МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Харківський національний університет імені В.Н.Каразіна

Факультет математики і інформатики

Кафедра теоретичної та прикладної інформатики

Кваліфікаційна робота

бакалавр

на тему: Проектування архітектури мобільного додатку «Mobil» для допомоги водіям на платформі Android

Виконав: студент 4 курсу, групи МФ-42

спеціальність 122 «Комп’ютерні науки»

освітня програма «Інформатика»

Олефіренко Ігор Ігорович

Керівник: ст. викладач Бережна Наталія Ігорівна

Рецензент:

Харків 2023

ЗМІСТ

1. ВСТУП……………………………………………………………...3

1.1 Мета та задачі роботи……………………………………………..4

1.2 Актуальність роботи………………………………………………5

2. РОЗРОБКА АРХІТЕКТУРИ ДОДАТКУ…………………………………7

2.1Опис функціональності системи………………………………….7

2.2 Огляд використаних технологій…………………………………...12

2.3 Моделювання бази даних…………………………………………..15

2.4 Концептуальна модель предметної галузі………………………...20

3. РОЗРОБКА СЕРВЕРНОЇ ЧАСТИНИ ДОДАТКУ……………………….28

3.1Основні архітектурні паттерни та шаблони……………………….28

3.2Опис моделі…………………………………………………………34

3.3Реалізація сервісів та контролерів………………………………….43

3.4Взаємодія моделі та контролера……………………………………51

3.5 Майбутні перспективи розвитку проекту………………………….54

ВИСНОВКИ…………………………………………………………………..56

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ………………………………..57

1. ВСТУП

Сучасний рівень розвитку інформаційних технологій дозволяє швидко та якісно реагувати на запити користувачів та ринку щодо найрізноманітніших потреб, задоволення яких забезпечують, зокрема, різноманітні веб та мобільні додатки.

Одним із перспективних та затребуваних напрямків є задоволення потреб автомобілістів. Власники автотранспортних засобів мають схожі проблеми, інформаційні запити тощо. Саме наявність значної кількості однакових або подібних пошукових запитів наштовхує на ідею їх узагальнення та створення інформаційного середовища, що дозволить автомобілістам знаходити інформацію в одному місці.

1.1 Мета та задачі роботи

Метою кваліфікаційної роботи є розробка мобільного додатку “Mobil” для автомобілістів, який надасть користувачам зручний інструментарій для керування даними про свої автомобілі, а також для знаходження корисної інформації та послуг.

Для досягнення поставленої мети необхідно виділити наступні задачі:

- Аналіз функціональності
- Розробка концептуальної моделі системи
- Проектування та розробка сховища даних
- Розробка серверної частини додатку

1.2 Актуальність роботи

У повсякденні люди часто використовують різні інструменти для спрощення їхнього життя, надавання актуальної інформації та різних важливих порад. Прикладом тут можуть бути додатки для здорового сну або здорового харчування. Як можна зрозуміти ці додатки допомагають своїм функціоналом стежити за певними речами у житті, тим самим допомагаючи заощаджувати час. Таку ціль і має наш додаток. Допомогти з актуалізацією інформації та порадами, зберігаючи час користувачів та надати їм зручний інструмент у повсякденному використанні

Актуальність теми створення додатку для автомобілістів базується на широкому поширенні автомобілів та зростаючих потребах власників авто щодо ефективного управління, обслуговування та забезпечення безпеки їх транспортного засобу.

По перше завдяки зручності та ефективності управління авто автомобіліст може легко відстежувати різні параметри свого автомобіля, такі як характеристики, дати останнього технічного обслуговування, заміни мастила, шин. Це допомагає власникам авто вчасно планувати та виконувати необхідні ремонтні роботи та обслуговування. Додаток також може надати користувачам інформацію про найпопулярніші або найдешевші сервіси по відгукам клієнтів для їх автомобілів. Це допомагає забезпечити оптимальний вибір запчастин та послуг майстрами, що зберігає час та гроші власників авто.

Більш того додаток має Форуми для обміну досвідом: Введення форумів у додаток надає можливість автомобілістам спілкуватися, ділитися своїм досвідом та знаходити відповіді на свої запитання. Це створює спільноту

автомобілістів, які можуть надавати підтримку та поради одне одному. Додаток може надсилати нагадування користувачам про заплановані терміни заміни мастила, шин, технічного обслуговування та інші важливі події. Це допомагає автомобілістам не пропускати важливе обслуговування, збільшує безпеку та термін служби автомобіля.

Додаток забезпечує зручний та швидкий доступ до всієї необхідної інформації. Користувачі можуть швидко перевірити характеристики свого авто, історію обслуговування, та іншу важливу інформацію, що дозволяє зекономити час та зусилля.

У першу чергу наш додаток має перевагу над іншими схожими у тому що він реалізує значний функціонал, тоді як інші більш вузького направлення. Також він надає можливість обмінюватися враженнями та досвідом у певних питаннях, та цим допомагає сформувати велику спілку людей зі схожими потребами та поглядами.

Загалом, створення додатку для автомобілістів актуально, оскільки воно відповідає зростаючому числу автомобілів на дорогах та залученням все більшої кількості людей до власництва авто, створюють потребу у зручних та ефективних інструментах для управління транспортними засобами та їх обслуговування.

2. РОЗРОБКА АРХІТЕКТУРИ ДОДАТКУ

2.1 Опис функціональності системи

Основні функції додатку:

1. Управління даними автомобіля:

1.1 Користувачі можуть вводити характеристики своїх автомобілів, такі як марка, модель, рік випуску, об'єм двигуна тощо.

1.2 Додаток буде зберігати інформацію про останній раз заміни мастила, заміни шин і інших планових обслуговувань.

2. Форуми:

2.1 Додаток буде мати функціонал форумів, де користувачі зможуть обговорювати різні теми, пов'язані з автомобілями.

2.2 Додаток надасть можливість користувачам обговорювати свої проблеми, знаходити поради та ділитися досвідом з іншими користувачами.

3. Інформація щодо ремонтних станцій:

3.1 Додаток надасть можливість знаходити адресу найближчих ремонтних станції до місця розташування користувача по відгукам попередніх автомобілістів.

3.2 За допомогою рейтингів та відгуків інших користувачів, додаток надасть можливість оцінити якість та надійність ремонтних станцій.

4. Індивідуальні рекомендації:

4.1 Додаток буде здатний надавати індивідуальні рекомендації користувачам на основі даних про їх автомобілі та вимог.

4.2 Засновуючись на історії використання автомобіля, додаток може рекомендувати оптимальний час для заміни деяких комплектуючих, огляду автомобіля.

5. Персоналізація:

5.1 Користувачі також зможуть налаштовувати сповіщення та отримувати повідомлення про актуальні події та новини від інших користувачів.

6. Аналітика:

6.1 Аналітичні інструменти дадуть змогу користувачам отримати уявлення про рівень задоволеності користувачів сервісами,

7. Монетизація

7.1 Додаток, у майбутньому може мати модель монетизації через рекламу або платні підписки.

Монетизаційні стратегії повинні бути розроблені з урахуванням інтересів користувачів та створити додаткову цінність для них.

При аналізі предметної галузі було виділено 2 типи акторів - користувач та адміністратор.

Функціональні можливості цих акторів формалізовані у вигляді відповідних use-case діаграм.

Для початку розглянемо можливості звичайного користувача.

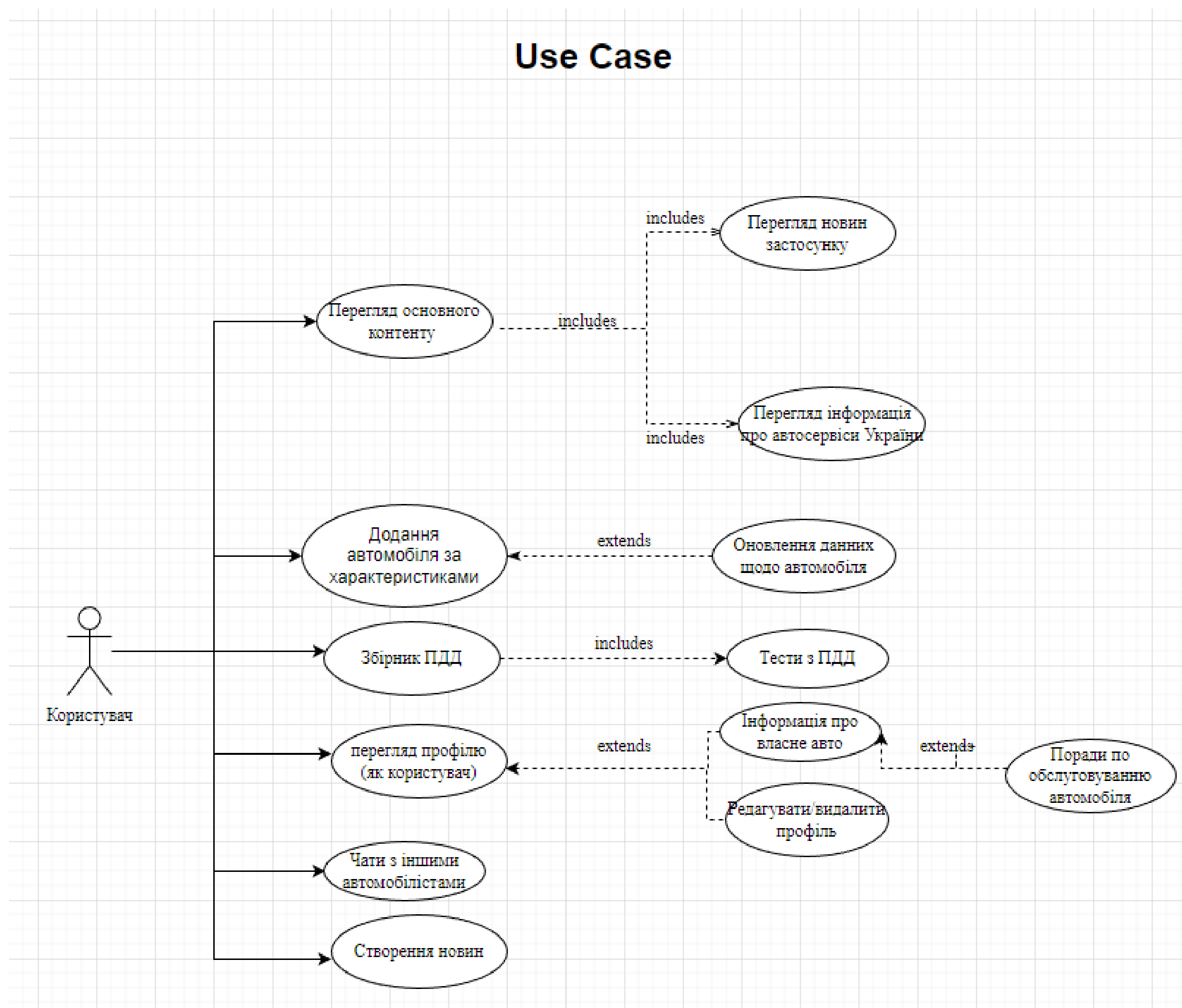

Рис. 1. Use-case користувач

Реєстрація та авторизація користувачів: Система надає можливість користувачам реєструватися за допомогою свого облікового запису або соціальних мереж. Також вона забезпечує механізм авторизації, що дозволяє користувачам увійти в систему та мати доступ до своїх особистих даних.

