення географічних координат у адреси. Код використовує uk-UA, що гарантує правильне форматування українських назв вулиць.

Далі у блоці try за допомогою geocoder.getFromLocation() метод отримує список можливих адрес за координатами початкової точки latLngStart. Якщо список не порожній, витягується перша адреса, і її вулична назва обрізається до останньої коми, щоб видалити зайву інформацію (назву міста та області). Отриманий результат записується у поле editTextStart.

Подібна операція виконується для кінцевої точки latLngEnd, де отримана адреса записується у editTextEnd.

Після отримання адрес змінюється доступність кнопок: coustText.setVisibility(View.VISIBLE); – відображає вартість поїздки. buttonOrder.setEnabled(false); – деактивує кнопку замовлення, imageView1, imageView2, imageView3 – вимикаються, щоб уникнути змін у параметрах поїздки після підтвердження. buttonCancel.setEnabled(true); – активується кнопка скасування, надаючи користувачеві можливість відмовитися від поїздки. LinearLayoutFull.setVisibility(View.GONE); – ховає макет, що містить перелік типів авто. buttonCancel.setVisibility(View.VISIBLE); – відображає кнопку скасування, роблячи її доступною.

Метод setText() не лише оновлює поля з адресами, але й налаштовує інтерфейс додатка відповідно до поточного статусу поїздки, забезпечуючи логічний і зручний досвід для користувача.

Показаний на рис. 17.4 метод відіграє важливу роль у моніторингу змін стану поїздки в реальному часі, використовуючи Firebase Realtime Database.

Даний механізм забезпечує оновлення інтерфейсу користувача відповідно до подій, що відбуваються під час поїздки. Основна логіка методу полягає у створенні ValueEventListener(постійного слухача БД), який відстежує зміни в trip-записах для конкретної поїздки, і його додаванні до відповідного Query. Спочатку визначається DatabaseReference, що вказує на вузол trip, та формується Query, який шукає запис з idtrip, що дорівнює id.

Перевіряється, чи активна сторінка клієнта (isCustomerActivity) та чи виконано інші умови (aBoolean).

Опрацьовуються можливі зміни у статусі поїздки. Якщо type = 1, це означає, що поїздку прийнято водієм. Тому встановлюється isDriver=true, викликається метод getDataOfDriver(id), що отримує інформацію про водія. Якщо type = 3, це свідчить, що водій скасував поїздку. У такому разі isDriver=false, відображається повідомлення "Водій відмінив поїздку", видаляється чат, викликається getDriver() метод для відображення на екрані пошуку нового водія, а якщо на карті є маркер попереднього водія (driverMarker), він видаляється. Якщо type == 4, це означає, що поїздка завершена. У такому випадку cleanAll() очищає дані, а aBoolean=false припиняє подальшу обробку змін.

Наведений метод забезпечує інтерактивне оновлення інтерфейсу клієнта у відповідь на зміни в поїздці, автоматично реагуючи на прийняття, скасування або завершення замовлення.

Цей фрагмент коду рис. 17.5 створений для відображення на екрані користувача даних про водія, що прийняв замовлення.

Метод getDataOfDriver(int id) відіграє ключову роль у процесі відображення інформації про водія, який прийняв поїздку. Його основне завдання — отримати ім’я водія з бази даних Firebase та відобразити його в інтерфейсі користувача.

При виклику цього методу формується запит до Firebase Realtime Database, що шукає запис у розділі trip за конкретним ідентифікатором idtrip. Якщо такий запис існує, зчитується ім’я водія та встановлюється у відповідне текстове поле на екрані. Це дозволяє пасажиру побачити, хто саме буде виконувати поїздку, і дає відчуття впевненості у процесі.

Окрім відображення імені водія, метод також оновлює інші елементи інтерфейсу. Індикатор завантаження зникає, повідомляючи користувача, що дані успішно завантажені. Текстове поле отримує підпис "Ваш водій: ", а кнопка чату стає активною, дозволяючи пасажиру зв’язатися з водієм у разі потреби. Водночас відображається загальна інформація про вартість поїздки, що є важливим елементом прозорості сервісу.

Після завершення цих дій метод викликає getDriverLocation(id), який займається відображенням місцезнаходження водія в реальному часі. Це дозволяє пасажиру стежити за пересуванням водія, очікуючи його прибуття, що значно підвищує рівень комфорту та зручності використання застосунку.

Наведений фрагмент коду рис 17.6 слугує для скасування поїздки користувачем.

Метод onExist() відповідає за ініціацію процесу скасування поїздки, пропонуючи користувачу підтвердити своє рішення. При виклику цього методу створюється спливаюче вікно AlertDialog, у якому відображається запит: "Скасувати поїздку?" разом із поясненням, що дія є незворотною. Користувач

має два варіанти дій: натиснувши "Так", він підтверджує скасування, що призводить до виклику методу onCancel(idTrip), який змінює статус поїздки в базі даних на значення 2, що означає її скасування. Далі викликається метод cleanAll(), який очищає відповідні змінні та елементи інтерфейсу, забезпечуючи коректне завершення операції. Якщо ж користувач натискає "Ні", вікно просто закривається без будь-яких змін у системі. Це рішення дозволяє уникнути випадкового скасування поїздки, залишаючи користувачу можливість передумати.

Метод onCancel(int id) відіграє технічну роль у зміні статусу поїздки в базі даних. Використовуючи запит до Firebase Realtime Database, він знаходить запис із відповідним idtrip і змінює його поле type, що сигналізує системі про завершення поїздки саме пасажиром.

Завдяки цій реалізації користувач отримує інтуїтивний інструмент для керування своїми поїздками, а система коректно обробляє їх зміни, забезпечуючи надійність та передбачуваність поведінки додатка.

Даний фрагмент коду слугує для отримання координат початку та завершення маршуту.

Методи getStart() та getEnd() відіграють важливу роль у визначенні початкової та кінцевої точок маршруту користувача.

Вони використовують клас Geocoder, який дозволяє перетворювати введену адресу у географічні координати (широту та довготу). Коли користувач вводить назву вулиці в editTextStart або editTextEnd, система намагається знайти відповідні координати за допомогою методу getFromLocationName(). Якщо геокодер знаходить адресу, вона зберігається як об'єкт LatLng, який надалі використовується для відображення місць на карті.

Якщо введена адреса є некоректною або геокодер не може знайти відповідну локацію, користувач отримає повідомлення з порадою встановити

точку вручну на карті. Додатково передбачено перевірку інтернет-з'єднання, оскільки Geocoder потребує доступу до онлайн-сервісів.

Реалізація цих методів забезпечує зручний та інтуїтивно зрозумілий механізм вибору маршрутів для користувача, дозволяючи йому вказувати точки як вручну, так і автоматично через пошук адрес.

Наступний метод рис. 17.8 дозволяє перевірити корекність маршуту між двома точками, та розрахувати вартість поїздки. Для цього використовується API Google Maps, зокрема, сервіс Directions API, який надає маршрути для автотранспорту.

Спочатку метод створює об'єкт GeoApiContext, що містить API-ключ для доступу до сервісів Google Maps. Далі формується запит через DirectionsApiRequest, де задаються початкова та кінцева точки маршруту у вигляді координат типу LatLng. Параметри запиту також включають режим подорожі (автомобільний транспорт) та одиниці вимірювання відстані (метричні). Коли запит надсилається через метод await(), отримані результати зберігаються в об'єкті DirectionsResult.

Якщо маршрути знайдені, система отримує перший маршрут і розраховує відстань між точками в кілометрах. Це значення використовують для обчислення вартості поїздки. Вартість залежить від відстані, типу автомобіля та базової плати. Результат виводиться на екран у вигляді тексту, що містить інформацію про вартість.

Додатково викликається метод createTrip(), який створює поїздку з переданими параметрами, включаючи відстань та тип автомобіля.

Якщо ж маршрути не знаходяться або відбувається помилка в процесі, користувач отримує відповідне повідомлення про помилку через Toast.

Даний метод забезпечує інтерактивний розрахунок вартості поїздки, враховуючи всі необхідні параметри, та змінює інтерфейс для виведення цієї інформації.

Метод показаний на рис. 17.9 займається створенням нової поїздки в БД Firebase та оновленням інтерфейсу користувача, щоб відобразити новий статус замовлення.

Процес починається із створення нового запису для поїздки в Firebase, використовуючи DatabaseReference для доступу до колекції "trip". Запит на отримання ідентифікатора для нової поїздки виконується через метод getIdTrip(), який використовує OnGetDataListener для обробки результатів.

Коли ідентифікатор отримано, відбувається оновлення змінних, зокрема створюється новий ідентифікатор для чату, записується початкова та кінцева точка маршруту (як координати), а також тип автомобіля.

Далі формується об'єкт tripData, який містить всі необхідні дані про поїздку: електронну пошту клієнта, тип поїздки, ім'я водія (яке ще не задано), GPS-координати водія (початково встановлені на (0,0)), вартість поїздки та тип автомобіля. Цей об'єкт передається до Firebase через tripsRef.updateChildren(), що оновлює запис у базі даних.

Після того, як інформація збережена, змінюється видимість елементів інтерфейсу. Останнім кроком викликається метод startListenerFirebase(), що слідкує за новим записом поїздки в Firebase, щоб обробляти подальші зміни.

Метод createTrip() не лише створює запис про поїздку в базі даних, але й оновлює інтерфейс користувача, щоб він бачив актуальний статус замовлення.

Наведений метод рис. 17.10 виконує очистку та відновлення інтерфейсу після завершення поїздки. Його головною метою є повернення елементів

інтерфейсу до початкового стану після того, як поїздка була завершена або скасована.

Початок методу супроводжується повідомленням для користувача через Toast, яке інформує його про завершення поїздки. Далі відбувається скидання значень різних змінних, зокрема координат стартової та кінцевої точок маршруту (latLngStart, latLngEnd, latLngGPS), а також налаштувань інтерфейсу.

Елементи інтерфейсу, такі як прогрес-бар, текстові поля для вартості та іншої інформації, кнопки, зображення і т.д., приховуються або вимикаються.

Кнопки для скасування та замовлення поїздки змінюють свій стан. Кнопка скасування стає невидимою та вимикається, а кнопка замовлення стає активною.

Інтерфейс для чат-іконки також очищується: зображення скидається до стандартного та елемент приховується. Крім того, відновлюються значення в текстових полях для введення місць поїздки (заміна підказок на "Звідки Їдемо" та "Куди Їдемо"), а також видаляється маркер водія на карті, якщо такий був.

Наприкінці методу відновлюються видимість основного елемента інтерфейсу та доступність кнопок вибору типу автомобіля (економ, універсал, мінівен, бізнес). Встановлюється значення змінної typeCar (тип автомобіля) на 0, що позначає скидання вибору, і перемикається прапорець aBoolean на false, що вказує на завершення операцій.

Загалом, метод cleanAll() ефективно відновлює стан додатку після завершення поїздки, очищуючи дані та забезпечуючи чистий інтерфейс для подальших дій користувача.

Наведений фрагмент коду рис. 17.11 відповідає за видалення чату з БД Firebase за допомогою ідентифікатора чату (id).

Процес починається з отримання посилання на конкретний чат у Firebase через DatabaseReference, вказуючи шлях до колекції "chat" і обраний запис чату за допомогою переданого ідентифікатора.

Далі метод використовує addListenerForSingleValueEvent() для одноразового отримання даних з Firebase. Якщо чат існує (перевіряється через dataSnapshot.exists()), викликається метод removeValue(), який видаляє запис чату з бази даних.

Якщо ж операція скасована з якихось причин (наприклад, через помилку з підключенням або іншими проблемами в мережі), це обробляється у методі onCancelled().

Загалом, метод deleteChat() реалізує просту і ефективну функцію для видалення чату з Firebase, з можливістю обробки результатів операції.

Наступний метод рис. 17.12 відповідає за отримання нового ідентифікатора для поїздки (trip) з бази даних Firebase. Він використовує механізм запитів до Firebase для того, щоб знайти останній запис у колекції "trip" і збільшити його ідентифікатор на одиницю.

Процес починається з створення посилання на колекцію "trip" у базі даних Firebase. Далі, за допомогою запиту limitToLast(1), обирається лише останній запис з цієї колекції. Це необхідно для того, щоб дізнатися поточний максимальний ідентифікатор і збільшити його на одиницю, таким чином створюючи новий унікальний ідентифікатор для наступної поїздки.

Метод використовує addListenerForSingleValueEvent() для одноразового отримання даних.

Якщо запис існує, то в циклі for кожен дочірній елемент перевіряється на наявність значення idtrip. Якщо таке значення є, воно зберігається в змінній

number, а потім до цього значення додається одиниця, що дає новий унікальний ідентифікатор для наступної поїздки.

Якщо записів немає або виникла інша помилка, то значення за замовчуванням — 1 — повертається як ідентифікатор. Результат роботи методу передається через виклик listener.onSuccess(id), що дозволяє передати новий ідентифікатор до того місця, де був викликаний метод getIdTrip().

Даний метод може ефективно генерувати унікальні ідентифікатори для нових поїздок на основі поточного стану бази даних, забезпечуючи послідовність і уникнення дублювання ідентифікаторів.

Даний метод рис. 17.13 відповідає за оновлення інтерфейсу в залежності від наявності нових повідомлень.

Спочатку створюється посилання на конкретний чат у Firebase за допомогою mDatabase.child("chat").child(String.valueOf(idTrip)), де idTrip — це ідентифікатор поточної поїздки. Це посилання дає доступ до колекції повідомлень для конкретної поїздки.

Далі додається слухач за допомогою addValueEventListener(), що дозволяє відслідковувати зміни у чаті в реальному часі. Кожного разу, коли з'являються нові дані в чаті, буде виконуватись метод onDataChange().

У методі onDataChange() відбувається очищення списку повідомлень chatDatesList.clear(), після чого кожен новий запис (повідомлення) з чату додається до цього списку.

Після додавання повідомлень у список відбувається перевірка, чи кількість повідомлень у списку перевищує певну кількість (numberChat, цей параметр оновлюється після відкриття користувачем вікна з чатом і дорівнює останній кількості повідомлень в ньону на момент відкриття). Якщо так,

змінюється іконка чату на нову (chaticon_new), що вказує на наявність нових повідомлень.

Якщо ж кількість повідомлень менша або рівна зазначеній кількості, іконка змінюється на стандартну (chaticon).

Метод ефективно здійснює відслідковування нових повідомлень у чаті, оновлюючи інтерфейс, якщо є нові повідомлення.

Даний фрагмент коду рис. 17.14 слугує для виходу з акаунту Google, та повертає користувача на початковий екран де він має змогу авторизуватися в іншому акаунті.

Даний рис. 17.15 , виконує важливу функцію автоматичного визначення точок маршруту на основі координат, вибраних користувачем на карті. Якщо змінна latLngOther містить значення, тобто користувач уже вибрав місце на карті, система присвоює ці координати змінній latLngStart або latLngEnd в залежності від натиснутої кнопки.

Це дозволяє автоматизувати процес введення точок початку та завершення маршуту та, позбавляє користувача необхідності вручну вказувати адресу.

Оскільки для отримання адреси система використовує онлайн-сервіси, можливі ситуації, коли інтернет-з'єднання нестабільне або сервери геокодування недоступні. У таких випадках може виникнути IOException, яку перехоплює блок catch, дозволяючи уникнути аварійного завершення програми.

Також у коді реалізований сценарій, коли latLngOther є null, тобто користувач ще не вибрав точку на карті. У такій ситуації система виводить підказку через Toast з повідомленням "Утримуйте на карті для створення точки", що допомагає користувачу зрозуміти, як саме додати місце.

Даний код рис. 17.16 відповідає за отримання поточного місця розташування користувача та автоматичне встановлення його як початкової точки маршруту. При натисканні на imageView3 система перевіряє, чи надано дозвіл на доступ до геолокації. Якщо дозвіл є, використовується FusedLocationProviderClient для отримання останнього відомого місцезнаходження. Якщо координати успішно отримані, створюється об'єкт LatLng, який містить широту та довготу.

Далі за допомогою Geocoder ці координати перетворюються в текстову адресу, що виводиться у editTextStart.

У випадку, якщо дозвіл на геолокацію ще не надано, програма запитує його у користувача через ActivityCompat.requestPermissions(). Завдяки цьому підходу система забезпечує зручний спосіб швидко встановити початкову точку маршруту без необхідності ручного введення адреси.

3.4 Головний екран водія.

Після успішної авторизації користувача з типом акаунту «Водій», він потрапляє на головний екран водія рис. 18. На даному екрані в режимі реального часу відображаються активні замовлення користувачів з інофрмацією про вартість замовлень та про їх відстань. Всі поїздки які відображаються конкретному водієві відповідають його типу авто.

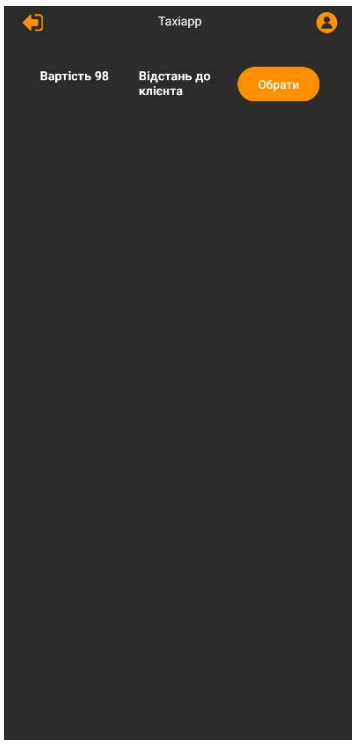


Рис. 18 Головний екран водія

Розгланемо основні методи які забеспечують коректну роботу даного вікна Додаток В рис. 19.1-19.5.

Продемонстрований метод рис. 19.1 відіграє важливу роль у забезпеченні роботи додатка для водія, перевіряючи наявність дозволу на доступ до геолокації.

Він використовує ContextCompat.checkSelfPermission(), щоб визначити, чи надано дозвіл ACCESS_FINE_LOCATION.

Якщо дозвіл уже є, викликаються два методи: getDriverVocation() - отримує поточне місцезнаходження водія, та startListeningToFirebaseChanges() - розпочинає відстеження змін у базі даних Firebase. Це забезпечує актуальне оновлення статусу водія та можливість отримання нових замовлень у режимі реального часу.

Якщо ж дозвіл не надано, метод ActivityCompat.requestPermissions() запитує його у користувача, відправляючи відповідний запит із кодом LOCATION_PERMISSION_REQUEST_CODE.

Наведений метод гарантує, що додаток працюватиме коректно лише за наявності необхідних прав, а також дотримується принципів безпеки Android, які вимагають явного підтвердження доступу до чутливих даних.

Даний метод рис. 19.2 відповідає за отримання поточного місцезнаходження водія та його збереження у вигляді координат LatLng.

Викликається метод getLastLocation(), який повертає об'єкт Location. Після того як дані успішно отримані, витягуються широта та довгота, далі створюється об'єкт LatLng, що зберігає координати водія.

Метод рис. 19.3 використовується для розрахунку відстані між водієм та клієнтом у кілометрах, використовуючи Google Maps Directions API.

Він отримує координати клієнта у вигляді LatLng latLngCustomer і повертає відстань у вигляді рядка.

Спочатку створюється об’єкт GeoApiContext, який містить API-ключ для доступу до сервісу Google. Далі формується запит DirectionsApiRequest, де вказується початкова точка (місцезнаходження водія), кінцева точка (місце розташування клієнта) та параметри маршруту. Отриманий результат зберігається у DirectionsResult, після чого перевіряється, чи існують маршрути.

Якщо хоча б один маршрут знайдено, береться його перший сегмент (DirectionsLeg), і визначається загальна дистанція у метрах, яка конвертується в кілометри. Якщо під час виконання запиту виникає помилка, її перехоплює catch (Exception e), що запобігає аварійному завершенню роботи програми.

Даний метод повертає знайдену відстань у вигляді текстового значення, яке далі виводиться на голоаному екрані водія.

Наведений метод рис.19.4 використовується для відстеження змін у базі даних Firebase у реальному часі, зокрема в розділі "trip.

Спочатку створюється посилання на DatabaseReference, яке вказує на колекцію "trip", а потім додається ValueEventListener, що викликається щоразу при зміні даних.

У методі onDataChange() список taxiDataModelList очищується, і відбувається перевірка кожного збереженого замовлення. Перевіряється, чи збігається тип автомобіля (typecar) з поточним типом (typeCar). Якщо це так, замовлення додається у список, але тільки у випадку, якщо його статус (type) дорівнює 0 або 3.

Для кожного замовлення отримуються такі дані: вартість (coust), ідентифікатор замовлення (idtrip), координати клієнта (latlngStart).

Координати клієнта використовуються для визначення відстані між водієм та клієнтом через getTripDistance().

Після цього формується новий об'єкт, мого власного класу, TaxiDataModel, який містить вартість, відстань та кнопку вибору поїздки, і додається до taxiDataModelList.

Наприкінці оновлюється adapter, що оновлює відображення списку доступних замовлень у додатку.

Цей клас рис. 19.5 використовується для збереження даних про поїздку що відображаються у списку доступних замовлень.

Об'єкти класу TaxiDataModel відіграють ключову роль у системі керування замовленнями.

Цей клас є моделлю даних, яка містить інформацію про кожну окрему поїздку та використовується для формування списку доступних замовлень для водіїв.

При створенні нового об'єкта цього класу він отримує значення для чотирьох основних полів: text1 – текстове поле, що зберігає інформацію про вартість поїздки, text2 – містить дані про відстань до клієнта. buttonLabel –кнопка з написом "Обрати". tripId – унікальний ідентифікатор поїздки, який дозволяє коректно опрацьовувати замовлення в базі даних.

Коли нове замовлення з’являється в системі, створюється об’єкт TaxiDataModel, який наповнюється відповідною інформацією. Потім цей об'єкт додається у список доступних поїздок. Список оновлюється в реальному часі завдяки обробці змін у базі даних Firebase, і всі активні водії отримують актуальну інформацію про нові замовлення.

Наведений клас рис. 19.6 дозволяє зручно відображати список доступних замовлень у вигляді інтерактивних елементів.

Цей клас є адаптером для RecyclerView, він відповідає за зв’язок між списком даних та графічним інтерфейсом користувача.

Основним елементом TaxiAdapter є список об'єктів TaxiDataModel, переданий через конструктор. Цей список містить усі доступні замовлення, кожне з яких має дані про вартість, відстань до клієнта та унікальний ідентифікатор поїздки.

В межах класу оголошено внутрішній клас ViewHolder, який містить три основні елементи інтерфейсу: textView1 – для відображення вартості поїздки. textView2 – для показу відстані до клієнта. button – кнопка, яка дозволяє водієві вибрати певне замовлення.

Створення елемента списку Метод onCreateViewHolder завантажує макет list_item.xml і створює новий об’єкт ViewHolder. Це гарантує, що всі елементи списку матимуть однакову структуру.

Метод onBindViewHolder отримує об'єкт TaxiDataModel для конкретної позиції у списку та передає відповідні значення у textView1, textView2 і button.

Коли водій натискає кнопку вибору, спрацьовує setOnClickListener, який створює Intent для переходу до нового екрану (driver_map). Водночас у Intent передаються дані про вибране замовлення (data), що дозволяє новій активності отримати всю необхідну інформацію про поїздку.

Метод getItemCount повертає загальну кількість замовлень, що гарантує правильне відображення всіх доступних варіантів.

Клас TaxiAdapter є важливим мостом між даними про поїздки та інтерфейсом програми. Завдяки цьому адаптеру водії можуть бачити оновлений список замовлень, переглядати ключову інформацію та легко обирати відповідну поїздку.

3.5 Екран з поїздкою для водія

Після вибору поїздки водієм він потрапляє на наступне вікно рис. 20. На даному екрані знаходиться карта на якій відображається поточне місцезнаходження водія та обраний маршут. Водій може переглянути інформацію про поїздку, її вартість та адреси початку і закінчення маршуту. На даному вікні є дві кнопки одна для відкриття чату інша, для завершення поїздки.

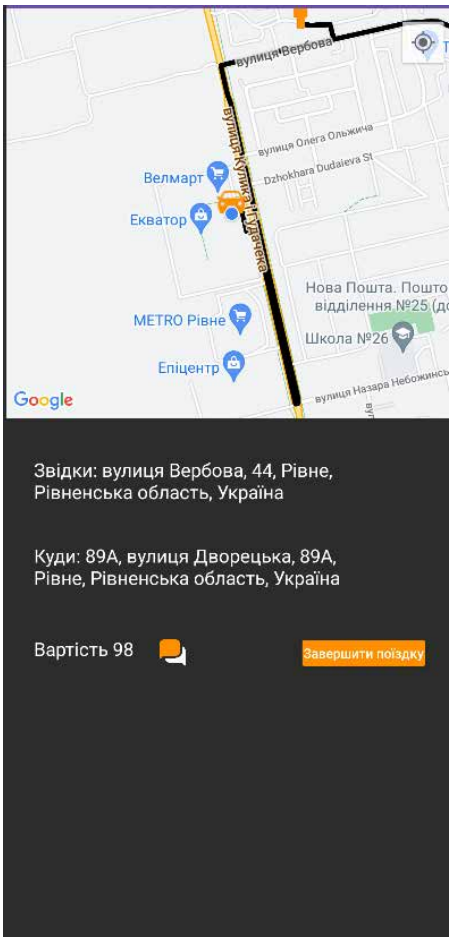


Рис. 20 Вікно з поїздкою

Для роботи даного вікна використовується наступний Додаок В код рис. 21.1-21.11

Даний фрагмент коду рис. 21.1 відповідає за отримання об’єкту класу TaxiDataModel з попереднього вікна. Даний об’єкт містить деяку інформацію

про поїзду а саме: варість, відстань до клієнта, та найголовніше ідентифікатор поїздки завдяки якому можна отримати всі інші неохідні дані про поїздку з БД.

Даний метод рис. 21.2 забезпечує отримання координат поїздки з БД. Його головне завдання — знайти запис у Firebase Realtime Database, який відповідає переданому унікальному ідентифікатору поїздки, та витягнути з нього ключові координати маршруту клієнта.

При виклику методу створюється запит до бази даних, який шукає запис у вузлі trip, де значення idtrip збігається з переданим id. Якщо запис знайдено, починається його обробка.

Метод по черзі аналізує вкладені дані, перевіряючи наявність координат початкової та кінцевої точок маршруту. Якщо вони присутні, то зчитуються значення широти та довготи для кожної з них.

Це дозволяє встановити точку, де клієнт розпочав поїздку (latlngCustomerStart), та місце призначення (latlngCustomerEnd). Отримані координати можна використовуються для візуалізації маршруту на карті. Завершальним етапом є виклик методу setText().

Наведений на рис. 21.3 метод забезпечує відображення точних адрес початкової та кінцевої точок маршруту клієнта.

Основна логіка цього методу полягає у використанні класу Geocoder, який дозволяє перетворювати координати (широту та довготу) у текстові адреси. Для цього створюється об’єкт Geocoder із зазначенням локалізації української мови, що гарантує отримання адрес у відповідному форматі.

Спочатку відбувається отримання адреси для точки початку маршруту (latlngCustomerStart). Метод getFromLocation() виконує зворотне геокодування, перетворюючи координати на список можливих адрес. Якщо отримано хоча б

одну адресу, з неї вилучається назва вулиці, яку потім форматують для коректного відображення.

Після обробки початкової точки аналогічна процедура виконується для кінцевого пункту призначення (latlngCustomerEnd). Отримані адреси встановлюються у відповідні TextView.

setText() відіграє важливу роль у покращенні взаємодії водія із застосунком, забезпечуючи зрозуміле та точне відображення маршрутних даних.

Основне призначення методу рис. 21.4 – встановити водія, який бере замовлення, і змінити статус поїздки.