Додавання автомобіля за характеристиками: Система надає можливість додавати автомобіль за його характеристиками.

Перегляд основного контенту: Додаток дає можливість переглянути новини у застосунку та перегляд інформації та відгуків про автосервіси.

Тести правил дорожнього руху: Система дає можливість складати тести для перевірки власних здібностей та дивитися результат.

Перегляд профілю: Система дозволяє переглянути власний профіль, або редагувати чи видалити його за потреби.

Чат з іншими автомобілістами: Дозволяє задавати питання та брати участь у обговоренні між користувачами.

Створення новин: Система дає можливість створювати новини, які будуть потенційно цікаві іншим користувачам.

Актор з роллю "Адміністратор" по перше має всі ті ж можливості, що і звичайний користувач, а також ряд додаткових. Додаткові можливості відображені на відповідній діаграмі (рис. 2)

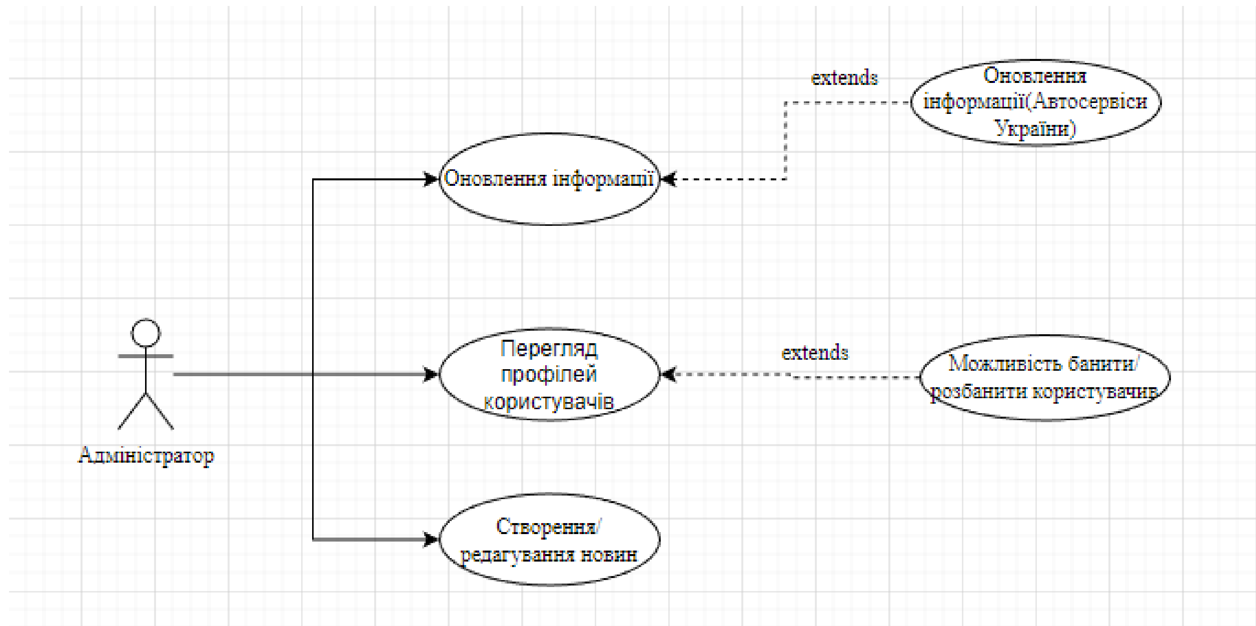

Рис. 2 Use-case адміністратор.

Оновлення інформації: Додаток дає можливість оновлювати да редагувати інформацію згідно автосервісів (видаляти автосервіси загалом чи видаляти або редагувати коментарі.

Перегляд профілей користувачів: Система дає можливість переглядати профілі користувачів та банити їх за необхідності.

Створення редагування новин: система дає можливість створювати чи редагувати новини.

2.2 Огляд використаних технологій

Java Spring Boot є однією з найпопулярніших технологій для розробки серверних додатків, і вона має багато переваг, які обґрунтовують її використання в проекті розробки додатку для автомобілістів.

Java є однією з найпоширеніших та найстабільніших мов програмування. Вона має велику спільноту розробників, багато ресурсів та документації, що полегшує розробку, підтримку та розширення додатків. Java також відома своєю надійністю та високою продуктивністю, що є важливими факторами для додатку, який має обробляти великі обсяги даних та багатокористувацькі взаємодії.

Spring Boot є фреймворком для розробки Java-додатків, який надає потужні інструменти для швидкого розгортання та налаштування серверних додатків. Він спрощує розробку, конфігурацію та управління додатками, дозволяючи розробникам зосередитись на бізнес-логіці додатку. Використання Java Spring Boot дозволяє розробникам швидко створювати серверну частину додатку. Фреймворк надає багато вбудованих функцій та інструментів, які спрощують конфігурацію та розгортання додатків. Це дозволяє ефективно використовувати час розробки і зосередитись на реалізації бізнес-логіки. Масштабованість: Spring Boot добре підтримує горизонтальне масштабування, що дозволяє збільшувати потужність додатку за потреби.

Простота конфігурації: Spring Boot Starter Web надає спрощений спосіб розгортання та налаштування веб-додатків. Він автоматично налаштовує сервлетний контейнер, обробник HTTP-запитів та інші необхідні компоненти. Це дозволяє розробникам швидко створювати веб-додатки без необхідності вручну налаштовувати багато компонентів.

Підтримка веб-стандартів: Spring Boot Starter Web вбудовує в себе підтримку веб-стандартів, таких як Servlet, WebSocket, RESTful API тощо. Це дозволяє розробникам створювати веб-додатки, які дотримуються стандартів та можуть легко інтегруватись з іншими веб-технологіями.

Автоматичне налаштування: Spring Boot Starter Web надає автоматичну настройку багатьох важливих аспектів веб-додатків. Він автоматично сканує та конфігурує контролери, фільтри та інші компоненти, що забезпечують обробку HTTP-запитів. Це спрощує розробку веб-додатків та зменшує кількість коду, який потрібно писати.

Spring Boot Starter Data JPA є одним з стартерів, що використовуються для роботи з Java Persistence API (JPA) в екосистемі Spring Boot. Спрощена конфігурація: Spring Boot Starter Data JPA надає простий спосіб налаштування та використання JPA в додатках. Він автоматично налаштовує необхідні компоненти, такі як EntityManagerFactory, TransactionManager та інші, що дозволяє розробникам зосередитись на бізнес-логіці замість налаштування JPA. Spring Boot Starter Data JPA базується на об'єктно-реляційному відображенні (ORM) і дозволяє розробникам працювати з базами даних з використанням об'єктно-орієнтованого підходу. Це спрощує взаємодію з базою даних та зменшує необхідність вручну писати складні SQL-запити. Spring Boot Starter Data JPA автоматично генерує SQL-запити на основі декларативного опису об'єктів JPA. Це дозволяє розробникам працювати з базою даних без необхідності писати власні SQL-запити, що спрощує розробку та зменшує кількість необхідного коду.

RESTful API: Розробка додатку для автомобілістів передбачає потребу у взаємодії з даними та функціями через мережу. RESTful API є популярним підходом для реалізації взаємодії між клієнтами та сервером.

Використання PostgreSQL як бази даних для додатку також має свої переваги і обґрунтування. PostgreSQL є однією з найбільш надійних та стабільних реляційних баз даних. Вона відома своєю високою стійкістю до відмов та забезпечує цілісність даних. Це особливо важливо для додатків, які працюють зі значною кількістю даних та вимагають високої надійності. PostgreSQL підтримує ACID (Atomicity, Consistency, Isolation, Durability) властивості, що забезпечують цілісність та надійність операцій з даними. Це важливо для додатків, які вимагають стійкості даних та можливості відновлення після збоїв. PostgreSQL надає широкий спектр функцій та можливостей, що дозволяють ефективно управляти та оптимізувати роботу з даними. Вона підтримує розширення за допомогою функцій, типів даних, мов програмування та інших компонентів. Це дозволяє адаптувати базу даних до специфічних потреб додатку та забезпечити гнучкість розробки.

2.3Моделювання бази даних

Для зберігання інформації було прийняте рішення використовувати реляційну базу даних.

Першим етапом проектування бази даних є виділення сутностей предметної галузі. Проаналізувавши як безпосередньо предметну галузь, так і функціонал системи, спираючись на розроблені use-case діаграми, можна виділити наступні сутності предметної галузі:

1. Користувач
2. Автомобіль
3. Сервіс
4. Тест
5. Новини
6. Відгук
7. Міста
8. Повідомлення

На другому етапі проектування бази даних були визначені зв'язки (та їх типи) між сутностями.

Моделювання бази даних для зв'язку користувача (юзера) з автомобілем було здійснено за допомогою використання реляційної моделі даних. У такій моделі будуть існувати таблиці: "User" (Користувач) та "Car" (Автомобіль). Доречно буде використати тип зв'язку один до одного (OneToOne) Зв'язок між ними був встановлений за допомогою стовпця "Car_id" в таблиці "User", який буде посилатися на первинний ключ "id" в таблиці "Car". Цей підхід дозволяє кожному користувачу мати один конкретний автомобіль і навпаки, кожен автомобіль пов'язаний з одним

користувачем. Таким чином, кожен запис в таблиці "User" буде мати посилання на відповідний запис в таблиці "Car", і навпаки. Таке моделювання бази даних дозволить вам легко здійснювати операції з отримання та збереження даних про користувачів та автомобілів. Наприклад, ви зможете отримати інформацію про автомобіль, пов'язаний з конкретним користувачем, або знайти користувача, який володіє певним автомобілем.

Далі розглянемо зв'язок користувача та новин. Було створено таблицю "News" для зберігання даних про новини. Вона містить поля, такі як "news_id" (унікальний ідентифікатор новини), "header" (заголовок новини), "Text" (зміст новини) та інші поля, які нам потрібні для новин. Далі було додано зовнішній ключ "user_id" до таблиці "News", який буде посилатися на "user_id" в таблиці "Users". Це створить зв'язок між користувачем і його новинами. Таким чином, кожен користувач може мати багато новин, але кожна новина пов'язана лише з одним користувачем.

Тепер опишемо зв’язок тесту і користувача, було реалізоване з’єднання зв’язком багато до багатьох (ManyToMany). Для моделювання зв'язку "багато до багатьох" між юзером і тестом, необхідна додаткова таблиця (User/Test), яка використовується як зв'язувальна таблиця між ними. Також у цю таблицю було додано поле score(оцінка користувача за певний тест). Така модель дозволяє зв'язувати кожного користувача з багатьма тестами і на навпаки, кожен тест може бути пов'язаний з багатьма користувачами. Зв'язок між ними визначається шляхом додавання відповідних записів у зв'язувальну таблицю "User/Test". Ця модель дозволяє ефективно виконувати операції, такі як отримання всіх тестів, пов'язаних з певним користувачем або всіх користувачів, які проходили певний тест. Також це

дає можливість легко додавати нові зв'язки між користувачами і тестами без необхідності змінювати структуру основних таблиць.

Тепер опишемо зв’язок користувача і повідомлення, очевидно, що один користувач може мати багато повідомлень, тож зв’язок буде один до багатьох. Для моделювання зв'язку "один до багатьох" між юзером і повідомленням, було додано стовпець "user_id" до таблиці "Message", який буде посилатися на первинний ключ "id" в таблиці "User". У цьому моделюванні, таблиця "User" містить дані про користувачів, а таблиця "Message" містить дані про повідомлення. У стовпці "user_id" таблиці "Message" зберігається ідентифікатор користувача, до якого належить повідомлення. Ця модель дозволяє кожному користувачу мати багато повідомлень, але кожне повідомлення належить лише одному користувачеві. Зв'язок між ними визначається шляхом встановлення значення стовпця "user_id" в таблиці "Message", яке посилається на ідентифікатор відповідного користувача в таблиці "User". Ця модель дозволяє легко виконувати операції, такі як отримання всіх повідомлень, пов'язаних з певним користувачем або всіх користувачів, які створили певне повідомлення.

Тепер розглянемо користувача та автосервіс. Для цього використаємо зв’язок багато до багатьох. Для моделювання зв'язку "багато до багатьох" між юзером і автосервісом було створено проміжну таблицю, яка використовується для зв'язку цих сутностей. У цьому моделюванні, таблиця "User" містить дані про користувачів, таблиця "Repair" містить дані про автосервіси, а проміжна таблиця "User_Repair" використовується для зв'язку юзерів і автосервісів. Кожен запис у проміжній таблиці

представляє зв'язок між певним користувачем і певним автосервісом. Ця модель дозволяє кожному користувачу мати багато автосервісів, а також кожен автосервіс може бути пов'язаний з багатьма користувачами. Зв'язок між ними визначається шляхом створення записів у проміжній таблиці "User_Repair", де стовпець "user_id" посилається на ідентифікатор користувача в таблиці "User", а стовпець "repair_id" посилається на ідентифікатор автосервісу в таблиці "Repair". Ця модель дозволяє легко виконувати операції, такі як отримання списку автосервісів, пов'язаних з певним користувачем або отримання списку користувачів, які використовують певний автосервіс.

Далі було створено сутність міста, яка буде зв’язана з юзером та автосервісами окремо зв’язком один до багатьох. Для моделювання зв'язку "один до багатьох" між містами, автосервісами і користувачами необхідно використовувати зовнішні ключі. Було додано у таблиці автосервіси та користувачі стовпець "city_id". Таким чином, кожен користувач може бути пов'язаний з одним містом, але місто може мати багато користувачів. Також кожен автосервіс може бути пов'язаний з одним містом, але місто може мати багато автосервісів. Кожен користувач також може бути пов'язаний з багатьма автосервісами і навпаки.

Тепер розглянемо сутність відгуки, яка має зв’язок з користувачами та автосервісами один до багатьох. Для моделювання зв'язку "один до багатьох" між відгуками, автосервісами і користувачами необхідно використовувати зовнішні ключі. У цьому моделюванні, таблиця "Feedback" містить дані про відгуки, а стовпці "user_id" та "repair_id" вказують на ідентифікатори користувача і автосервісу відповідно. Кожен відгук пов'язаний з конкретним користувачем і автосервісом. Таблиці

"User" і "Repair" містять інформацію про користувачів і автосервіси відповідно. Це моделювання дозволяє легко виконувати операції, такі як отримання списку відгуків, залишених певним користувачем або пов'язаних з певним автосервісом. Зв'язки між сутностями встановлюються за допомогою зовнішніх ключів, які посилаються на відповідні ідентифікатори в інших таблицях.

Наведено ER-діаграму з встановленими зв’язками та потрібними стовпцями:

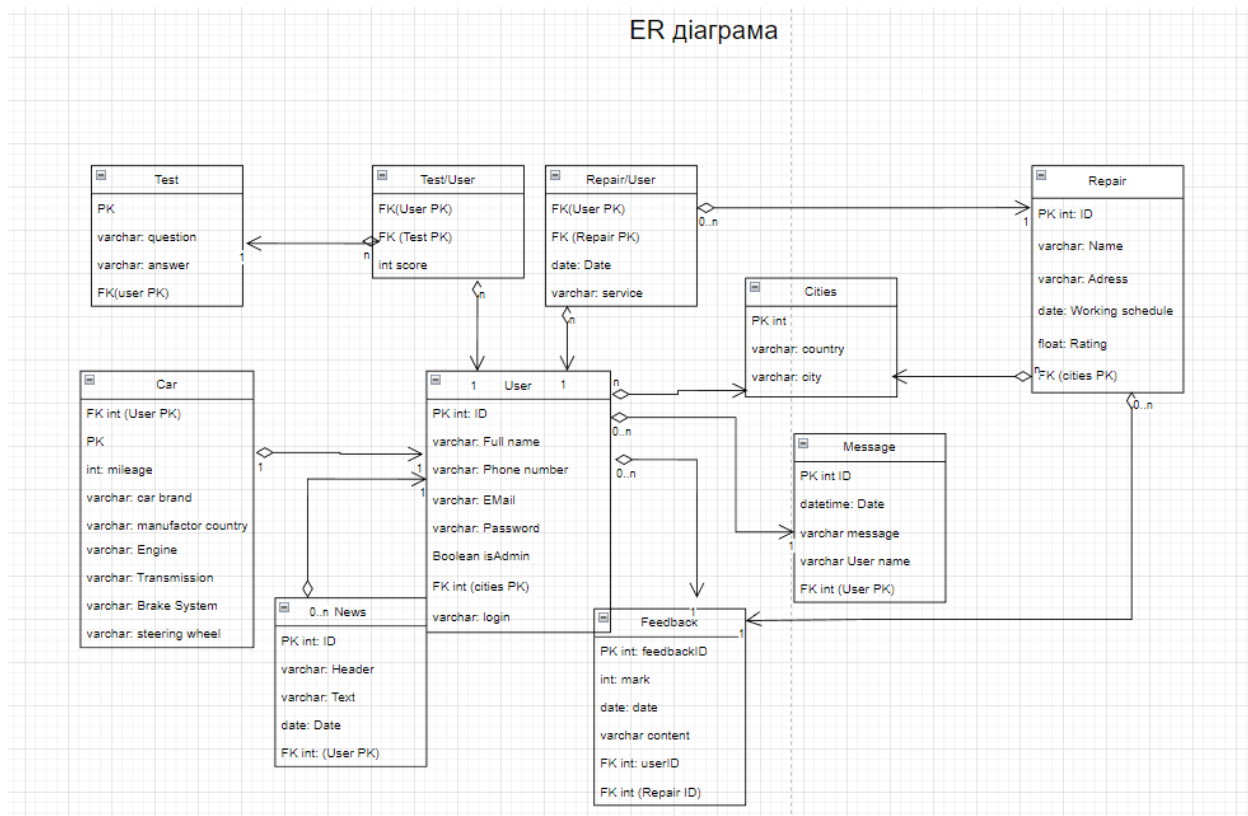

Рис. 3. ER-діаграма

2.4 Концептуальна модель предметної галузі

Розглянемо створений клас користувача. Він має назву Account та містить наступні поля:

```
private Long accountId;
private String fullName;
private String phoneNumber;
private String email;
private String login;
private String password;
private Boolean isAdmin;
private City city;
private List<News> newsList;
private List<Message> messages;
private Set<Repair> repairs = new HashSet<>();
private List<RepairAccount> repairAccounts = new ArrayList<>();
private List<Feedback> feedbacks = new ArrayList<>();
private List<TestAccount> testAccounts = new ArrayList<>();
private Car car;
```

Він має асоціації: "Один до одного" із автомобілем, "багато до одного" із містом, "один до багатьох" із новинами та повідомленнями, та "багато до багатьох" з автосервісами та тестами, також таблиці для з’єднання.