Спочатку метод отримує доступ до гілки trip в Firebase, де зберігається інформація про всі активні замовлення. Використовуючи Query, він шукає конкретний запис, у якому значення idtrip збігається з переданим id. Далі, якщо запит успішно знаходить відповідний запис (snapshot.exists()), метод проходить по всіх знайдених DataSnapshot. У кожному такому об’єкті отримується посилання (getRef()) на потрібний запис, після чого оновлюються два важливих параметри. Поле namedriver отримує значення driverEmail, тобто адресу електронної пошти водія, який прийняв замовлення. Це дозволяє системі зафіксувати, хто саме буде виконувати поїздку. Поле type змінюється на 1.

Цей метод виконується лише один раз для конкретного замовлення (addListenerForSingleValueEvent), що запобігає повторному виклику оновлення при зміні бази даних.

Метод рис.21.5 створений для оновлення геолокації водія у БД Firebase. Це особливо важливо для покращення комунікації між водієм і клієнтом.

При виклику методу спочатку створюється посилання на гілку trip у Firebase, де зберігаються всі активні замовлення. За допомогою Query система знаходить запис, у якому значення idtrip відповідає переданому id.

Якщо запис знайдено (snapshot.exists()), виконується оновлення поля gpsdriver, яке зберігає поточне місцезнаходження водія. Значення для цього поля береться з змінної latLngDriverGPS, яка містить координати у форматі LatLng.

Використання методу addListenerForSingleValueEvent дозволяє виконати це оновлення лише один раз, що мінімізує навантаження на базу даних і запобігає повторному запису тих самих координат. Таким чином, метод setDriverLocation(int id) забезпечує ефективну інтеграцію геолокаційних даних у систему, що сприяє покращенню взаємодії між учасниками поїздки та підвищує точність роботи додатка у реальних умовах.

Даний метод рис. 21.6 створений для оновлення координат водія, та для передачі нових даних до БД Firebase.

При виклику методу спочатку створюється об'єкт LocationRequest, який налаштовує параметри отримання геолокації. Пріоритет встановлюється як PRIORITY_HIGH_ACCURACY, що означає використання GPS для максимально точної локації.

Інтервал оновлення задається в 5000 мілісекунд (5 секунд), що забезпечує часте отримання даних без зайвого навантаження на пристрій. Для обробки нових координат створюється LocationCallback, який виконується кожного разу при отриманні нового місцезнаходження. Спершу перевіряється, чи активна поточна активність (isActivity). Далі метод отримує останні координати через locationResult.getLastLocation(). Якщо нова локація присутня, вона зберігається у змінній mLastLocation, після чого оновлюється змінна latLngDriverGPS, що

містить LatLng-координати водія. На основі нових координат викликається метод createRoute(), який будує оновлений маршрут до місця призначення.

Також оновлюється місцезнаходження водія у Firebase шляхом виклику setDriverLocation(tripID).

Якщо на карті вже існує маркер mUserMarker, він видаляється, щоб оновити позицію без накопичення зайвих маркерів. Далі створюється новий маркер (MarkerOptions) із координатами водія, використовується спеціальна іконка автомобіля (caricon), а також карта масштабується на цю точку (mMap.moveCamera(CameraUpdateFactory.newLatLngZoom(...))).

Щоб уникнути проблем із дозволами, метод перед його виконанням перевіряє наявність дозволу ACCESS_FINE_LOCATION. Якщо дозволу немає, система запитує його у користувача за допомогою requestPermissions().

Метод рис. 21.7 виконує важливу задачу: він бере географічні координати поточного місця водія, місця посадки клієнта та кінцевої точки призначення, після чого взаємодіє з Google Maps Directions API, щоб отримати найкоротший і найзручніший шлях між цими точками.

Процес побудови маршруту розпочинається з перевірки даних. Якщо координати водія (latLngDriverGPS), клієнта (latlngCustomerStart) або точки призначення (latlngCustomerEnd) відсутні, то маршрут не буде створено.

Коли всі необхідні координати є в наявності, створюється спеціальний об'єкт GeoApiContext, що містить API-ключ, який дозволяє відправляти запити для побудови маршруту.

Далі відбувається безпосередньо виклик Google Directions API. Перший запит визначає маршрут від місцезнаходження водія до клієнта, а другий — від клієнта до місця призначення. Обидва запити виконуються у режимі автомобільного руху (TravelMode.DRIVING) та використовують метричну

систему вимірювання відстаней. Як тільки маршрути побудовані, вони передаються у функцію displayRouteOnMap(), яка займається їх відображенням на карті.

Завдяки даному методу водій отримує візуально зрозумілий маршрут.

Цей метод рис. 21.8 відповідає за графічне представлення маршруту водія на карті Google Maps.

Він приймає DirectionsResult result, який містить дані про маршрут, та об'єкт GoogleMap map, на якому відображається шлях. При виклику методу спочатку перевіряється, чи result не є null. Якщо маршрут отримано успішно, створюється об'єкт PolylineOptions, який відповідає за візуальне зображення маршруту на карті.

Далі отримується EncodedPolyline з overviewPolyline, який містить закодовану лінію маршруту. Вона декодується у список точок (decodePath()), і кожна координата додається до PolylineOptions, що формує безперервний шлях на карті.

Щоб маршрут був чітко видимим, створена лінія (Polyline) набуває чорного кольору.

Окрім побудови маршруту, метод також додає на карту маркери для позначення початкової (latlngCustomerStart) та кінцевої (latlngCustomerEnd) точок подорожі. Для цього використовується MarkerOptions, а іконки маркерів створюються через метод resizeBitmap(), що дозволяє адаптувати зображення до потрібного розміру. Таким чином, displayRouteOnMap() є важливим елементом візуалізації поїздки, який допомагає водію чітко бачити шлях на карті, а також орієнтуватися в точках посадки та висадки пасажира.

Наведений на рис. 21.9 метод відповідає за налаштування слухача для змін у даних про поїздки в БД. Спочатку створюється запит до Firebase, щоб

отримати поїздки з конкретним значенням поля idtrip, яке дорівнює переданому параметру id. Для цього використовується метод orderByChild("idtrip").equalTo(id). Далі створюється об'єкт ValueEventListener, який реагує на зміни у даних.

У методі onDataChange() перевіряється, чи активна активність (параметр isActivity). Якщо так, то для кожного елемента у результатах запиту перевіряється значення поля type. Якщо значення type дорівнює 2, викликається метод goDriverMain(), що відправляє водія до вікна із замовленями та виводить повідомлення про скасування замовлення клі’єнтом. Якщо значення type дорівнює 4, викликається метод goDriverMainF(), який повертає водія до вікна замовлень та виводить повідомлення про завершення поїздки.

Метод рис. 21.10 відповідає за завершення поїздки, оновлюючи значення поля type для конкретної поїздки в базі даних Firebase.

Спочатку створюється запит до бази даних Firebase, щоб знайти поїздки, де значення поля idtrip дорівнює переданому параметру id.

Далі викликається метод addListenerForSingleValueEvent(), щоб виконати запит лише один раз. У методі onDataChange() перевіряється, чи існують результати запиту (за допомогою snapshot.exists()). Якщо поїздка знайдена, виконується оновлення значення поля type на 4.

finishTrip() дозволяє оновити статус поїздки, позначивши її як завершену, змінюючи значення поля type.

Даний метод рис. 21.11 відповідає за оновлення інтерфейсу в залежності від наявності нових повідомлень.

Спочатку створюється посилання на конкретний чат у Firebase за допомогою mDatabase.child("chat").child(String.valueOf(tripID)). Це посилання дає доступ до колекції повідомлень для конкретної поїздки.

Далі додається слухач за допомогою addValueEventListener(), що дозволяє відслідковувати зміни у чаті в реальному часі. Кожного разу, коли з'являються нові дані в чаті, буде виконуватись метод onDataChange().

У методі onDataChange() відбувається очищення списку повідомлень chatDatesList.clear(), після чого кожен новий запис (повідомлення) з чату додається до цього списку.

Після додавання повідомлень у список відбувається перевірка, чи кількість повідомлень у списку перевищує певну кількість (numberChat, цей параметр оновлюється після відкриття водєм вікна з чатом і дорівнює останій кількості повідомлень на момент відкриття). Якщо так, змінюється іконка чату на нову (chaticon_new), що вказує на наявність нових повідомлень.

Якщо ж кількість повідомлень менша або рівна зазначеній кількості, іконка змінюється на стандартну (chaticon).

Метод ефективно здійснює відслідковування нових повідомлень у чаті, оновлюючи інтерфейс, якщо є нові повідомлення.

Дані ментоди рис. 21.12 створені для коректної обробки відмови від поїздки водієм.

Метод onExist() відповідає за ініціацію процесу скасування поїздки, пропонуючи водію підтвердити своє рішення.

При виклику цього методу створюється спливаюче вікно AlertDialog, у якому відображається запит: "Відмовитись від поїздки?". Водій має два варіанти дій: натиснувши "Так", він підтверджує скасування, що призводить до виклику методу onCancel().

Метод onCancel() виконує відміну поїздки шляхом оновлення даних у базі даних Firebase.

За допомогою запиту до Firebase шукається поїздка з ідентифікатором id. Якщо поїздка знайдена, очищується ім'я водія (поля namedriver), оновлюється тип поїздки на 3 (для позначення скасованої поїздки), а також встановлюється значення GPS-координат водія на (0.0, 0.0).

Після скасування поїздки, змінна isActivity встановлюється в false, і користувача перенаправляється на екран вибору активних замовлень за допомогою Intent.

Методи onExist() та onCancel() забезпечують функціонал відмови від поїздки та очищення даних про водія.

3.6 Вбудований чат

В мобільному застосунку для таксі важливою проблемою є забезпечення зручного та швидкого зв’язку між пасажиром та водієм. Вбудований чат вирішує цю проблему, дозволяючи користувачам миттєво обмінюватися повідомленнями без необхідності телефонних дзвінків.

Основна мета чату — забезпечити оперативну комунікацію, що допомагає уточнювати деталі поїздки, місце зустрічі або зміни маршруту. Інтеграція з базою даних дозволяє зберігати історію переписки, що підвищує зручність використання сервісу.

Після того як певну поїздку вибере водій, користувачі зможуть відкрити вікно чату рис. 22. Щоб детальніше ознайомитись з роботою даного вікна додатку розглянемо код що продемонтрований в додатку Г рис. 23.1-23.6

Рис. 22 Вікно чату

Даний метод рис. 23.1 забезпечує можливість відправлення повідомлень та їх збереження у БД. Цей механізм дозволяє користувачам швидко передавати текстові повідомлення, які потім відображаються в інтерфейсі додатку.

Робота методу починається зі створення посилання на відповідну гілку бази даних Firebase. Використовуючи chatId, додаток ідентифікує конкретний чат, а виклик push() створює новий унікальний запис, що дозволяє зберігати кожне нове повідомлення окремо, не перезаписуючи попередні.

Далі метод формує структуру повідомлення, використовуючи HashMap<String, Object>, де буде збережений текст повідомлення та його тип.

Перед тим як додати повідомлення в базу даних, проводиться важливий етап валідації введеного тексту. Використовуючи регулярний вираз

matches(".*\\w.*"), метод перевіряє, чи містить повідомлення хоча б один літерний або числовий символ. Це дозволяє уникнути випадкових порожніх або некоректних повідомлень, які можуть з’явитися внаслідок помилки введення.

Якщо повідомлення проходить перевірку, воно додається в chatData під ключем "Text", а також отримує додатковий атрибут "type", що дорівнює 0. Цей параметр вказує, що повідомлення надійшло від клієнта, що в подальшому дозволяє правильно відображати його в інтерфейсі.

Після цього метод викликає updateChildren(chatData), передаючи сформовані дані в Firebase. Далі метод setTxt() очищує поля введення chatTextCastomer.setText(""), що дозволяє користувачеві швидко ввести наступне повідомлення без необхідності вручну видаляти попередній текст.

Наведений метод рис. 23.2 відповідає за динамічне оновлення списку повідомлень у чаті, підключаючись до БД та відстежуючи зміни в чаті між клієнтом і водієм.

Він забезпечує автоматичне відображення нових повідомлень. При виклику методу створюється посилання на відповідну гілку бази даних chat, де chatId використовується для ідентифікації конкретного чату.

Далі перевіряється булеве значення bool, що зупиняти надсилання запитів до БД коли вікно не активне, це в свою чергу, дозволяє зменшити навантаження на БД.

Основна робота відбувається в методі onDataChange(), який викликається кожного разу, коли в чаті з’являються нові дані або відбувається їхнє оновлення. На початку очищується список chatDatesList, щоб запобігти дублюванню повідомлень. Потім іде обхід всіх записів у dataSnapshot, де кожне повідомлення обробляється окремо. Для кожного запису перевіряється його існування (snapshot.exists()). Далі зчитується поле "type" — це числовий ідентифікатор, що вказує, ким було надіслано повідомлення: Якщо type == 0, то повідомлення належить клієнту. Його текст отримується з поля "Text" і

створюється об'єкт кастомного класу ChatData, в якому текст клієнта додається до списку повідомлень (chatDatesList). Якщо type == 1, аналогічно зчитується текст, але тепер уже для повідомлення, надісланого водієм.

Після додавання всіх повідомлень у chatDatesList перевіряється його розмір. Якщо в ньому є елементи, виконується автоматична прокрутка recyclerView до останнього повідомлення (scrollToPosition(chatDatesList.size() - 1)), що дозволяє користувачам одразу бачити найновіші повідомлення.

Клас ChatData рис. 23.3 відіграє ключову роль у моделюванні повідомлень, що обмінюються між клієнтом і водієм у чаті. Основними атрибутами класу є: text1 – використовується для збереження тексту повідомлення одного з учасників чату. text2 – зберігає текст іншого учасника. Ця структура дозволяє ефективно відображати повідомлення в залежності від їхнього типу (тип повідомлення визначає його відправника клі’єнта чи водія).

Конструктор ChatData(String text1, String text2, int type) ініціалізує ці поля при створенні нового об'єкта. Це дозволяє програмі коректно зберігати й обробляти повідомлення в залежності від їхнього походження.