Наведено клас Account з діаграми класів (Рис 4):

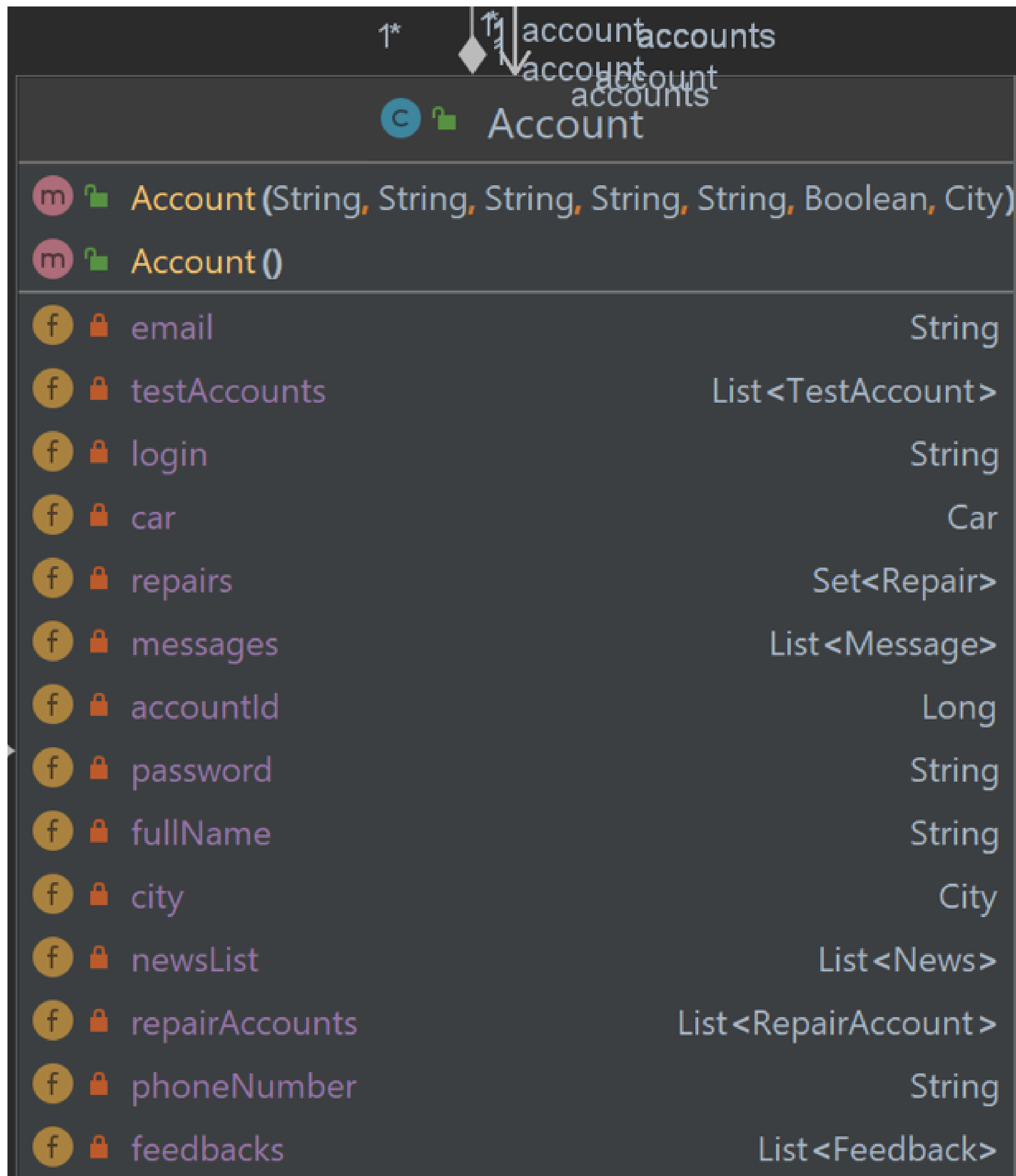

Рис. 4. Class-Diagram Account

Для зразка було наведемо з діаграми класів зв'язок користувача та повідомлення із повністю заповненими полями та методами:

Можна побачити реалізоване відношення "один до багатьох" користувача та повідомлень.

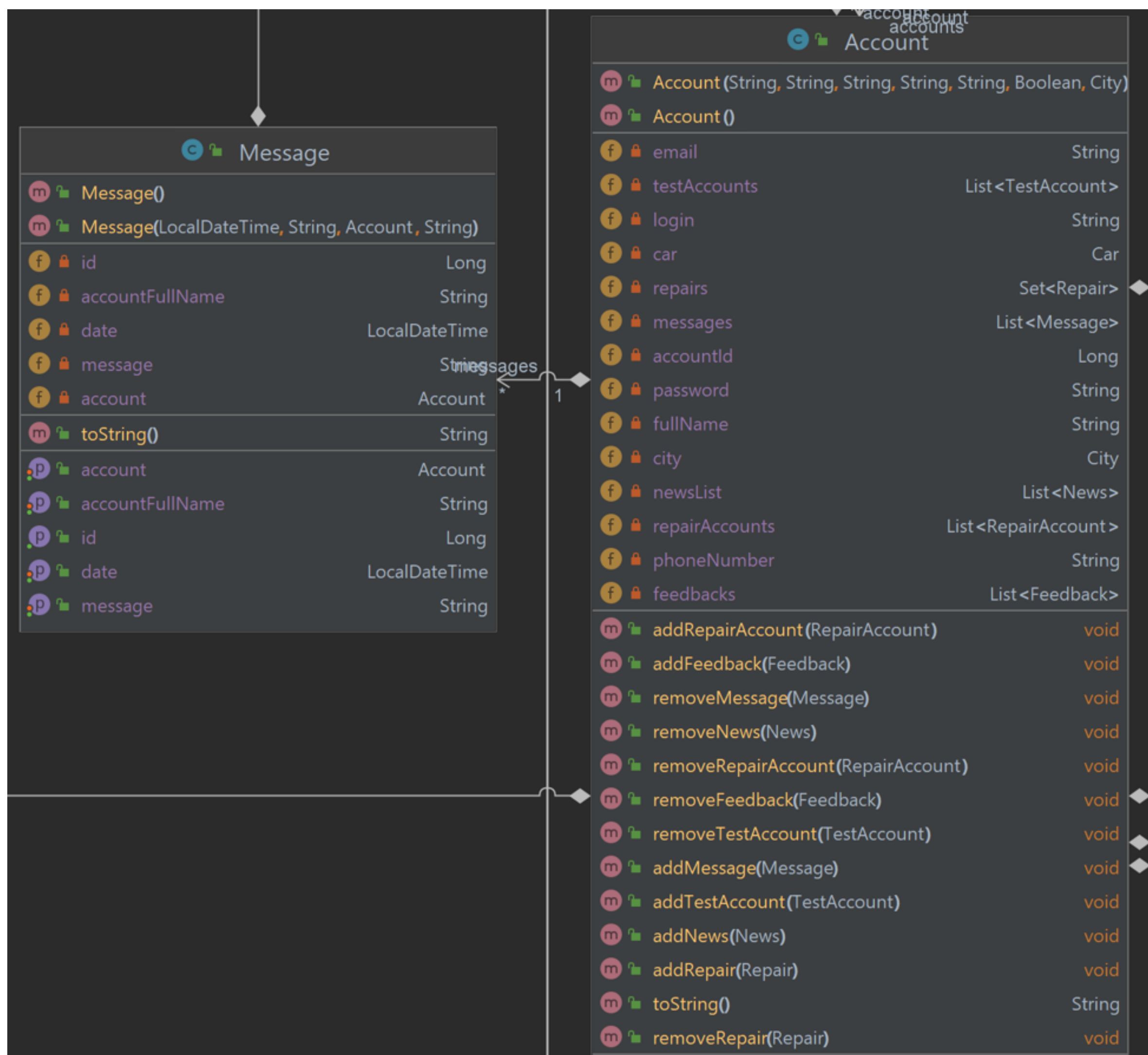

Рис. 5. Account-Message class diagram

Клас Message – повідомлення, наведений на попередній діаграмі (Рис. 5) та він має асоціацію "багато до одного" з класом Account.

Клас News – новини, містить наступні поля та асоціацію "багато до одного" з класом Account (Рис. 6).

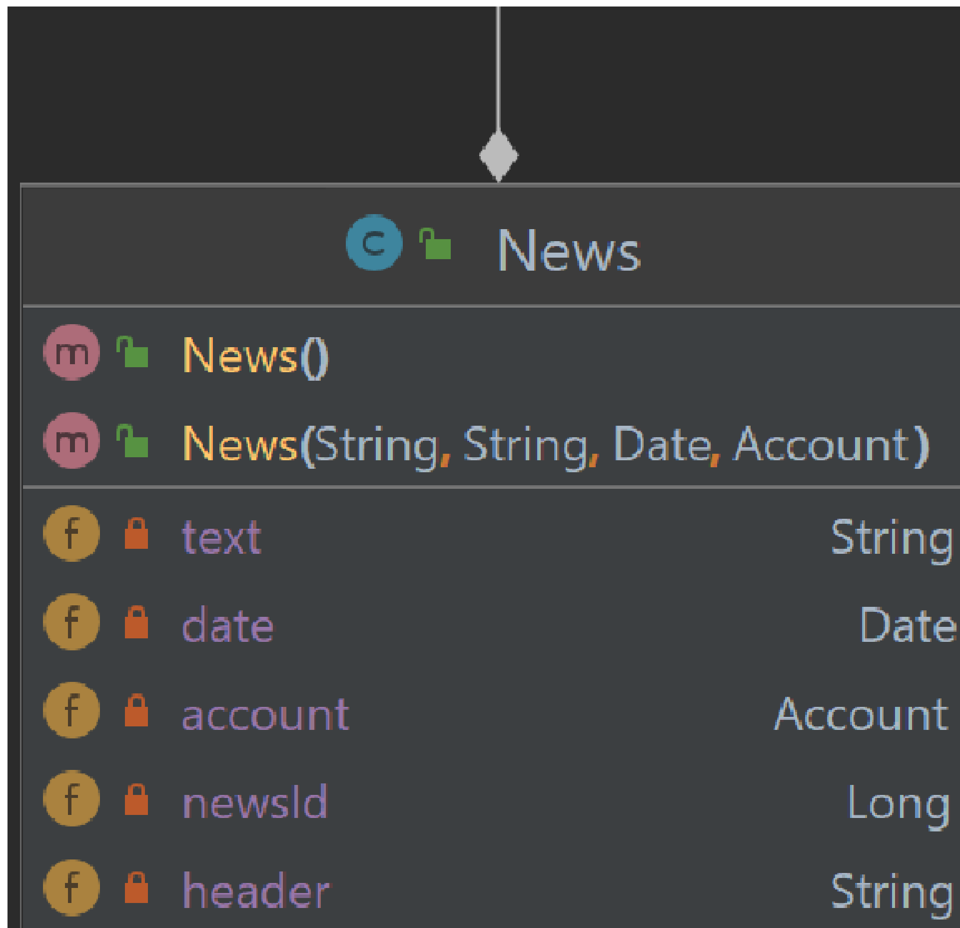

Рис. 6. News class diagram

Клас Car – автомобіль, містить наступні поля та асоціацію "один до одного" з класом Account (Рис. 7).