Для корекного графічного відображення повідомлень використовується клас ChatAdapter рис. 23.4

Даний клас є невід'ємною частиною функціоналу чату, оскільки він відповідає за виведення повідомлень у RecyclerView. Цей клас працює з колекцією об'єктів класу ChatData, кожен з яких містить текст повідомлення та інформацію про його тип (від кого воно надійшло).

Конструктор ChatAdapter(List<ChatData> dataList) приймає список об'єктів ChatData і зберігає його у внутрішній змінній dataList. Це дозволяє адаптеру отримувати доступ до всіх повідомлень і передавати їх у RecyclerView.

Внутрішній клас ViewHolder розширює RecyclerView.ViewHolder і містить дві TextView, які використовуються для відображення тексту повідомлень (textView1 та textView2).

Конструктор ViewHolder(View itemView) ініціалізує ці компоненти, знаходячи їх за id у макеті chat_items.xml.

Метод onCreateViewHolder(ViewGroup parent, int viewType) створює новий об'єкт ViewHolder, використовуючи LayoutInflater для завантаження XML-файлу chat_items.xml. Він повертає новий екземпляр ViewHolder, який буде використаний для відображення повідомлень.

Метод onBindViewHolder(ChatAdapter.ViewHolder holder, int position) виконує прив’язку даних до елементів ViewHolder. Він отримує об'єкт ChatData на певній позиції position у списку dataList та встановлює текстові значення для textView1 та textView2.

Метод getItemCount() повертає кількість елементів у dataList, визначаючи, скільки повідомлень буде відображено у RecyclerView.

Завдяки ChatAdapter, повідомлення виводяться в зручному для користувача форматі. Водій та клієнт можуть бачити власні та отримані повідомлення, що забезпечує ефективне спілкування. Використання RecyclerView дозволяє динамічно оновлювати список повідомлень без значних витрат ресурсів, що робить чат зручним і продуктивним.

4 РЕКОМЕНДАЦІЇ ЩОДО ВПРОВАДЖЕННЯ ТА ЕКСПЛУАТАЦІЇ СИСТЕМИ

Цей розділ містить рекомендації щодо впровадження, тестування та експлуатації розробленого мобільного застосунку служби таксі, а також опис вимог до апаратного й програмного забезпечення та склад інсталяційного пакету.

4.1 Тестування системи

Для забезпечення коректної роботи мобільного застосунку було проведено комплексне тестування, що охоплює як функціональні, так і нефункціональні характеристики.

Основними цілями тестування є перевірка працездатності основних функцій, зручності користування, стабільності та надійності додатку.

Функціональне тестування охоплювало перевірку наступних модулів:

- реєстрація та авторизація користувачів (водій/пасажир);
- замовлення поїздки з вибором пункту відправлення та призначення;
- обробка замовлення водієм;
- побудова маршруту за допомогою Google Maps API;
- система сповіщень;
- відображення поточного розташування на мапі;
- завершення поїздки та підрахунок вартості;
- оплата (тестова реалізація без справжніх платіжних сервісів).

Тестування проводилося на різних версіях Android (від 8.0 до 13.0), а також на пристроях із різними характеристиками. У результаті тестування були виявлені та усунені незначні помилки, пов'язані із відображенням інтерфейсу на екранах з малим розширенням та затримками при повільному інтернет-з’єднанні.

Методика тестування: застосовувалися як ручні методи перевірки (manual testing), так і елементи модульного тестування окремих класів Java з використанням фреймворку JUnit.

Результати тестування підтверджують стабільну роботу програми за умов дотримання мінімальних технічних вимог.

4.2 Вимоги до апаратного та програмного забезпечення

Для коректної роботи мобільного застосунку служби таксі необхідно дотримуватися наступних вимог:

Апаратне забезпечення (для клієнта та водія):

- смартфон або планшет з операційною системою Android версії 8.0 або вище;
- мінімум 2 ГБ оперативної пам’яті;
- мінімум 200 МБ вільного місця на внутрішньому накопичувачі;
- GPS-модуль;
- стабільне з’єднання з Інтернетом (Wi-Fi або мобільні дані 3G/4G).

Програмне забезпечення:

- встановлений Google Play Services;
- наявність дозволів до геолокації та інтернет-з’єднання;

4.3 Склад інсталяційного пакету

Інсталяційний пакет мобільного застосунку, розробленого для автоматизації роботи служби таксі, є комплексом файлів і документів, які забезпечують його повноцінне встановлення, налаштування та початкову експлуатацію.

Комплектність пакету має важливе значення не лише для технічної підтримки та обслуговування, а й для забезпечення простоти впровадження системи кінцевими користувачами.

Основним елементом інсталяційного пакету є інсталяційний файл з розширенням .apk, який містить скомпільований мобільний застосунок. Мій файл TaxiApp.apk може бути встановлений на пристрій вручну або завантажений через корпоративний сервер, внутрішній маркетплейс або з допомогою зовнішнього хмарного сервісу.

Окрім самого додатку, до інсталяційного пакету додається інструкція з встановлення. Вона включає покроковий опис процесу інсталяції на мобільний пристрій Android, перелік необхідних дозволів (наприклад, доступ до геолокації, мережі, повідомлень), а також рекомендації щодо безпечного встановлення з невідомих джерел, що є типовою вимогою для розробницьких версій додатків.

Ще одним важливим компонентом пакету є короткий посібник користувача. Цей документ, містить базову інформацію про функціональні можливості застосунку, включно з інтерфейсом, можливостями реєстрації, створенням та обробкою замовлень, користуванням мапою, отриманням сповіщень тощо. Посібник розроблений у зрозумілій формі, розрахований на водіїв, які можуть не мати глибокої технічної підготовки.

Даний інсталяційний пакет мобільного застосунку включає не лише технічну частину, але й усю необхідну документацію, що забезпечує прозоре, безпечне та ефективне впровадження системи в реальні умови експлуатації. Такий підхід дозволяє уникнути помилок при встановленні, зменшити витрати на навчання персоналу та гарантує зручність у підтримці додатку в майбутньому.

ВИСНОВКИ

У дипломній роботі я розробив мобільний додаток для служби таксі, який забезпечує ефективну взаємодію між водієм і пасажиром.

Для кращого розуміння потреб та вимог користувачів я проаналізував наявні на ринку служби таксі, та встановив необхідні вимоги до мого власного додаку.

Основною метою проєкту було створення мінімалістичного, інтуїтивно-зрозумілого інтерфейсу користувача. Розробка зручного та функціонального мобільного застосунку для замовлення та виконання поїздок, що враховує сучасні вимоги до сервісу таксі.

У процесі розробки було спроєктовано та реалізовано ключові компоненти додатка, зокрема систему реєстрації та авторизації користувачів, механізм пошуку та вибору водія, а також інтеграцію з картографічними сервісами.

Однією з найважливіших функцій стала побудова маршруту з урахуванням реального місцезнаходження водія та пасажира, що дозволяє оптимізувати процес перевезення.

Окремо варто відзначити, систему чату між користувачами, яка дає змогу пасажирам та водіям швидко обмінюватися повідомленнями без використання сторонніх месенджерів. Це підвищує рівень комунікації та сприяє кращій координації поїздок. Для збереження та обробки даних, я використав хмарну базу даних Firebase, що забезпечує швидкий обмін інформацією та збереження історії поїздок.

У розробці застосунку використовувалася мова програмування Java, а також API Google Maps для реалізації картографічних можливостей.

Тестування додатка підтвердило його працездатність та відповідність поставленим вимогам. Застосунок коректно обробляє запити на замовлення таксі, здійснює передачу даних між клієнтом та водієм та забезпечує безперебійний зв’язок.

Розроблений додаток має потенціал до подальшого вдосконалення. Можливими напрямами розвитку є інтеграція з платіжними системами для автоматизованої оплати поїздок, додавання рейтингової системи для оцінки водіїв та пасажирів, а також впровадження алгоритмів штучного інтелекту для розрахунку оптимальних маршрутів.

Створений додаток є ефективним рішенням для служби таксі, яке забезпечує швидке та зручне замовлення автомобілів. Реалізовані функції відповідають сучасним тенденціям у сфері мобільних технологій та можуть бути корисними як для водіїв, так і для клієнтів.

СПИСОК ВИКОРАСТАНИХ ДЖЕРЕЛ

1. Android Developer Documentation – [Електронний ресурс] – Режим доступу: https://developer.android.com/docs (Дата звернення 11.12.2024)
2. Firebase Documentation – [Електронний ресурс] – Режим доступу: https://firebase.google.com/docs (Дата звернення 11.12.2024)
3. "Android Studio User Guide", Google Inc. – [Електронний ресурс] – Режим доступу: https://developer.android.com/studio/intro (Дата звернення 01.01.2024)
4. "Firebase Realtime Database", Google Inc. – [Електронний ресурс] – Режим доступу: https://firebase.google.com/docs/database (Дата звернення 11.12.2024)
5. "Authentication", Firebase Documentation – [Електронний ресурс] – Режим доступу: https://firebase.google.com/docs/auth (Дата звернення 13.12.2024)
6. "Cloud Messaging", Firebase Documentation – [Електронний ресурс] – Режим доступу: https://firebase.google.com/docs/cloud-messaging (Дата звернення 13.12.2024)
7. "Cloud Firestore", Firebase Documentation – [Електронний ресурс] – Режим доступу: https://firebase.google.com/docs/firestore (Дата звернення 13.12.2024)
8. "Crashlytics", Firebase Documentation – [Електронний ресурс] – Режим доступу: https://firebase.google.com/docs/crashlytics (Дата звернення 13.12.2024)
9. "Analytics", Firebase Documentation – [Електронний ресурс] – Режим доступу: https://firebase.google.com/docs/analytics (Дата звернення 13.12.2024)
10. "App Distribution", Firebase Documentation – [Електронний ресурс] – Режим доступу: https://firebase.google.com/docs/app-distribution (Дата звернення 17.12.2024)

11. "I Taxi" – [Електронний ресурс] – Режим доступу: https://itaxi.com.ua/reytynhy/reyting-taksi/(Дата звернення 20.01.2025)

ДОДАТОК А

Код методів що забезпечують авторизацію користувачів.

```
@Override
protected void onStart() {
    GoogleSignInAccount account = GoogleSignIn.getLastSignedInAccount( context: this);
    if (account != null) {
        String email = account.getEmail();
        checkUserInDatabase(email);
    }
    else signIn();
    super.onStart();
}
```

Рис. 13.1 Метод для перевірки чи авторизовувся користувач раніше

```
    private void signIn() {  2 usages
        Intent signintent = gsc.getSignInIntent();
        startActivityForResult(signintent,  requestCode: 1000);
    }

    @Override
    protected void onActivityResult(int requestCode, int resultCode, Intent data) {
        super.onActivityResult(requestCode, resultCode, data);
        if (requestCode == 1000) {
            try {
                Task<GoogleSignInAccount> task = GoogleSignIn.getSignedInAccountFromIntent(data);
                GoogleSignInAccount account = task.getResult(ApiException.class);
                if (account != null) {
                    String userEmail = account.getEmail();
                    checkUserInDatabase(userEmail);

                } else {
                    signIn();
                }
            } catch (ApiException e) {
                e.printStackTrace();
                navigateToMainActivity();

            }
        } else {
            navigateToMainActivity();

        }
    }
```

Рис. 13.2 Процес аунтифікації користувачів

```
private void checkUserInDatabase(String email) {  2 usages
    mDatabase.child( pathString: "users").orderByChild( path: "email").equalTo(email).addListenerForSingleValueEvent(new ValueEventListener() {
        @Override
        public void onDataChange(@NonNull DataSnapshot dataSnapshot) {
            if (dataSnapshot.exists()) {
                DatabaseReference usersRef = mDatabase.child( pathString: "users");
                Query query = usersRef.orderByChild( path: "email").equalTo(email);
                query.addValueEventListener(new ValueEventListener() {
                    @Override
                    public void onDataChange(@NonNull DataSnapshot snapshot) {
                        if (snapshot.exists()) {
                            for (DataSnapshot childSnapshot : snapshot.getChildren()) {
                                Long type = childSnapshot.child( path: "type").getValue(Long.class);
                                if (type != null) {
                                    int typeValue = Math.toIntExact(type);
                                    navigateToSecond(typeValue);
                                }
                            }
                        }
                    }

                    @Override
                    public void onCancelled(@NonNull DatabaseError error) {

                    }
                });

            } else {
                register(email);
                createUserInDatabase(email);
```

```
                }
            }

            @Override
            public void onCancelled(@NonNull DatabaseError databaseError) {

            }
        });
    }
```

Рис. 13.3 Перевірка наявності користувача у БД

```
private void createUserInDatabase(String email) {  1 usage
    DatabaseReference usersRef = mDatabase.child( pathString: "users").push();
    int type = 0;
    Map<String, Object> userData = new HashMap<>();
    userData.put( k: "email", email);
    userData.put( k: "type", type);
    usersRef.updateChildren(userData);
    navigateToSecond(type);
}
```

Рис. 13.4 Створення нового запису в БД про користувача

```
private void navigateToSecond(int userType) {  2 usages
    Intent intent;
    if (userType == 0) {
        intent = new Intent( packageContext: MainActivity.this, customer_main.class);
    } else {
        intent = new Intent( packageContext: MainActivity.this, driver.class);
        intent.putExtra( name: "typeCar", userType);
    }
    startActivity(intent);
    finish();
}
```

Рис. 13.5 Відкриття наступного вікна в залежності від типу облікового запису

ДОДАТОК Б

Код для забезпечення роботи вікна пасажира

```
private void checkOnTrip(String email){  1 usage
    DatabaseReference databaseReference = mDatabase.child( pathString: "trip");
    Query query = databaseReference.orderByChild( path: "emailcustomer").equalTo(email);
    query.addListenerForSingleValueEvent(new ValueEventListener() {
        @Override
        public void onDataChange(@NonNull DataSnapshot date) {
            if (date.exists()) {
                for (DataSnapshot dataSnapshot : date.getChildren()) {
                    Long type = dataSnapshot.child( path: "type").getValue(Long.class);
                    if(type==1||type==3){
                        idTrip = dataSnapshot.child( path: "idtrip").getValue(Integer.class);
                        aBoolean=true;
                        getInfo(idTrip);
                        startListenerFirebase(idTrip);
                        getNewMessage();
                    }
                    else if(type==0){
                        idTrip = dataSnapshot.child( path: "idtrip").getValue(Integer.class);
                        getInfo(idTrip);
                        aBoolean=true;
                        getDriver();
                        startListenerFirebase(idTrip);
                    }
                }
            }
        }
        @Override
        public void onCancelled(@NonNull DatabaseError error) {
        }
```