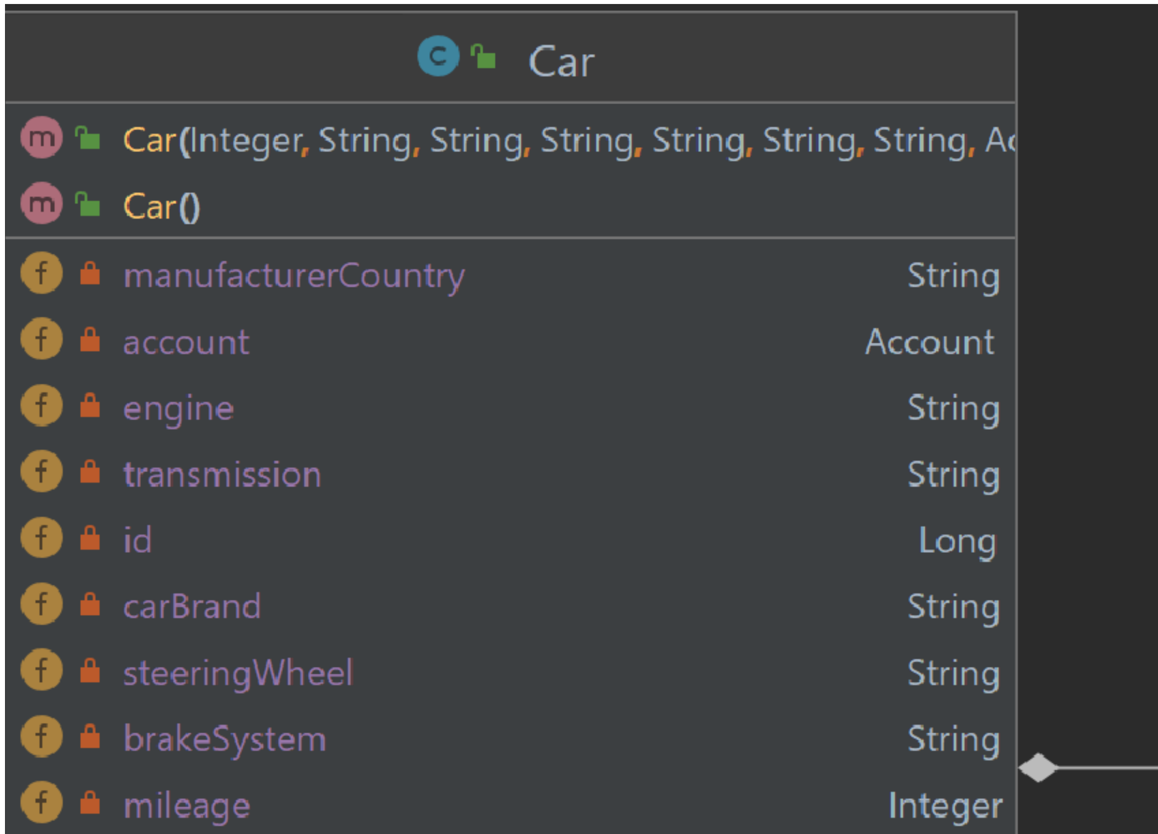

Рис. 7. Car class diagram

Клас Test – тест, містить наступні поля та асоціацію "багато до багатьох" з класом Account реалізовану через клас TestAccount (Рис.8).

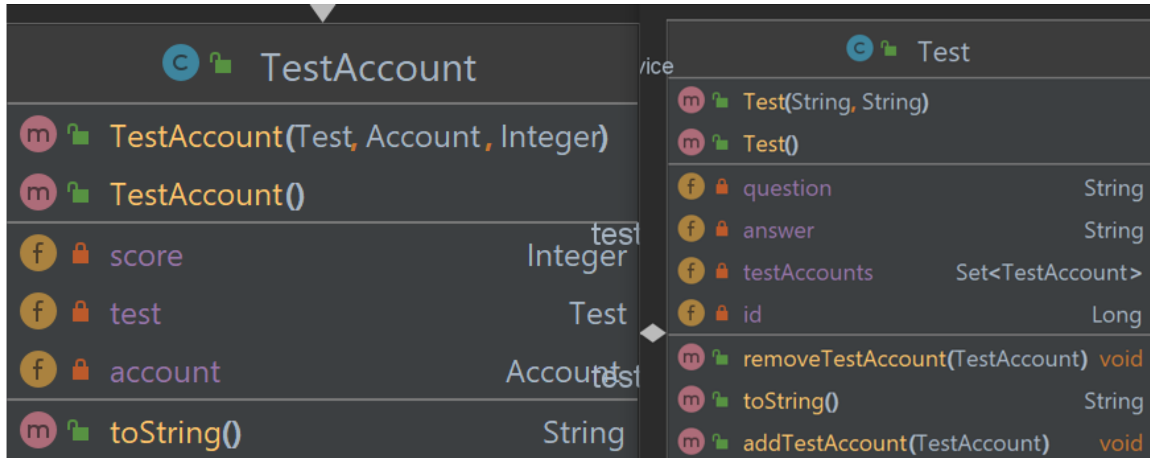

Рис 8. Test class diagram

Клас City – міста, містить наступні поля(Рис. 9).

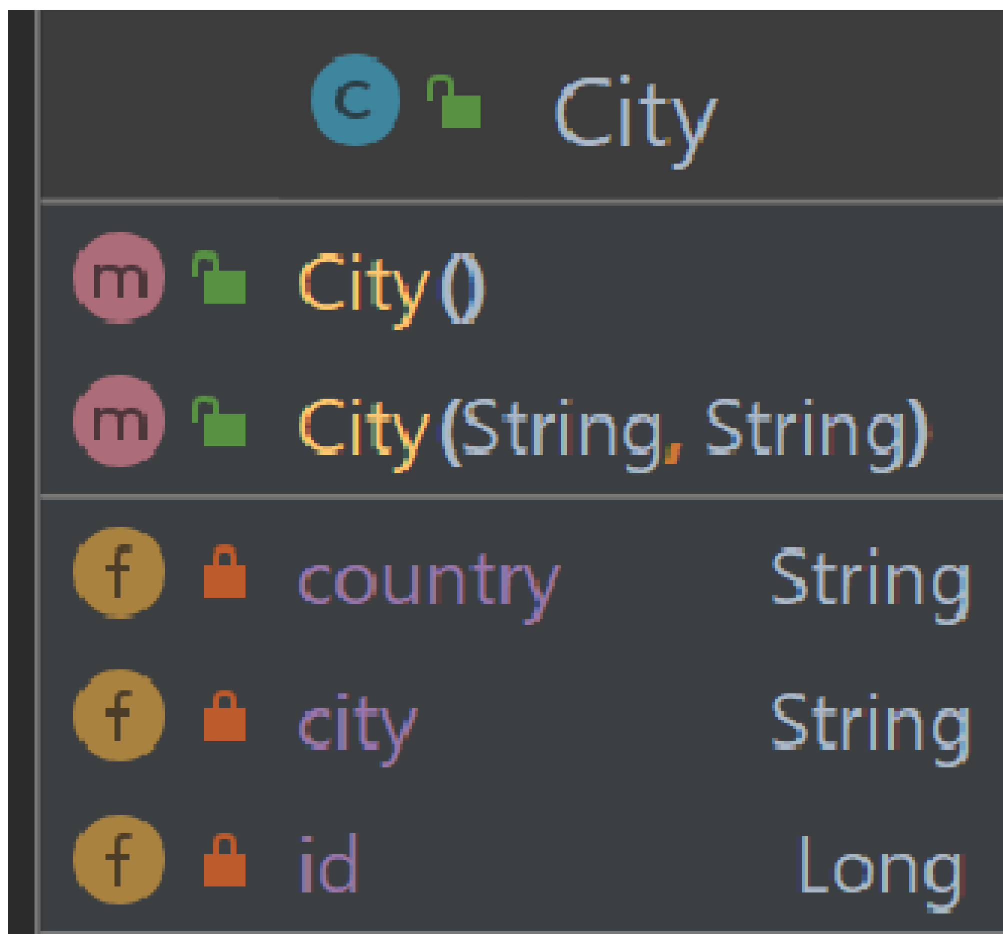

Рис. 9. City class diagram

Клас Repair – автосервіси, містить наступні поля та асоціацію "багато до багатьох" з класом Account реалізовану через клас RepairAccount, з City асоціацію "багато до одного" та з Feedback- відгук, "один до багатьох".