Рис. 17.1 Перевірка незавершеної поїздки в користувача

```
private void getInfo(int id) {  2 usages
    DatabaseReference databaseReference = mDatabase.child( pathString: "trip");
    Query query = databaseReference.orderByChild( path: "idtrip").equalTo(id);
    query.addListenerForSingleValueEvent(new ValueEventListener() {
        @Override
        public void onDataChange(@NonNull DataSnapshot date) {
            if (date.exists()) {
                for (DataSnapshot userShot : date.getChildren()) {
                    Long coust = userShot.child( path: "coust").getValue(Long.class);
                    coustText.setText("Вартість "+coust);
                    DataSnapshot latlngStartSnapshot = userShot.child( path: "latlngStart");
                    if (latlngStartSnapshot.hasChild( path: "latitude") && latlngStartSnapshot.hasChild( path: "longitude")) {
                        double latitude = latlngStartSnapshot.child( path: "latitude").getValue(Double.class);
                        double longitude = latlngStartSnapshot.child( path: "longitude").getValue(Double.class);
                        latLngStart = new LatLng(latitude, longitude);

                    }

                    DataSnapshot latlngEndShot = userShot.child( path: "latlngEnd");
                    if (latlngEndShot.hasChild( path: "latitude") && latlngEndShot.hasChild( path: "longitude")) {
                        double latitude = latlngEndShot.child( path: "latitude").getValue(Double.class);
                        double longitude = latlngEndShot.child( path: "longitude").getValue(Double.class);
                        latLngEnd = new LatLng(latitude, longitude);
                    }
                    setText();
                }

            }
        }
```

Рис. 17.2 Метод для отримання координат маршуту

```
private void setText() {  1 usage
    Locale locale = new Locale( language: "uk",  country: "UA");
    Geocoder geocoder = new Geocoder( context: customer_main.this, locale);

    try {
        List<Address> addresses = geocoder.getFromLocation(latLngStart.latitude, latLngStart.longitude,  maxResults: 1);
        if (!addresses.isEmpty()) {
            Address address = addresses.get(0);
            String streetName = address.getAddressLine( index: 0);
            if (streetName.contains(",")) {
                editTextStart.setText(streetName.substring(0, streetName.lastIndexOf( str: ",")).trim());
            } else {
                editTextStart.setText(streetName.trim());
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
    }

    try {
        List<Address> addressesEnd = geocoder.getFromLocation(latLngEnd.latitude, latLngEnd.longitude,  maxResults: 1);
        if (!addressesEnd.isEmpty()) {
            Address address = addressesEnd.get(0);
            String streetNameE = address.getAddressLine( index: 0);
            if (streetNameE.contains(",")) {
                editTextEnd.setText(streetNameE.substring(0, streetNameE.lastIndexOf( str: ",")).trim());
            } else {
                editTextEnd.setText(streetNameE.trim());
            }
        }
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
    coustText.setVisibility(View.VISIBLE);
    buttonOrder.setEnabled(false);
    imageView1.setEnabled(false);
    imageView2.setEnabled(false);
    imageView3.setEnabled(false);
    buttonCancel.setEnabled(true);
    LinearLayoutFull.setVisibility(View.GONE);
    buttonCancel.setVisibility(View.VISIBLE);

}
```

Рис. 17.3 Метод setText()

```
private void startListenerFirebase(int id){  3 usages
    Query query = databaseReference.orderByChild( path: "idtrip").equalTo(id);
    listenerType = new ValueEventListener() {
        @Override
        public void onDataChange(@NonNull DataSnapshot snapshot) {
            if (aBoolean&&isCustomerActivity){
                for (DataSnapshot tripSnapshot : snapshot.getChildren()) {
                    Long type = tripSnapshot.child( path: "type").getValue(Long.class);
                    if(type==1){
                        isDriver=true;
                        getDataOfDriver(id);
                        break;
                    }
                    else if (type == 3) {
                        isDriver=false;
                        String idchat = String.valueOf(id);
                        idchat=extractDigits(idchat);
                        Toast.makeText( context: customer_main.this, text: "Водій відмінив поїздку", Toast.LENGTH_SHORT).show();
                        deleteChat(idchat);
                        getDriver();
                        if (driverMarker!=null)driverMarker.remove();
                        break;
                    }
                    else if (type == 4) {
                        isDriver=false;
                        cleanAll();
                        aBoolean=false;
                        break;
                    }

                }
```

Рис. 17.4 Метод startListenerFirebase()

```
private void getDataOfDriver(int id){  1 usage
    DatabaseReference databaseReference = mDatabase.child( pathString: "trip");
    Query query = databaseReference.orderByChild( path: "idtrip").equalTo(id);
    query.addListenerForSingleValueEvent(new ValueEventListener() {
        @Override
        public void onDataChange(@NonNull DataSnapshot date) {
            if (date.exists()) {
                for (DataSnapshot dataSnapshot : date.getChildren()) {
                    String nameDriver = dataSnapshot.child( path: "namedriver").getValue(String.class);
                    textViewDriverName.setText(nameDriver);
                }
                loadingProgressBar.setVisibility(View.INVISIBLE);
                textViewInfo.setText("Ваш водій: ");
                textViewInfo.setVisibility(View.VISIBLE);
                imageViewChat.setEnabled(true);
                imageViewChat.setVisibility(View.VISIBLE);
                textViewDriverName.setVisibility(View.VISIBLE);
                coustText.setVisibility(View.VISIBLE);
                getDriverLocation(id);
            }

        }

        @Override
        public void onCancelled(@NonNull DatabaseError error) {

        }
    });
```

```
private void getDriverLocation(int id){  1 usage

    DatabaseReference databaseReference = mDatabase.child( pathString: "trip");
    Query query = databaseReference.orderByChild( path: "idtrip").equalTo(id);
    listenerType = new ValueEventListener() {
        @Override
        public void onDataChange(@NonNull DataSnapshot snapshot) {
            if (aBoolean&&isDriver) {
                if(driverMarker!=null){
                    driverMarker.remove();
                }
                for (DataSnapshot driverSnapshot : snapshot.getChildren()) {
                        DataSnapshot driverGPS = driverSnapshot.child( path: "gpsdriver");
                        if(driverGPS.hasChild( path: "latitude")&&driverGPS.hasChild( path: "longitude")){
                            double latitude = driverGPS.child( path: "latitude").getValue(Double.class);
                            double longitude = driverGPS.child( path: "longitude").getValue(Double.class);
                            LatLng location = new LatLng(latitude, longitude);
                            BitmapDescriptor carIcon = BitmapDescriptorFactory.fromResource(R.drawable.caricon);
                            driverMarker = mMap.addMarker(new MarkerOptions()
                                    .position(location)
                                    .title("Ваш Водій")
                                    .icon(carIcon));
                        }
                }
            }
        }
```

Рис 17.5 Методи для отримання даних про водія

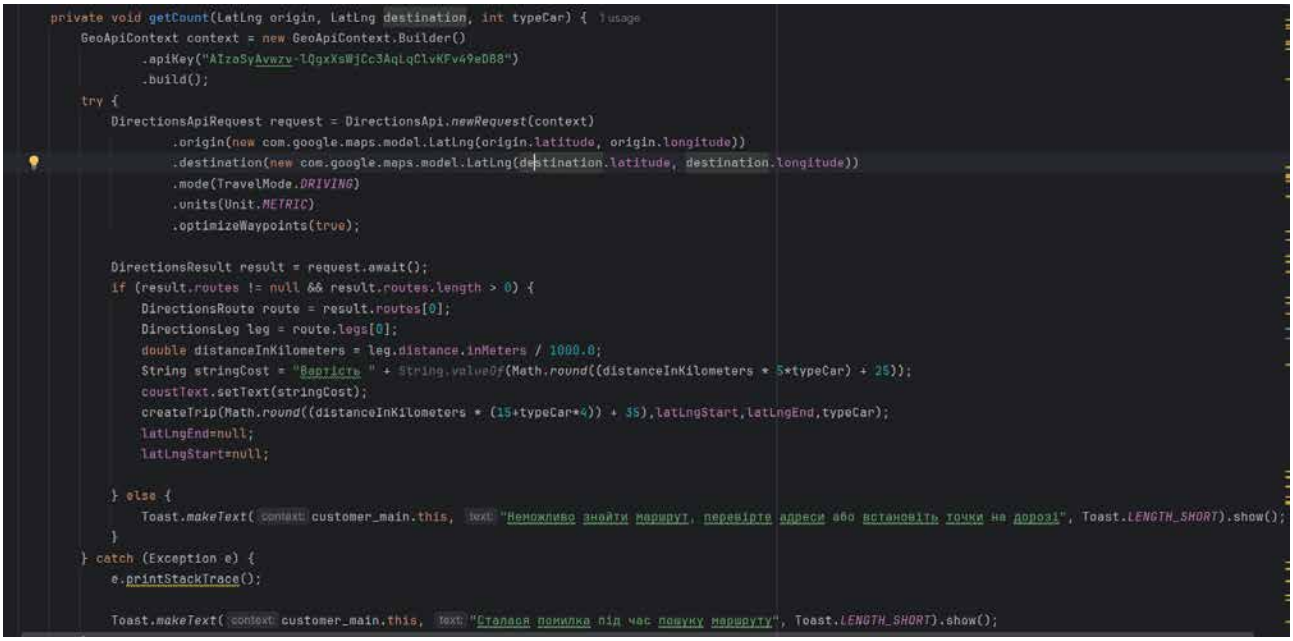
```
private void onExist() {  2 usages
    AlertDialog.Builder builder = new AlertDialog.Builder( context: this);
    builder.setTitle("Скасувати поїздку?");
    builder.setMessage("Ви впевнені, що хочете скасувати поїздку?");
    builder.setPositiveButton( text: "Так", new DialogInterface.OnClickListener() {
        @Override
        public void onClick(DialogInterface dialog, int which) {
            onCancel(idTrip);
            cleanAll();
        }
    });
    builder.setNegativeButton( text: "Ні", new DialogInterface.OnClickListener() {
        @Override
        public void onClick(DialogInterface dialog, int which) { dialog.dismiss(); }
    });
    builder.show();
}
private void onCancel(int id){  1 usage
    DatabaseReference databaseReference = mDatabase.child( pathString: "trip");
    Query query = databaseReference.orderByChild( path: "idtrip").equalTo(id);
    query.addListenerForSingleValueEvent(new ValueEventListener() {
        @Override
        public void onDataChange(@NonNull DataSnapshot snapshot) {
            if (snapshot.exists()) {
                for (DataSnapshot dataSnapshot : snapshot.getChildren()) {
                    dataSnapshot.getRef().child( pathString: "type").setValue(2);
                }
                aBoolean=false;

            }
```

Рис. 17.6 Методи для скасування поїздки

```java
private void getEnd() {  2 usages
    String streetName = String.valueOf(editTextEnd.getText()).trim();
    Geocoder geocoder = new Geocoder( context: customer_main.this, Locale.getDefault());

    if (!streetName.isEmpty()) {
        try {

            List<Address> addresses = geocoder.getFromLocationName( locationName: streetName + ", Україна",  maxResults: 1);

            if (!addresses.isEmpty()) {
                Address address = addresses.get(0);
                double latitude = address.getLatitude();
                double longitude = address.getLongitude();
                LatLng latLng = new LatLng(latitude, longitude);
                latLngEnd = latLng;
            } else {
                Toast.makeText( context: customer_main.this,
                        text: "Невірна адреса пункту призначення, спробуйте встановити міткою на карті",
                        Toast.LENGTH_LONG).show();
            }
        } catch (IOException e) {
            e.printStackTrace();
            Toast.makeText( context: customer_main.this,
                    text: "Помилка під час обробки адреси. Перевірте підключення до інтернету.",
                    Toast.LENGTH_SHORT).show();
        }
    } else {
        Toast.makeText( context: customer_main.this,  text: "Вкажіть місце призначення", Toast.LENGTH_SHORT).show();
    }
```

```java
private void getStart(){  2 usages
    String streetName = String.valueOf(editTextStart.getText()).trim();
    Geocoder geocoder = new Geocoder( context: customer_main.this, Locale.getDefault());

    if (!streetName.isEmpty()) {
        try {

            List<Address> addresses = geocoder.getFromLocationName( locationName: streetName + ", Україна",  maxResults: 1);

            if (!addresses.isEmpty()) {
                Address address = addresses.get(0);
                double latitude = address.getLatitude();
                double longitude = address.getLongitude();
                LatLng latLng = new LatLng(latitude, longitude);
                latLngStart = latLng;
            } else {
                Toast.makeText( context: customer_main.this,
                        text: "Невірна адреса пункту призначення, спробуйте встановити міткою на карті",
                        Toast.LENGTH_LONG).show();
            }
        } catch (IOException e) {
            e.printStackTrace();
            Toast.makeText( context: customer_main.this,
                    text: "Помилка під час обробки адреси. Перевірте підключення до інтернету.",
                    Toast.LENGTH_SHORT).show();
        }
    } else {
        Toast.makeText( context: customer_main.this,  text: "Вкажіть початок маршуту", Toast.LENGTH_SHORT).show();
    }
}
```