(Рис. 10)

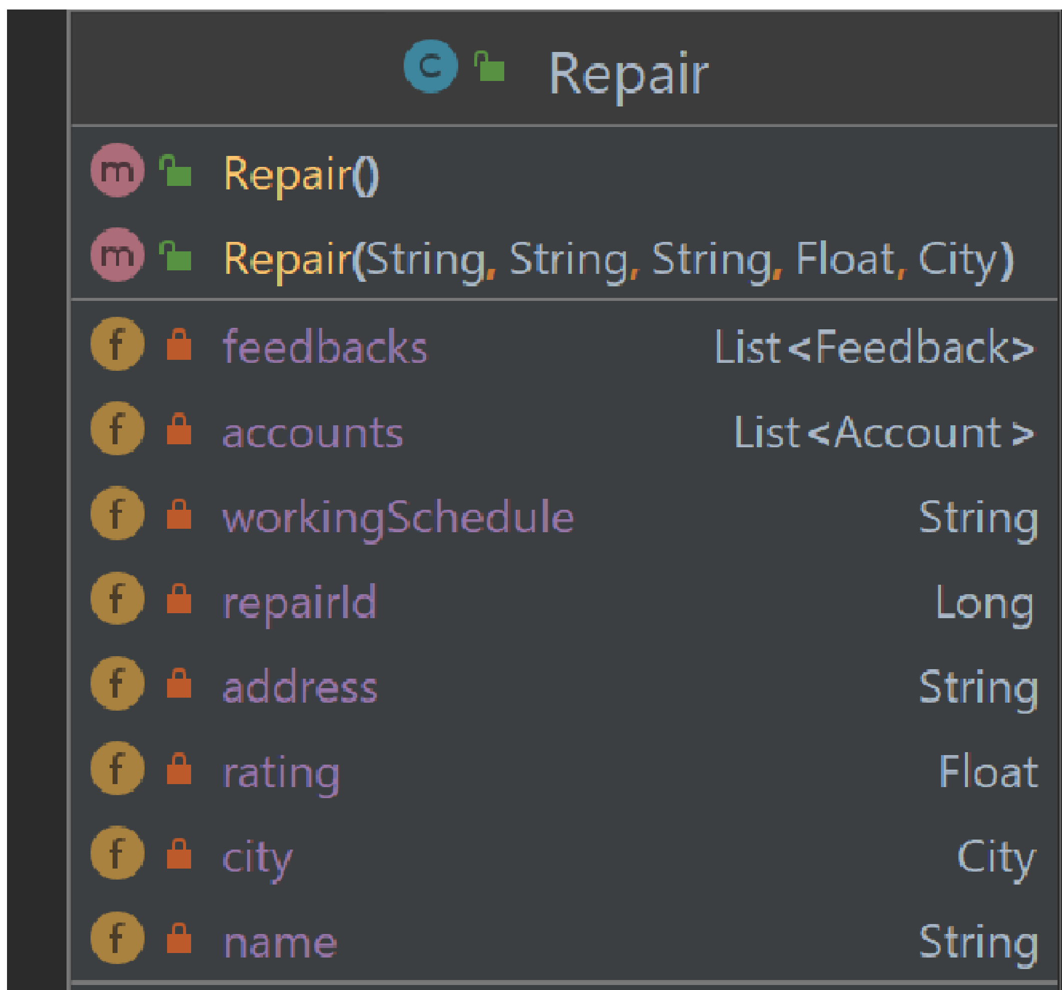

Рис. 10. Repair class diagram

Клас Feedback- відгук, містить наступні поля та асоціацію "багато до одного " з класом Account, з Repair асоціацію "багато до одного". (Рис. 11)

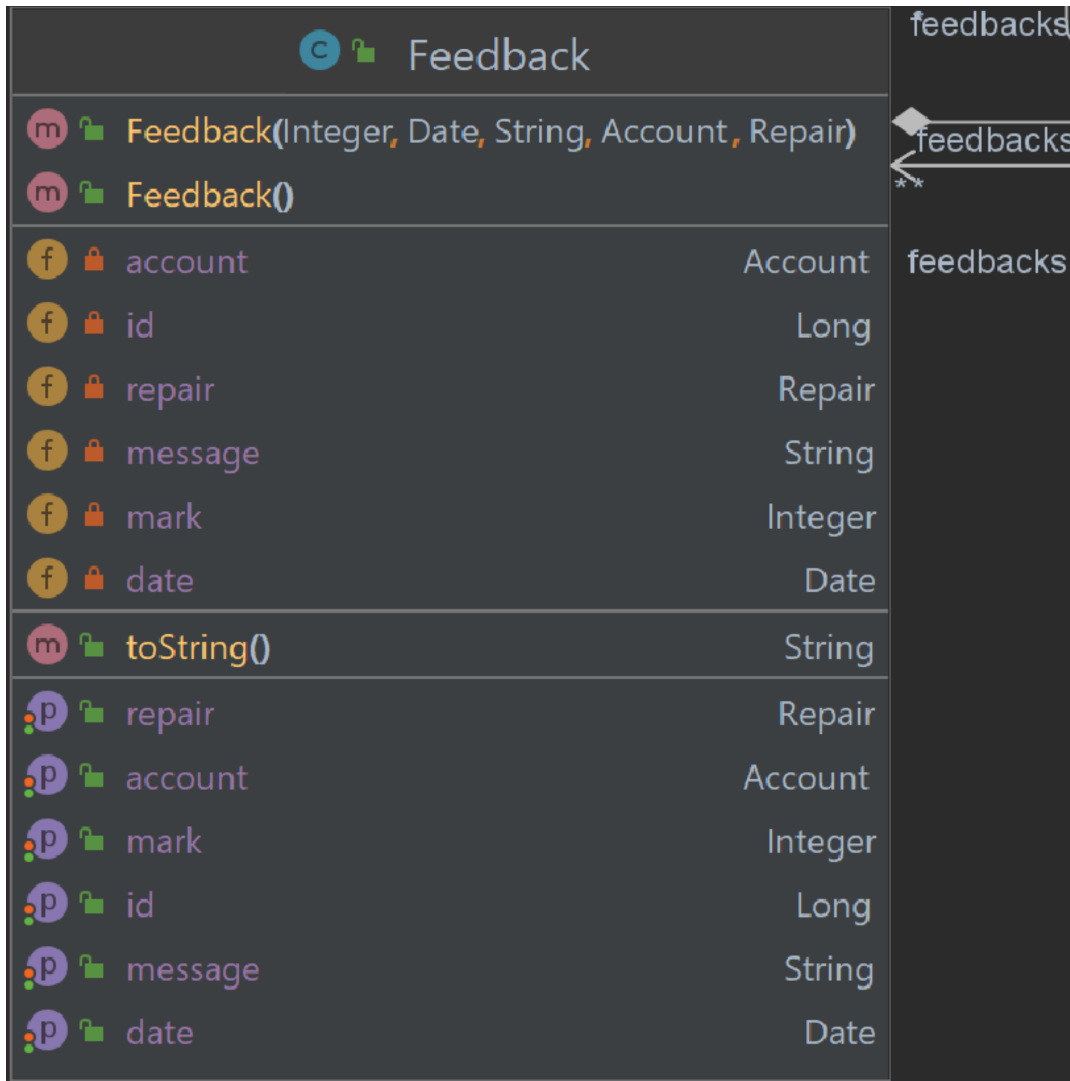

Рис. 11. Feedback class diagram

Вище було описано усі основні класи та їх зв’язки між собою, розглянуто як вони відносяться один до одного і яким типом зв’язків з’єднані.

3. РОЗРОБКА СЕРВЕРНОЇ ЧАСТИНИ ДОДАТКУ

3.1 Основні архітектурні паттерни та шаблони

Паттерн MVC (Model – View – Controller):

Паттерн MVC (Model-View-Controller) є одним з найпоширеніших архітектурних паттернів для розробки програмного забезпечення. Він використовується для розділення логіки програми на три основні компоненти: Модель (Model), Представлення (View) та Контролер (Controller). Кожен з цих компонентів виконує свою відповідальність і взаємодіє з іншими компонентами для забезпечення ефективної роботи програми.

Основні ролі та відповідності компонентів у паттерні MVC:

Модель (Model): Модель представляє бізнес-логіку, дані та стан програми. Вона відповідає за обробку даних, бізнес-правил і логіки, без прив'язки до способу відображення або взаємодії з користувачем. Модель може містити методи для отримання та зміни даних, а також виконання розрахунків або операцій над даними.

Представлення (View): Представлення відповідає за візуалізацію даних, відображення графічного інтерфейсу користувача та взаємодію з користувачем. Воно отримує дані з моделі і відображає їх у зрозумілій формі. Представлення може також реагувати на події користувача і сповіщати контролер про ці події.

Контролер (Controller): Контролер служить посередником між моделлю і представленням. Він обробляє події користувача, взаємодіє з представленням та моделлю для забезпечення обробки запитів та оновлення стану програми.

Контролер також відповідає за перетворення даних, отриманих від користувача або представлення, перед тим, як вони будуть передані до моделі для обробки.

Взаємодія між компонентами у паттерні MVC зазвичай відбувається за наступним сценарієм:

Користувач взаємодіє зі стороною представлення (наприклад, вводить дані або натискає кнопку на веб-сторінці).

Представлення передає введені дані або подію контролеру.

Контролер приймає дані або подію від представлення та виконує відповідну обробку. Він може звертатися до моделі для отримання даних або зміни їх стану.

Контролер відповідає на запит або подію шляхом оновлення моделі або відправки відповіді до представлення.

Модель змінює свій стан або повертає дані, що відповідають запиту контролера.

Контролер отримує відповідь від моделі та передає її до представлення.

Представлення отримує дані від контролера і відображає їх у відповідному форматі (наприклад, веб-сторінка з виведеними даними або зміненою стороною користувача).

Переваги паттерна MVC включають:

Розділення логіки програми на окремі компоненти, що полегшує розуміння та розвиток коду.

Повторне використання компонентів, оскільки модель, представлення та контролер можуть бути незалежно міняються або розширюються.

Забезпечення більшої гнучкості та масштабованості програми, оскільки зміни в одному компоненті не впливають на інші.

DTO (Data Transfer Object)- шаблон проектування:

DTO використовується для передачі даних між компонентами системи. Це простий контейнер, який містить поля для зберігання даних та може використовуватись для передачі цих даних через мережу або між різними шарами системи. Основна ідея застосування DTO полягає в тому, щоб зменшити кількість даних, що передаються між компонентами, та виключити непотрібну інформацію. DTO дозволяє сконцентруватись лише на потрібних полях та даних для певного контексту взаємодії.

Основні переваги використання DTO включають:

Зменшення навантаження мережі: DTO дозволяє передавати лише необхідні дані, що допомагає знизити обсяг передачі даних між клієнтом та сервером.

Використання DTO дозволяє забезпечити контроль над тим, які дані передаються між компонентами системи. Можна виключити чутливі дані або зашифрувати їх, щоб забезпечити безпеку під час передачі через мережу.

Розділення бізнес-логіки та представлення: DTO допомагає розділити бізнес-логіку системи від представлення даних. Класи DTO відображають лише необхідні поля та дані, які передаються між компонентами, і не містять логіки обробки.

Покращення продуктивності: Використання DTO може покращити продуктивність системи, оскільки дозволяє зменшити навантаження на сервер та оптимізувати передачу даних.

DTO можуть бути поділені на DTO для запиту (Request DTO) і DTO для відповіді (Response DTO) залежно від їхньої ролі у взаємодії між клієнтом і сервером.

Request DTO використовується для передачі даних з клієнта на сервер. Він містить поля, які вказують на дані, які клієнт хоче надіслати на сервер для виконання певного дії. Наприклад, при створенні нового користувача клієнт може відправити запит з Request DTO, який містить ім'я, електронну пошту та пароль нового користувача.