Рис. 17.7 Методи для отримання координат початку та закінчення маршуту



```
private void getCount(LatLng origin, LatLng destination, int typeCar) {  1 usage
    GeoApiContext context = new GeoApiContext.Builder()
            .apiKey("AIzaSyAvwzv-lQgxXsWjCc3AqLqClvKFv49eDB8")
            .build();
    try {
        DirectionsApiRequest request = DirectionsApi.newRequest(context)
                .origin(new com.google.maps.model.LatLng(origin.latitude, origin.longitude))
                .destination(new com.google.maps.model.LatLng(destination.latitude, destination.longitude))
                .mode(TravelMode.DRIVING)
                .units(Unit.METRIC)
                .optimizeWaypoints(true);

        DirectionsResult result = request.await();
        if (result.routes != null && result.routes.length > 0) {
            DirectionsRoute route = result.routes[0];
            DirectionsLeg leg = route.legs[0];
            double distanceInKilometers = leg.distance.inMeters / 1000.0;
            String stringCost = "Вартість " + String.valueOf(Math.round((distanceInKilometers * 5*typeCar) + 25));
            coustText.setText(stringCost);
            createTrip(Math.round((distanceInKilometers * (15+typeCar*4)) + 35),latLngStart,latLngEnd,typeCar);
            latLngEnd=null;
            latLngStart=null;

        } else {
            Toast.makeText( context: customer_main.this,  text: "Неможливо знайти маршрут, перевірте адреси або встановіть точки на дорозі", Toast.LENGTH_SHORT).show();
        }
    } catch (Exception e) {
        e.printStackTrace();

        Toast.makeText( context: customer_main.this,  text: "Сталася помилка під час пошуку маршруту", Toast.LENGTH_SHORT).show();
```

Рис. 17.8 Метод для перевірки можливості побудови маршуту між двома точками та розрахунку вартості.

```
private void createTrip( double coust,LatLng Start, LatLng End, int typeCar){  1 usage
    getIdTrip(new OnGetDataListener() {
        public void onSuccess(int id) {
            aBoolean=true;
            chatId= String.valueOf(id);
            idTrip=id;
            LatLng gpsDriver= new LatLng( latitude: 0, longitude: 0);
            int type = 0;
            Map<String, Object> tripData = new HashMap<>();
            tripData.put( k: "emailcustomer", userEmail);
            tripData.put( k: "type", type);
            tripData.put( k: "namedriver",  v: " ");
            tripData.put( k: "latlngStart", Start);
            tripData.put( k: "latlngEnd", End);
            tripData.put( k: "gpsdriver", gpsDriver);
            tripData.put( k: "idtrip", id);
            tripData.put( k: "coust", coust);
            tripData.put( k: "typecar",typeCar);
            tripsRef.updateChildren(tripData);
            LinearLayoutFull.setVisibility(View.GONE);
            buttonOrder.setEnabled(false);
            buttonCancel.setVisibility(View.VISIBLE);
            buttonCancel.setEnabled(true);
            textViewInfo.setVisibility(View.VISIBLE);
            coustText.setVisibility(View.VISIBLE);
            imageView1.setEnabled(false);
            imageView2.setEnabled(false);
            imageView3.setEnabled(false);
            loadingProgressBar.setVisibility(View.VISIBLE);
            startListenerFirebase(idTrip);
        }
```

Рис. 17.9 Метод для створення нової поїзди в БД

```
private void cleanAll(){  2 usages
    Toast.makeText( context: customer_main.this, text: "Поїздка завершена",Toast.LENGTH_SHORT).show();
    latLngStart =null;
    latLngEnd=null;
    latLngGPS=null;
    loadingProgressBar.setVisibility(View.INVISIBLE);
    coustText.setVisibility(View.INVISIBLE);
    textViewInfo.setVisibility(View.INVISIBLE);
    textViewDriverName.setVisibility(View.INVISIBLE);
    buttonCancel.setVisibility(View.INVISIBLE);
    imageView1.setEnabled(true);
    imageView2.setEnabled(true);
    imageView3.setEnabled(true);
    buttonCancel.setEnabled(false);
    buttonOrder.setEnabled(true);
    imageViewChat.setImageResource(R.drawable.chaticon);
    imageViewChat.setEnabled(false);
    imageViewChat.setVisibility(View.INVISIBLE);
    editTextStart.setText("");
    editTextEnd.setText("");
    editTextStart.setHint("Звідки Їдемо");
    editTextEnd.setHint("Куди Їдемо");
    if(driverMarker!=null) driverMarker.remove();
    LinearLayoutFull.setVisibility(View.VISIBLE);
    LinearLayoutFull.setEnabled(true);
    econom.setBackground(null);
    universal.setBackground(null);
    miniven.setBackground(null);
    bisness.setBackground(null);
    typeCar=0;
    aBoolean=false;
```

Рис. 17.10 Метод cleanAll()

```
private void deleteChat(String id) {  1 usage
    DatabaseReference ref = FirebaseDatabase.getInstance().getReference( path: "chat").child(id);
    ref.addListenerForSingleValueEvent(new ValueEventListener() {
        @Override
        public void onDataChange(@NonNull DataSnapshot dataSnapshot) {
            if (dataSnapshot.exists()) {
                ref.removeValue()
                        .addOnSuccessListener(new OnSuccessListener<Void>() {
                            @Override
                            public void onSuccess(Void aVoid) {

                            }
                        })
                        .addOnFailureListener(new OnFailureListener() {
                            @Override
                            public void onFailure(@NonNull Exception e) {

                            }
                        });
            }
        }

        @Override
        public void onCancelled(@NonNull DatabaseError databaseError) {
        }
    });
}
```

Рис. 17.11 Метод deleteChat()

```
private void getIdTrip(final OnGetDataListener listener) {  1 usage
    DatabaseReference databaseReference = FirebaseDatabase.getInstance().getReference( path: "trip");
    Query lastQuery = databaseReference.limitToLast(1);
    lastQuery.addListenerForSingleValueEvent(new ValueEventListener() {
        @Override
        public void onDataChange(DataSnapshot dataSnapshot) {
            int id = 1;
            if (dataSnapshot.exists()) {
                for (DataSnapshot childSnapshot : dataSnapshot.getChildren()) {
                    if (childSnapshot.exists()) {
                        int number = childSnapshot.child( path: "idtrip").getValue(Integer.class);
                        id = number + 1;
                    }
                }
            }
            listener.onSuccess(id);
        }

        @Override
        public void onCancelled(DatabaseError databaseError) {

            listener.onSuccess( id: 1);
        }
    });
}
```

Рис. 17.12 Метод getIdTrip()

```
private void getNewMessage(){  4 usages
    DatabaseReference databaseReference =  mDatabase.child( pathString: "chat").child(String.valueOf(idTrip));
    databaseListener = new ValueEventListener() {
        List<ChatData> chatDatesList = new ArrayList<>();  3 usages
        @SuppressLint("NotifyDataSetChanged")
        @Override
        public void onDataChange(DataSnapshot dataSnapshot) {
            chatDatesList.clear();
            for (DataSnapshot snapshot : dataSnapshot.getChildren()) {
                if(snapshot.exists()){
                        ChatData chatData = new ChatData( text1: "", text2: "", type: 1);
                        chatDatesList.add(chatData);
                    if(chatDatesList.size()>numberChat){
                        imageViewChat.setImageResource(R.drawable.chaticon_new);
                    }
                    else {
                        imageViewChat.setImageResource(R.drawable.chaticon);
                    }
                    }
                }
        }
        @Override
        public void onCancelled(DatabaseError databaseError) {
        }
    };
        databaseReference.addValueEventListener(databaseListener);

    }
```

Рис. 17.13 Метод для перевірки нових повідомлень в чаті

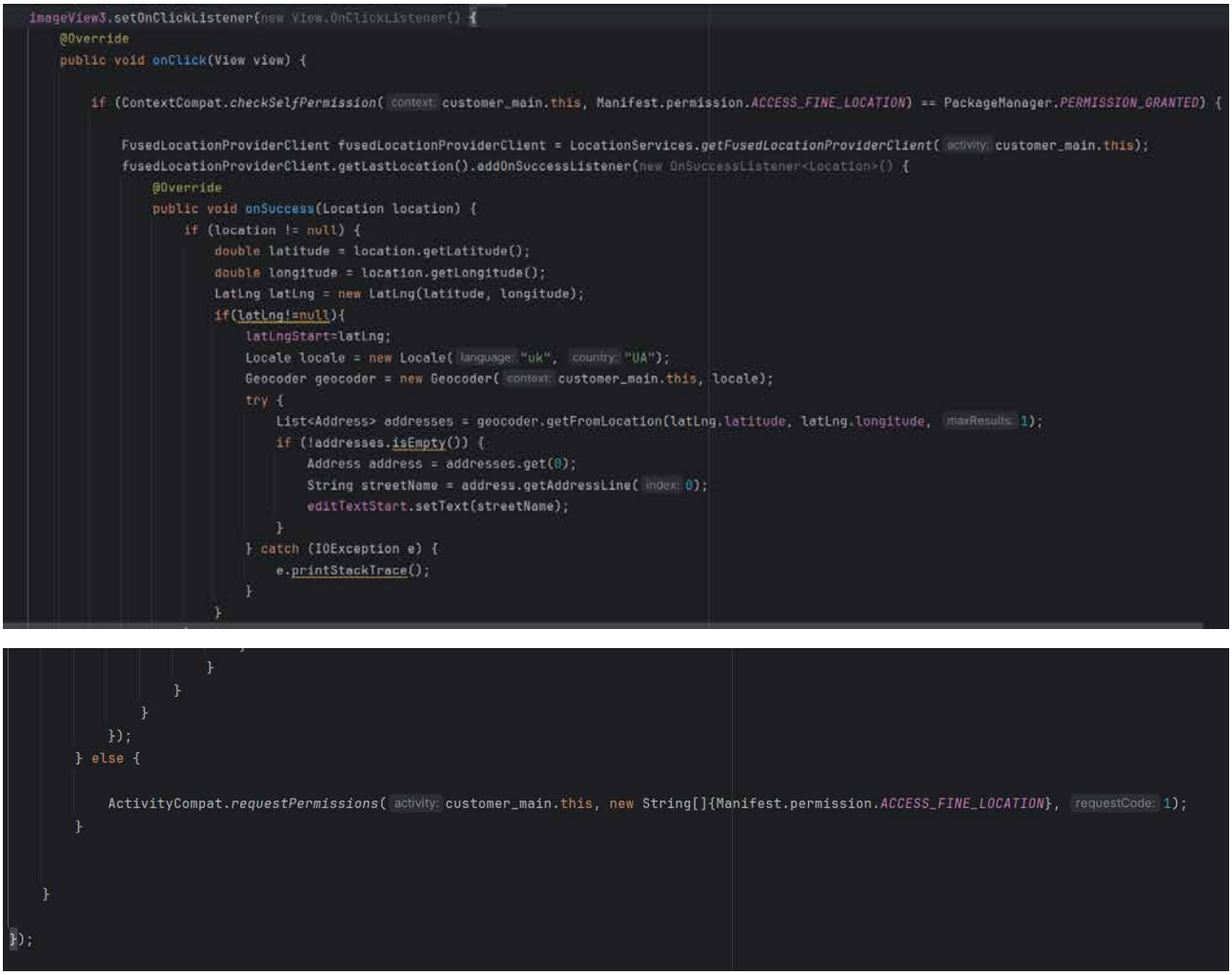
```
    imageViewS.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        gsc.signOut().addOnCompleteListener(new OnCompleteListener<Void>() {
            @Override
            public void onComplete(Task<Void> task) {
                finish();
                startActivity(new Intent( packageContext: customer_main.this,MainActivity.class));
            }
        });
    }

});
```

Рис. 17.14 Метод для виходу з акаунту Google

```
imageView1.setOnClickListener(new View.OnClickListener() {
@Override
public void onClick(View view) {

    if(latLngOther!=null){
        latLngStart=latLngOther;
        Locale locale = new Locale( language: "uk",  country: "UA");
        Geocoder geocoder = new Geocoder( context: customer_main.this, locale);

        try {
            List<Address> addresses = geocoder.getFromLocation(latLngOther.latitude, latLngOther.longitude,  maxResults: 1);
            if (!addresses.isEmpty()) {
                Address address = addresses.get(0);
                String streetName = address.getAddressLine( index: 0);
                editTextStart.setText(streetName.substring(0,streetName.indexOf(", ")));

            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    else {
        Toast.makeText( context: customer_main.this, text: "Утримуйте на карті для створення точки",Toast.LENGTH_SHORT).show();
    }
```

```
imageView2.setOnClickListener(new View.OnClickListener() {
@Override
public void onClick(View view) {
    if(latLngOther!=null){
        latLngEnd=latLngOther;
        Locale locale = new Locale( language: "uk",  country: "UA");
        Geocoder geocoder = new Geocoder( context: customer_main.this, locale);
        try {
            List<Address> addresses = geocoder.getFromLocation(latLngOther.latitude, latLngOther.longitude,  maxResults: 1);
            if (!addresses.isEmpty()) {
                Address address = addresses.get(0);
                String streetName = address.getAddressLine( index: 0);
                editTextEnd.setText(streetName.substring(0,streetName.indexOf(", ")));

            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    else {
        Toast.makeText( context: customer_main.this, text: "Утримуйте на карті для створення точки",Toast.LENGTH_SHORT).show();
    }
}
```

Рис. 17.15 Метод для отримання координат з мітки на карті



```
imageView3.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {

        if (ContextCompat.checkSelfPermission( context: customer_main.this, Manifest.permission.ACCESS_FINE_LOCATION) == PackageManager.PERMISSION_GRANTED) {

            FusedLocationProviderClient fusedLocationProviderClient = LocationServices.getFusedLocationProviderClient( activity: customer_main.this);
            fusedLocationProviderClient.getLastLocation().addOnSuccessListener(new OnSuccessListener<Location>() {
                @Override
                public void onSuccess(Location location) {
                    if (location != null) {
                        double latitude = location.getLatitude();
                        double longitude = location.getLongitude();
                        LatLng latLng = new LatLng(latitude, longitude);
                        if(latLng!=null){
                            latLngStart=latLng;
                            Locale locale = new Locale( language: "uk",  country: "UA");
                            Geocoder geocoder = new Geocoder( context: customer_main.this, locale);
                            try {
                                List<Address> addresses = geocoder.getFromLocation(latLng.latitude, latLng.longitude,  maxResults: 1);
                                if (!addresses.isEmpty()) {
                                    Address address = addresses.get(0);
                                    String streetName = address.getAddressLine( index: 0);
                                    editTextStart.setText(streetName);
                                }
                            } catch (IOException e) {
                                e.printStackTrace();
                            }
                        }
                    }
                }
            });
        } else {

            ActivityCompat.requestPermissions( activity: customer_main.this, new String[]{Manifest.permission.ACCESS_FINE_LOCATION},  requestCode: 1);
        }


    }

});
```

Рис. 17.16 Метод для отримання координат початку маршуту від геолокації

ДОДАТОК В

Код для забезпечення роботи вікнон для водія

```
private void checkLocationPermissionAndSignIn() {  1 usage
    if (ContextCompat.checkSelfPermission( context: driver.this, Manifest.permission.ACCESS_FINE_LOCATION) == PackageManager.PERMISSION_GRANTED) {
        getDriverVocation();
        startListeningToFirebaseChanges();
    } else {
        ActivityCompat.requestPermissions( activity: driver.this, new String[]{Manifest.permission.ACCESS_FINE_LOCATION}, LOCATION_PERMISSION_REQUEST_CODE);
    }
}
```

Рис. 19.1 Метод для перевірки дозволу на використання місцезнахоження

```
private void getDriverVocation() {  1 usage
    if (ContextCompat.checkSelfPermission( context: driver.this, Manifest.permission.ACCESS_FINE_LOCATION) == PackageManager.PERMISSION_GRANTED) {

        FusedLocationProviderClient fusedLocationProviderClient = LocationServices.getFusedLocationProviderClient( activity: driver.this);
        fusedLocationProviderClient.getLastLocation().addOnSuccessListener(new OnSuccessListener<Location>() {
            @Override
            public void onSuccess(Location location) {
                if (location != null) {
                    double latitude = location.getLatitude();
                    double longitude = location.getLongitude();
                    latLngDriver= new LatLng(latitude, longitude);
                }
            }
        });
    } else {

        ActivityCompat.requestPermissions( activity: driver.this, new String[]{Manifest.permission.ACCESS_FINE_LOCATION},  requestCode: 1);
    }
}
```

Рис. 19.2 Метод для отримання координат поточного місцезнаходження водія

```
public String getTripDistance(LatLng latLngCustomer) {  1 usage
    String distanceTxt = "";
    GeoApiContext context = new GeoApiContext.Builder()
            .apiKey("AIzaSyDIsA6JL3GY8LQfaCC2tTN2XsZmltLaaIU")
            .build();
    try {
        DirectionsApiRequest request = DirectionsApi.newRequest(context)
                .origin(new com.google.maps.model.LatLng(latLngDriver.latitude, latLngDriver.longitude))
                .destination(new com.google.maps.model.LatLng(latLngCustomer.latitude, latLngCustomer.longitude))
                .mode(TravelMode.DRIVING)
                .units(Unit.METRIC)
                .optimizeWaypoints(true);
        DirectionsResult result = request.await();
        if (result.routes != null && result.routes.length > 0) {
            DirectionsRoute route = result.routes[0];
            DirectionsLeg leg = route.legs[0];
            double distanceInKilometers = leg.distance.inMeters / 1000.0;
            distanceTxt = String.valueOf(distanceInKilometers);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }

    return distanceTxt;
}
```

Рис. 19.3 Метод який вираховує відстань до замовника

```
private void startListeningToFirebaseChanges() {  2 usages
    DatabaseReference databaseReference = mDatabase.child( pathString: "trip");
    databaseListener = new ValueEventListener() {
        @SuppressLint("NotifyDataSetChanged")
        @Override
        public void onDataChange(DataSnapshot dataSnapshot) {
            taxiDataModelList.clear();
            for (DataSnapshot snapshot : dataSnapshot.getChildren()) {
                if (snapshot.hasChild( path: "coust") && snapshot.hasChild( path: "latlngStart")&&snapshot.hasChild( path: "idtrip")&&snapshot.hasChild( path: "type")) {
                    Long type =snapshot.child( path: "type").getValue(Long.class);
                    Long type_car = snapshot.child( path: "typeCar").getValue(Long.class);
                    if(type_car==typeCar){
                        if(type==0||type==3){
                            Long coust = snapshot.child( path: "coust").getValue(Long.class);
                            String coustString = String.valueOf(coust);
                            int tripId = snapshot.child( path: "idtrip").getValue(Integer.class);
                            DataSnapshot latlngStartSnapshot = snapshot.child( path: "latlngStart");
                            if (latlngStartSnapshot.hasChild( path: "latitude") && latlngStartSnapshot.hasChild( path: "longitude")) {
                                double latitude = latlngStartSnapshot.child( path: "latitude").getValue(Double.class);
                                double longitude = latlngStartSnapshot.child( path: "longitude").getValue(Double.class);
                                LatLng customerStart = new LatLng(latitude, longitude);
                                String distance = getTripDistance(customerStart);
                                String buttonName = "Обрати";
                                TaxiDataModel dataModel = new TaxiDataModel( text1: "Вартість "
                                        + coustString,  text2: "Відстань до клієнта " + distance, buttonName, tripId);
                                taxiDataModelList.add(dataModel);
                            }
                        }
                    }
                }
```

```
                                taxiDataModelList.add(dataModel);
                            }
                        }
                    }
                }
            }
            adapter.notifyDataSetChanged();
        }

        @Override
        public void onCancelled(DatabaseError databaseError) {

        }
    };

    databaseReference.addValueEventListener(databaseListener);
}
```

Рис. 19.4 Метод для перевірки нових замовлень в БД та виведення їх на екран

```java
package com.example.taxi;

import java.io.Serializable;

public class TaxiDataModel implements Serializable {  8 usages
    private String text1;  2 usages
    private String text2;  2 usages
    private String buttonLabel;  2 usages

    private int tripId;  2 usages

    public TaxiDataModel(String text1, String text2, String buttonLabel,int tripId) {  1 usage
        this.text1 = text1;
        this.text2 = text2;
        this.buttonLabel = buttonLabel;
        this.tripId=tripId;
    }

    public String getText1() { return text1; }

    public String getText2() { return text2; }

    public String getButtonLabel() { return buttonLabel; }
    public int getID() { return tripId; }
}
```

Рис. 19.5 Клас TaxiDataModel

```java
public class TaxiAdapter extends RecyclerView.Adapter<TaxiAdapter.ViewHolder> {  3 usages
    private List<TaxiDataModel> dataList;  3 usages

    public TaxiAdapter(List<TaxiDataModel> dataList) { this.dataList = dataList; }

    public class ViewHolder extends RecyclerView.ViewHolder {  4 usages
        public TextView textView1;  2 usages
        public TextView textView2;  2 usages
        public Button button;  3 usages

        public ViewHolder(View itemView) {  1 usage
            super(itemView);
            textView1 = itemView.findViewById(R.id.textView1);
            textView2 = itemView.findViewById(R.id.textView2);
            button = itemView.findViewById(R.id.button);
        }
    }

    @Override
    public ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
        View view = LayoutInflater.from(parent.getContext()).inflate(R.layout.list_item, parent,  attachToRoot: false);
        return new ViewHolder(view);
    }
```

```java
    @Override
    public void onBindViewHolder(ViewHolder holder, int position) {
        TaxiDataModel data = dataList.get(position);
        holder.textView1.setText(data.getText1());
        holder.textView2.setText(data.getText2());
        holder.button.setText(data.getButtonLabel());


        holder.button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {

                Intent intent = new Intent(view.getContext(), driver_map.class);

                intent.putExtra( name: "data", data);

                view.getContext().startActivity(intent);
            }
        });

    }

    @Override
    public int getItemCount() { return dataList.size(); }
}
```

Рис. 19.6 Клас TaxiAdapter

```java
Intent intent = getIntent();
TaxiDataModel data = (TaxiDataModel) intent.getSerializableExtra( name: "data");
```

Рис. 21.1 Фрагмент коду для отримання даних з поперенього вікна

```
private void getInfo(int id) {  1 usage
    DatabaseReference databaseReference = mDatabase.child( pathString: "trip");
    Query query = databaseReference.orderByChild( path: "idtrip").equalTo(id);
    query.addListenerForSingleValueEvent(new ValueEventListener() {
        @Override
        public void onDataChange(@NonNull DataSnapshot date) {
            if (date.exists()) {
                for (DataSnapshot userShot : date.getChildren()) {
                    DataSnapshot latlngStartSnapshot = userShot.child( path: "latlngStart");
                    if (latlngStartSnapshot.hasChild( path: "latitude") && latlngStartSnapshot.hasChild( path: "longitude")) {
                        double latitude = latlngStartSnapshot.child( path: "latitude").getValue(Double.class);
                        double longitude = latlngStartSnapshot.child( path: "longitude").getValue(Double.class);
                        latlngCustomerStart = new LatLng(latitude, longitude);

                    }

                    DataSnapshot latlngEndShot = userShot.child( path: "latlngEnd");
                    if (latlngEndShot.hasChild( path: "latitude") && latlngEndShot.hasChild( path: "longitude")) {
                        double latitude = latlngEndShot.child( path: "latitude").getValue(Double.class);
                        double longitude = latlngEndShot.child( path: "longitude").getValue(Double.class);
                        latlngCustomerEnd = new LatLng(latitude, longitude);
                    }
                    setText();
                }

            }
        }
```

Рис. 21.2 Метод для отримання координат маршуту з БД

```
private void setText() {  1 usage
    Locale locale = new Locale( language: "uk",  country: "UA");
    Geocoder geocoder = new Geocoder( context: driver_map.this, locale);
    try {
        List<Address> addressesStart = geocoder.getFromLocation(latlngCustomerStart.latitude, latlngCustomerStart.longitude,  maxResults: 1);
        if (!addressesStart.isEmpty()) {
            Address address = addressesStart.get(0);
            String streetName = address.getAddressLine( index: 0);
            if (streetName.contains(",")) {
                ThenTxt.setText("Звідки: " + streetName.substring(0, streetName.lastIndexOf( str: ",")).trim());
            } else {
                ThenTxt.setText("Звідки: " +streetName.trim());
            }

        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    try {
        List<Address> addressesEnd = geocoder.getFromLocation(latlngCustomerEnd.latitude, latlngCustomerEnd.longitude,  maxResults: 1);
        if (!addressesEnd.isEmpty()) {
            Address address = addressesEnd .get(0);
            String streetName = address.getAddressLine( index: 0);
            if (streetName.contains(",")) {
                WhereTxt.setText("Куди: " + streetName.substring(0, streetName.lastIndexOf( str: ",")).trim());
            } else {
                WhereTxt.setText("Куди: " +streetName.trim());
            }

        }
```

Рис. 21.3 Метод для відображення адрес початку та завершення маршуту

```
private void setDateOnFirebase(int id) {  1 usage
    DatabaseReference databaseReference = mDatabase.child( pathString: "trip");
    Query query = databaseReference.orderByChild( path: "idtrip").equalTo(id);
    query.addListenerForSingleValueEvent(new ValueEventListener() {
        @Override
        public void onDataChange(@NonNull DataSnapshot snapshot) {
            if (snapshot.exists()) {
                for (DataSnapshot dataSnapshot : snapshot.getChildren()) {
                    dataSnapshot.getRef().child( pathString: "namedriver").setValue(driverEmail);
                    dataSnapshot.getRef().child( pathString: "type").setValue(1);
                }

            }

        }

        @Override
        public void onCancelled(@NonNull DatabaseError error) {

        }
    });

}
```

Рис. 21.4 Метод для передачі даних про водія в БД

```
private void setDriverLocation(int id){  1 usage
    DatabaseReference databaseReference = mDatabase.child( pathString: "trip");
    Query query = databaseReference.orderByChild( path: "idtrip").equalTo(id);
    query.addListenerForSingleValueEvent(new ValueEventListener() {
        @Override
        public void onDataChange(@NonNull DataSnapshot snapshot) {
            if (snapshot.exists()) {
                for (DataSnapshot dataSnapshot : snapshot.getChildren()) {
                    dataSnapshot.getRef().child( pathString: "gpsdriver").setValue(latLngDriverGPS);
                }

            }

        }

        @Override
        public void onCancelled(@NonNull DatabaseError error) {

        }
    });
}
```

Рис. 21.5 Метод для передачі геолокації водія до БД

```
private void startLocationUpdates() {  1 usage
    LocationRequest locationRequest = LocationRequest.create();
    locationRequest.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY);
    locationRequest.setInterval(5000);
    LocationCallback locationCallback = new LocationCallback() {
        @Override  no usages
        public void onLocationResult(LocationResult locationResult) {
            if (isActivity) {
                if (locationResult != null) {
                    Location location = locationResult.getLastLocation();
                    if (location != null) {
                        mLastLocation = location;
                        double latitude = location.getLatitude();
                        double longitude = location.getLongitude();
                        latLngDriverGPS = new LatLng(latitude, longitude);
                        createRoute();
                        setDriverLocation(tripID);
                        if (mUserMarker != null) {
                            mUserMarker.remove();
                        }
                        LatLng userLatLng = new LatLng(location.getLatitude(), location.getLongitude());
                        BitmapDescriptor carIcon = BitmapDescriptorFactory.fromResource(R.drawable.caricon);
                        mUserMarker = mMap.addMarker(new MarkerOptions()
                                .position(userLatLng)
                                .title("Ви тут")
                                .icon(carIcon));
                        mMap.moveCamera(CameraUpdateFactory.newLatLngZoom(userLatLng,  zoom: 15));
                    }
                }
            }
```