Response DTO, з іншого боку, використовується для передачі даних з сервера на клієнт у відповідь на запит. Він містить дані, які сервер повертає клієнту після обробки запиту. Наприклад, при запиті на отримання інформації про користувача сервер може повернути Response DTO, який містить ім'я, електронну пошту та інші дані про користувача.

Поділ на Request DTO і Response DTO допомагає забезпечити чітку сегрегацію даних, які передаються між клієнтом і сервером. Вони відображають роль і функцію кожного типу DTO в процесі взаємодії і полегшують розуміння та управління передачею даних у системі. Крім того, така сегрегація дозволяє гнучко керувати тим, які поля даних передаються в різних контекстах і запитах, забезпечуючи ефективну комунікацію між клієнтом і сервером.

Розглянемо AccountRequestDTO та MessageRequestDTO з діаграми класів (Рис. 12):

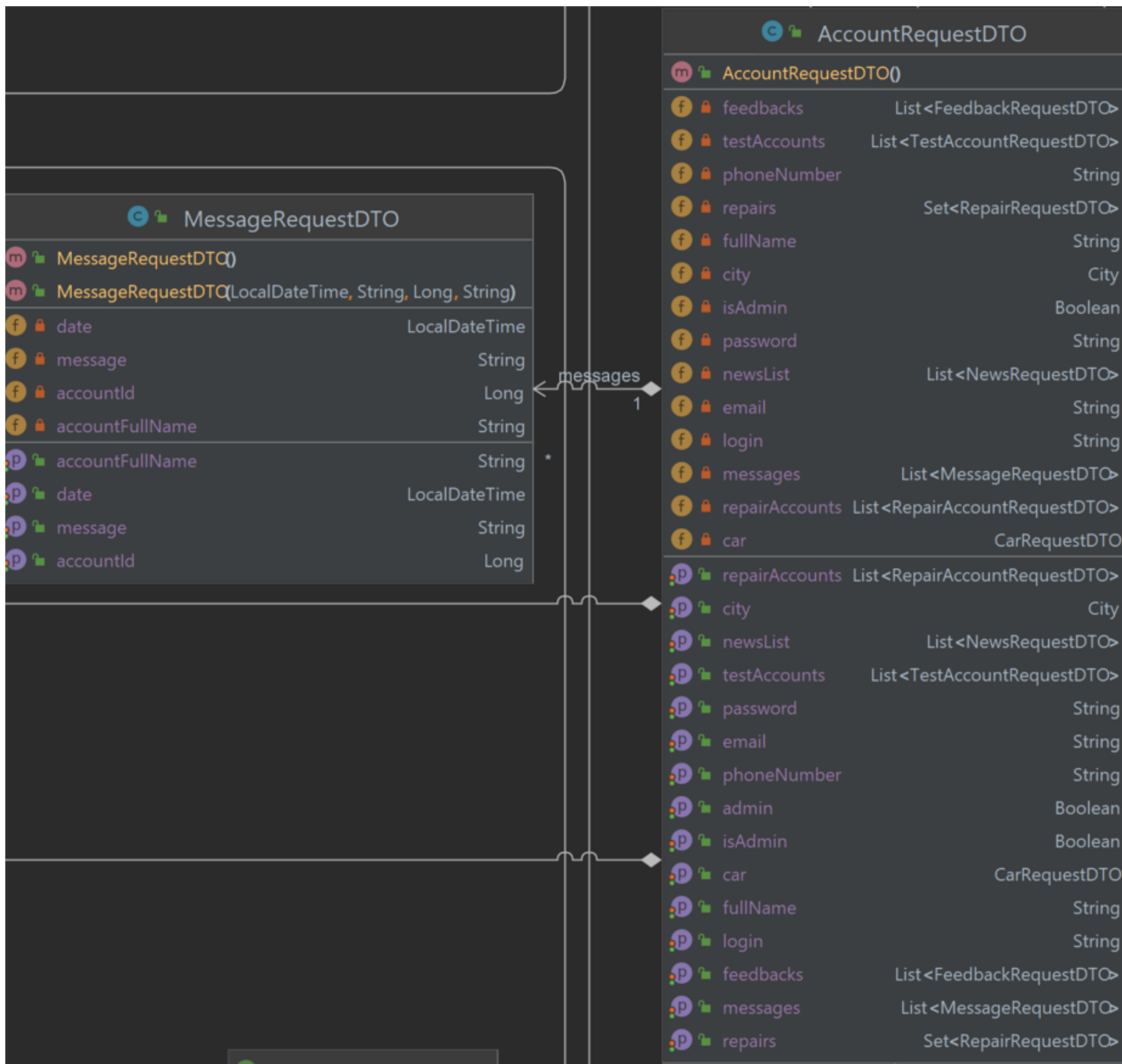

Рис. 12. Account/Message Request DTO class diagram

Приклад AccountResponseDTO та RepairResponseDTO з діаграми классів (Рис. 13):

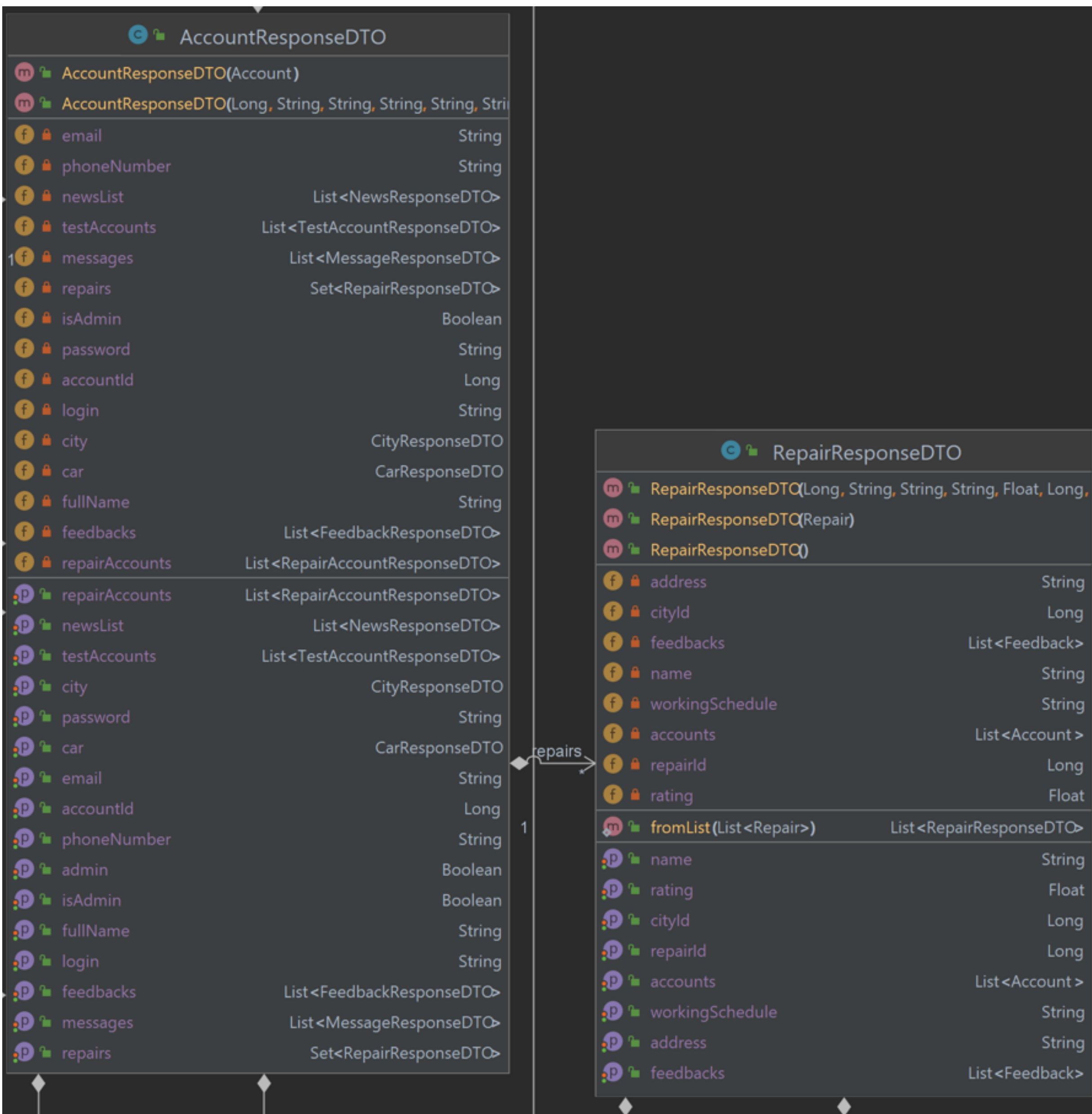

Рис. 13. Account/Repair ResponseDTO class diagram

3.2 Опис моделі

Для реалізації моделі було прийняте рішення використовувати JPA- сутності. JPA-сутності є основними будівельними блоками при роботі з об'єктно-реляційним відображенням (ORM) в Java.

JPA-сутності представляють таблиці бази даних і дозволяють здійснювати операції збереження, оновлення, видалення та вибірки даних з бази даних за допомогою об'єктно-орієнтованого підходу. Кожна JPA-сутність відповідає одній таблиці в базі даних і має відповідні поля, які відображаються на стовпці таблиці.

Ми розглянемо основні з них.

Для реалізації моделі даних за допомогою JPA-сутностей було визначено класи, які відповідають сутностям нашої моделі, використовуючи анотації JPA. Ці анотації дозволяють вказати маппінг між полями класу і стовпцями таблиці, встановити зв'язки між різними сутностями (такі як зв'язок один до одного, один до багатьох, багато до багатьох) та визначити інші аспекти поведінки сутностей.

JPA надає різні анотації для роботи з JPA-сутностями, такі як @Entity, @Table, @Column, @Id, @GeneratedValue та багато інших. Ці анотації дозволяють налаштовувати взаємодію між сутностями та базою даних.

Загалом, використання JPA-сутностей дозволяє легко і зручно працювати з моделлю даних у відображенні об'єктно-орієнтованого підходу, спрощує розробку і збереження даних в базі даних.

Розглянемо реалізацію сутності Account для прикладу:

Клас Account є сутністю JPA, оголошеною за допомогою анотації @Entity. Ця анотація вказує JPA, що клас представляє таблицю в базі даних (Рис. 14).