Рис. 21.6 Метод періодичного оновлення даних про геолокацію водія

```
private void createRoute() {  1 usage
    if (latlngCustomerStart != null && latlngCustomerEnd!= null&& latLngDriverGPS!=null) {
        LatLng origin = latLngDriverGPS;
        LatLng destination1 = latlngCustomerStart;
        LatLng destination2 = latlngCustomerEnd;
        GeoApiContext context = new GeoApiContext.Builder()
                .apiKey("AIzaSyAvwzv-lQgxXsWjCc3AqLqClvKFv49eDB8")
                .build();

        try {
            DirectionsResult result1 = DirectionsApi.newRequest(context)
                    .mode(TravelMode.DRIVING)
                    .origin(new com.google.maps.model.LatLng(origin.latitude, origin.longitude))
                    .destination(new com.google.maps.model.LatLng(destination1.latitude, destination1.longitude))
                    .units(Unit.METRIC)
                    .await();
            DirectionsResult result2 = DirectionsApi.newRequest(context)
                    .mode(TravelMode.DRIVING)
                    .origin(new com.google.maps.model.LatLng(destination1.latitude, destination1.longitude))
                    .destination(new com.google.maps.model.LatLng(destination2.latitude, destination2.longitude))
                    .units(Unit.METRIC)
                    .await();

            displayRouteOnMap(result1, mMap);
            displayRouteOnMap(result2, mMap);

        } catch (InterruptedException | IOException e) {
            e.printStackTrace();
        } catch (com.google.maps.errors.ApiException e) {
            throw new RuntimeException(e);
        }
```

Рис. 21.7 Метод для побудови маршуту на карті

```
private void displayRouteOnMap(DirectionsResult result, GoogleMap map) { 2 usages
    if (result != null) {
        PolylineOptions lineOptions = new PolylineOptions();
        EncodedPolyline polyline = result.routes[0].overviewPolyline;
        List<com.google.maps.model.LatLng> decodedPath = polyline.decodePath();

        for (com.google.maps.model.LatLng latLng : decodedPath) {
            lineOptions.add(new LatLng(latLng.lat, latLng.lng));
        }

        Polyline mapPolyline = map.addPolyline(lineOptions);
        mapPolyline.setColor(ContextCompat.getColor( context: driver_map.this, R.color.black));
        mapPolyline.setClickable(true);
        int sizeInPx = (int) (30 * getResources().getDisplayMetrics().density);
        BitmapDescriptor marcetEnd = resizeBitmap(R.drawable.endicon, sizeInPx, sizeInPx);
        BitmapDescriptor marcetStart = resizeBitmap(R.drawable.staricon, sizeInPx, sizeInPx);

        map.addMarker(new MarkerOptions()
                .position(latlngCustomerStart)
                .title("Маркер 1")
                .icon(marcetStart));

        map.addMarker(new MarkerOptions()
                .position(latlngCustomerEnd)
                .title("Маркер 2")
                .icon(marcetEnd));
    }
```

Рис. 21.8 Метод для графічного відображення маршуту на карті

```
private void startLisenerType(int id){  1 usage

    DatabaseReference databaseReference = mDatabase.child( pathString: "trip");
    Query query = databaseReference.orderByChild( path: "idtrip").equalTo(id);
    listenerType = new ValueEventListener() {
        @Override
        public void onDataChange(@NonNull DataSnapshot snapshot) {
            if (isActivity){
                for (DataSnapshot tripSnapshot : snapshot.getChildren()) {
                    Long type = tripSnapshot.child( path: "type").getValue(Long.class);
                    if (type == 2) {
                        goDriverMain();
                        canselCastomer = true;
                        break;
                    }
                    else if (type == 4) {
                        canselCastomer = true;
                        goDriverMainF();
                        break;
                    }
                }
        }
        }

        @Override
        public void onCancelled(@NonNull DatabaseError error) {

        }
    };
    query.addValueEventListener(listenerType);
}
```

Рис. 21.9 Метод для отримнння оновлених даних про поїздку

```
private void finishTrip(int id){  1 usage
        DatabaseReference databaseReference = mDatabase.child( pathString: "trip");
        Query query = databaseReference.orderByChild( path: "idtrip").equalTo(id);
        query.addListenerForSingleValueEvent(new ValueEventListener() {
            @Override
            public void onDataChange(@NonNull DataSnapshot snapshot) {
                if (snapshot.exists()) {
                    for (DataSnapshot dataSnapshot : snapshot.getChildren()) {
                        dataSnapshot.getRef().child( pathString: "type").setValue(4);
                    }
                }
            }
            @Override
            public void onCancelled(@NonNull DatabaseError error) {
            }
        });
    }
```

Рис. 21.10 Метод для завершення поїздки

```
private void getNewMessage(){  3 usages
    DatabaseReference databaseReference =  mDatabase.child( pathString: "chat").child(String.valueOf(tripID));
    databaseListener = new ValueEventListener() {
        List<ChatData> chatDatesList = new ArrayList<>();  3 usages
        @SuppressLint("NotifyDataSetChanged")
        @Override
        public void onDataChange(DataSnapshot dataSnapshot) {
            chatDatesList.clear();
            for (DataSnapshot snapshot : dataSnapshot.getChildren()) {
                if(snapshot.exists()){
                    ChatData chatData = new ChatData( text1: "", text2: "", type: 1);
                    chatDatesList.add(chatData);
                    if(chatDatesList.size()>numberChat){
                        imageViewChatDriver.setImageResource(R.drawable.chaticon_new);
                    }
                    else {
                        imageViewChatDriver.setImageResource(R.drawable.chaticon);
                    }
                }
            }
        }
        @Override
        public void onCancelled(DatabaseError databaseError) {
        }
    };
    databaseReference.addValueEventListener(databaseListener);

}
```

Рис. 21.11 Метод для перевірки нових повідомлень в чаті

```
private void onExist() {  1 usage
    AlertDialog.Builder builder = new AlertDialog.Builder( context: this);
    builder.setTitle("Відмоаитись від поїздки?");
    builder.setMessage("Ви впевнені, що хочете відмовитись?");
    builder.setPositiveButton( text: "Так", new DialogInterface.OnClickListener() {
        @Override
        public void onClick(DialogInterface dialog, int which) { onCancel(tripID); }
    });
    builder.setNegativeButton( text: "Ні", new DialogInterface.OnClickListener() {
        @Override
        public void onClick(DialogInterface dialog, int which) { dialog.dismiss(); }
    });
    builder.show();
}
private void onCancel(int id){  2 usages

        DatabaseReference databaseReference = mDatabase.child( pathString: "trip");
        Query query = databaseReference.orderByChild( path: "idtrip").equalTo(id);
        query.addListenerForSingleValueEvent(new ValueEventListener() {
            @Override
            public void onDataChange(@NonNull DataSnapshot snapshot) {
                if (snapshot.exists()) {
                    for (DataSnapshot dataSnapshot : snapshot.getChildren()) {
                        dataSnapshot.getRef().child( pathString: "namedriver").setValue("");
                        dataSnapshot.getRef().child( pathString: "type").setValue(3);
                        LatLng latLngnull = new LatLng( latitude: 0.0,  longitude: 0.0);
                        dataSnapshot.getRef().child( pathString: "gpsdriver").setValue(latLngnull);
                    }

                }
```

Рис. 21.12 Методи для відмови від поїздки

ДОДАТОК Г

Код методів для забезпечення роботи чату

```
private void setTxt(){  1 usage
    DatabaseReference chatRef = mDatabase.child( pathString: "chat").child(chatId).push();
    Map<String, Object> chatData = new HashMap<>();
    String castomerText= String.valueOf(chatTextCastomer.getText());
        if(castomerText.matches( regex: ".*\\w.*")){
            chatData.put( k: "Text",castomerText);
            chatData.put( k: "type", v: 0);
        }
    chatRef.updateChildren(chatData);
    chatTextCastomer.setText("");
}
```

Рис. 23.1 Метод для відправки нових повідомлень

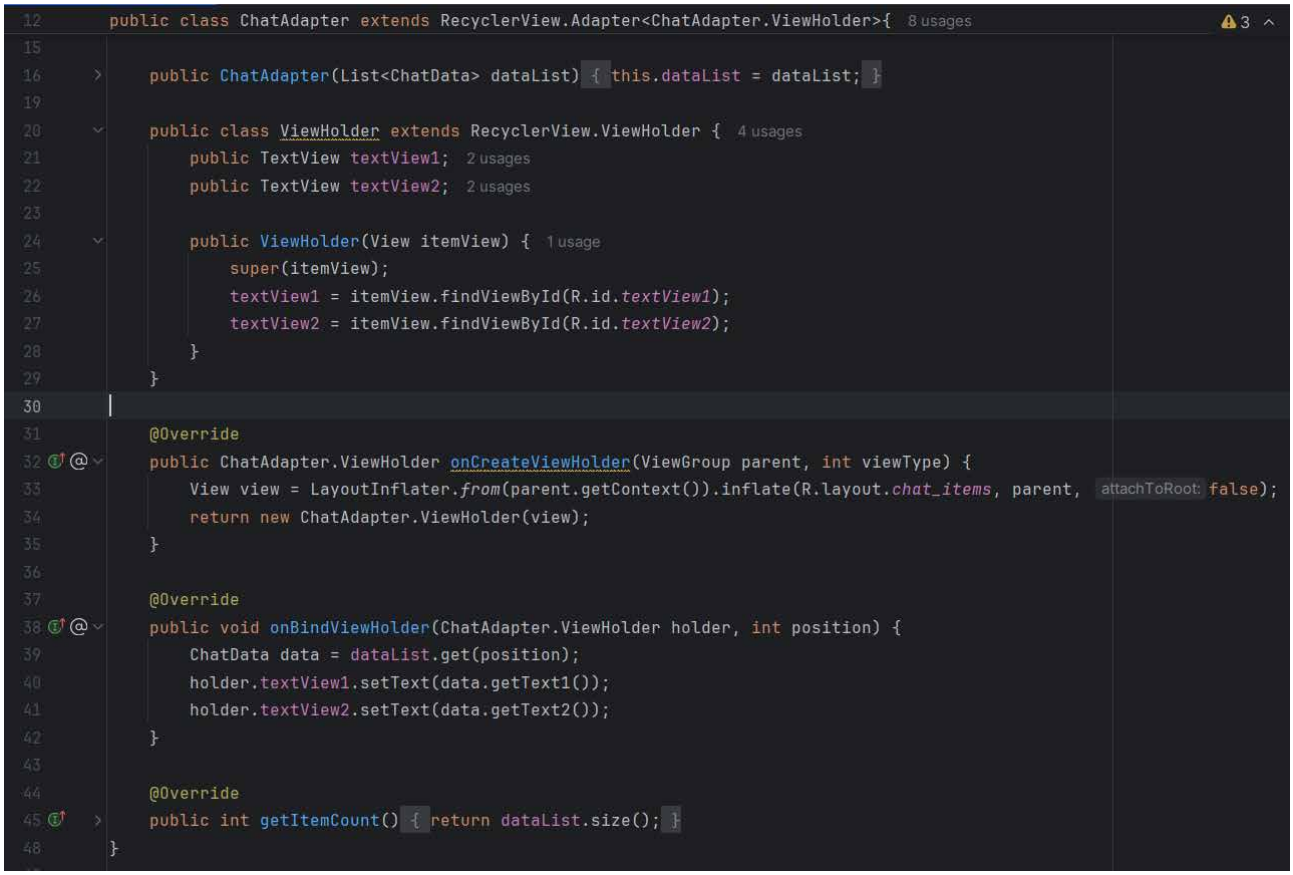
```
private void startListenetFirebaseTxt(boolean bool){  2 usages
    DatabaseReference databaseReference =  mDatabase.child( pathString: "chat").child(chatId);
    if(bool){databaseListener = new ValueEventListener() {
        @SuppressLint("NotifyDataSetChanged")
        @Override
        public void onDataChange(DataSnapshot dataSnapshot) {
            chatDatesList.clear();
            for (DataSnapshot snapshot : dataSnapshot.getChildren()) {
                if(snapshot.exists()){
                    Long type = snapshot.child( path: "type").getValue(Long.class);
                    if(type==0){
                        String text = snapshot.child( path: "Text").getValue(String.class);
                        ChatData chatData = new ChatData( text1: "",text, type: 0);
                        chatDatesList.add(chatData);
                    }
                    if(type==1){
                        String text = snapshot.child( path: "Text").getValue(String.class);
                        ChatData chatData = new ChatData(text, text2: "", type: 1);
                        chatDatesList.add(chatData);
                    }

                }

                if (chatDatesList.size() > 0) {
                    recyclerView.scrollToPosition(chatDatesList.size() - 1);
                }
            }
            adapterChatCastomer.notifyDataSetChanged();
        }
```

```
            @Override
            public void onCancelled(DatabaseError databaseError) {

            }
        };

            databaseReference.addValueEventListener(databaseListener);

        }else {
            if (databaseListener != null) {
                databaseReference.removeEventListener(databaseListener);
            }
    }
    }
```

Рис. 23.2 Метод для перевірки наявності нових повідомлень в БД

```
package com.example.taxi;

import java.io.Serializable;

public class ChatData implements Serializable {  19 usages
    private String text1;  2 usages
    private String text2;  2 usages

    private int type;  2 usages

    public ChatData(String text1, String text2, int type){  6 usages
        this.text1 = text1;
        this.text2 = text2;
        this.type=type;
    }
    public String getText1() { return text1; }

    public String getText2() { return text2; }

    public int getType(){return type;}  no usages

}
```

Рис. 23.3 Клас ChatData



```
public class ChatAdapter extends RecyclerView.Adapter<ChatAdapter.ViewHolder>{  8 usages
    public ChatAdapter(List<ChatData> dataList) { this.dataList = dataList; }

    public class ViewHolder extends RecyclerView.ViewHolder {  4 usages
        public TextView textView1;  2 usages
        public TextView textView2;  2 usages

        public ViewHolder(View itemView) {  1 usage
            super(itemView);
            textView1 = itemView.findViewById(R.id.textView1);
            textView2 = itemView.findViewById(R.id.textView2);
        }
    }

    @Override
    public ChatAdapter.ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
        View view = LayoutInflater.from(parent.getContext()).inflate(R.layout.chat_items, parent,  attachToRoot: false);
        return new ChatAdapter.ViewHolder(view);
    }

    @Override
    public void onBindViewHolder(ChatAdapter.ViewHolder holder, int position) {
        ChatData data = dataList.get(position);
        holder.textView1.setText(data.getText1());
        holder.textView2.setText(data.getText2());
    }

    @Override
    public int getItemCount() { return dataList.size(); }
}
```

Рис. 23.4 Клас ChatAdapter