Анотація @Table(name = "account") вказує назву таблиці, з якою пов'язана сутність Account (Рис. 14).

Кожне поле в класі Account відповідає стовпцю таблиці, і це вказується за допомогою анотації @Column. Наприклад, поле fullName відображається на стовпець "fullName" в таблиці "account" (Рис. 14).

Анотація @Id позначає поле accountId як первинний ключ для сутності Account (Рис. 14).

Анотація @GeneratedValue(strategy = GenerationType.AUTO) вказує, що значення поля accountId буде генеруватись автоматично (Рис. 14).

```
@Entity
@Table(name = "account")
public class Account {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "account_id")
    private Long accountId;

    @Column(name = "fullName", nullable = false)
    private String fullName;
```

Рис. 14. Account class name and fields

Зв'язки між сутностями також були оголошені за допомогою анотацій. Наприклад, зв'язок багато-до-одного з сутністю City було визначено за допомогою анотації @ManyToOne і @JoinColumn(name = "city_id"). Це означає, що в таблиці "account" буде стовпець "city_id", який посилається на первинний ключ таблиці "city" (Рис. 15).

Зв'язок один-до-багатьох з сутностями News, Message, також був оголошений за допомогою анотації @OneToMany. За допомогою параметру mappedBy вказано, яке поле в інших сутностях використовується для зв'язку з об'єктом Account (Рис. 15).

Анотація @OneToMany має параметр cascade, який вказує, які операції повинні каскадно виконуватись на зв'язані сутності. Значення CascadeType.ALL означає, що всі операції (створення, оновлення, видалення) на сутності Account будуть каскадно виконуватись на зв'язані сутності, в даному випадку News, Message. Це зручно, оскільки не потрібно окремо викликати методи збереження або видалення на зв'язані сутності, а JPA автоматично робить це за нас при виконанні відповідної операції на сутності Account. Параметр orphanRemoval = true вказує, що коли сутність Account видаляється, всі пов'язані з нею сутності, які стали "сиротами" (тобто не мають більше жодного батьківського зв'язку), такі як News, Message, також будуть видалені. Це допомагає підтримувати консистентність бази даних і видаляти всі пов'язані записи, які більше не мають сенсу після видалення сутності Account. Таким чином, використання CascadeType.ALL та orphanRemoval = true дозволяє автоматично каскадно виконувати операції на зв'язані сутності та видаляти "сирот" при видаленні основної сутності, що спрощує роботу з моделлю даних (Рис. 15).

Анотація @ManyToMany вказує на зв'язок багато-до-багатьох між сутностями Account і Repair. Цей зв'язок реалізований через проміжну таблицю account_repair, яка використовується для зберігання пари значень ідентифікаторів account_id та repair_id. Анотація @JoinTable вказує на використання проміжної таблиці account_repair для збереження зв'язку між сутностями. name = "account_repair" вказує назву таблиці, joinColumns = { @JoinColumn(name = "account_id") } вказує, що колонка account_id в таблиці account_repair буде посилатися на первинний ключ id в таблиці account, і inverseJoinColumns = { @JoinColumn(name = "repair_id") } вказує, що колонка repair_id в таблиці account_repair буде посилатися на первинний ключ id в таблиці repair. Колекція repairs представлена типом Set<Repair>, що вказує на множинну зв'язаність між сутностями Account і Repair. У даному випадку використовується реалізація HashSet, яка забезпечує унікальність елементів у колекції. Це означає, що одна сутність Account може мати багато зв'язаних сутностей Repair, але кожен Repair буде унікальним для даної сутності Account (Рис. 15).

```
@ManyToOne
@JoinColumn(name = "city_id")
private City city;

@OneToMany(mappedBy = "account", cascade = CascadeType.ALL, orphanRemoval = true)
private List<News> newsList;

@OneToMany(mappedBy = "account", cascade = CascadeType.ALL, orphanRemoval = true)
private List<Message> messages;

@ManyToMany(cascade = { CascadeType.ALL })
@JoinTable(
        name = "account_repair",
        joinColumns = { @JoinColumn(name = "account_id") },
        inverseJoinColumns = { @JoinColumn(name = "repair_id") }
)
private Set<Repair> repairs = new HashSet<>();
```

Рис. 15. Account Relations

Перший конструктор: public Account(String fullName, String phoneNumber, String email, String login, String password, Boolean isAdmin, City city): Цей конструктор приймає параметри, які представляють поля сутності Account. Він використовується для створення об'єкту Account з встановленими значеннями для цих полів. Крім того, він ініціалізує зв'язані колекції newsList, messages, repairs, repairAccounts, feedbacks і testAccounts пустими списками або множинами (Рис. 16).

Другий конструктор без параметрів в JPA сутностях має кілька потенційних застосувань (Рис. 16):

Якщо ви використовуєте JPA для отримання даних з бази даних, JPA потребує створення конструктора без параметрів, щоб створити об'єкти сутностей під час виконання запитів. Це може включати вибірку даних з бази даних або

вставку нових записів. Деякі ORM-бібліотеки, такі як Hibernate, використовують конструктор без параметрів для створення проксі-об'єктів або при використанні лінивої ініціалізації. Це дозволяє зберегти пам'ять і час, не завантажуючи всі дані з бази даних, які можуть бути необхідні.

Ініціалізація колекцій в конструкторі без параметрів (другому конструкторі) може мати кілька причин (Рис. 16):

Зручність в роботі: Ініціалізація колекцій в конструкторі без параметрів дозволяє уникнути помилок та забезпечити належну ініціалізацію об'єкта без необхідності передавати параметри при його створенні.

Запобігання NullPointerException: Колекції в JPA-сутностях можуть бути використані в різних місцях коду, включаючи методи, які викликаються під час завантаження об'єкта з бази даних. Ініціалізація колекцій в конструкторі дозволяє гарантувати, що колекції завжди будуть не нульовими, що допомагає уникнути NullPointerException.

Забезпечення одноразової ініціалізації: Ініціалізація колекцій в конструкторі дозволяє гарантувати, що колекції створюються тільки один раз при створенні об'єкта і не можуть бути перезаписані в інших місцях коду, що забезпечує консистентність даних та уникнення непередбачуваних станів.

Загальною метою ініціалізації колекцій в конструкторі є забезпечення належного початкового стану об'єкта та зручного способу роботи з ним.

```
public Account(String fullName, String phoneNumber, String email, String login,
               String password, Boolean isAdmin, City city) {
    this.fullName = fullName;
    this.phoneNumber = phoneNumber;
    this.email = email;
    this.login = login;
    this.password = password;
    this.isAdmin = isAdmin;
    this.city = city;
    this.newsList = new ArrayList<>();
    this.messages = new ArrayList<>();
    this.repairs = new HashSet<>();
    this.repairAccounts = new ArrayList<>();
    this.feedbacks = new ArrayList<>();
    this.testAccounts = new ArrayList<>();
}

public Account() {
    this.newsList = new ArrayList<>();
    this.messages = new ArrayList<>();
    this.repairs = new HashSet<>();
    this.repairAccounts = new ArrayList<>();
    this.feedbacks = new ArrayList<>();
    this.testAccounts = new ArrayList<>();
}
```

Рис. 16. Account Constructors

В кінці описані наступні методи: геттери, сеттери та додавання, видалення елементів з колекцій.

Для зручності був перевизначений toString().

Інші основні JPA-сутності були створені з описаних класів схожим чином.

Розглянемо за приклад проміжну сутність для зв'язку багато-до-багатьох RepairAccount (Рис. 17).

Сутність RepairAccount є JPA-сутністю, що відображає таблицю repair_account у базі даних (Рис. 17).

Основні анотації, використані для визначення сутності (Рис. 17):

@Entity: Вказує, що клас є JPA-сутністю.

@Table(name = "repair_account"): Вказує назву таблиці в базі даних, до якої буде відображатися сутність.

Опис полів сутності:

@Id: Вказує, що поле id є первинним ключем сутності.

@GeneratedValue(strategy = GenerationType.AUTO): Вказує спосіб автоматичного генерування значення для поля id.

@Column(name = "id"): Вказує назву стовпця в базі даних, до якого буде відображатися поле id.

@ManyToOne: Вказує на відношення багато-до-одного між сутностями Repair і Account.

@JoinColumn(name = "repair_id"): Вказує назву стовпця в базі даних, яке використовується для зв'язку з сутністю Repair.

@JoinColumn(name = "account_id"): Вказує назву стовпця в базі даних, яке використовується для зв'язку з сутністю Account.

@Column(name = "date"): Вказує назву стовпця в базі даних, до якого буде відображатися поле date.

@Column(name = "repair_name"): Вказує назву стовпця в базі даних, до якого буде відображатися поле repairName.

Клас RepairAccount містить два конструктори, так як JPA-сутність повинна мати конструктор без параметрів (Рис. 17).

```
@Entity
@Table(name = "repair_account")
public class RepairAccount {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "id")
    private Long id;

    @ManyToOne
    @JoinColumn(name = "repair_id")
    private Repair repair;

    @ManyToOne
    @JoinColumn(name = "account_id")
    private Account account;

    @Column(name = "date")
    private Date date;

    @Column(name = "repair_name")
    private String repairName;

    public RepairAccount(Repair repair, Account account, Date date, String repairName) {
        this.repair = repair;
        this.account = account;
        this.date = date;
        this.repairName = repairName;
    }

    public RepairAccount() {
    }
```

Рис. 17. RepairAccount Entity

В кінці були описані наступні методи: геттери, сеттери та перевизначений toString().

Інші проміжні JPA-сутності були створені схожим чином.

3.3 Реалізація сервісів та контролерів

Спочатку давайте визначимо для чого нам сервіси: У Spring Boot сервіси є компонентами, які виконують бізнес-логіку в додатку і надають певні функції та операції. Вони є частиною архітектури Spring-додатків і використовуються для виконання операцій, таких як обробка запитів, взаємодія з базою даних, виконання складних бізнес-процесів та інші.

Основні особливості сервісів у Spring Boot:

У Spring Boot сервіси визначаються як Spring-компоненти, за допомогою анотації @Service. Це дозволяє Spring виявити сервіси під час компонування (dependency injection) і автоматично створити їх екземпляри. Сервіси виконують бізнес-логіку додатку. Вони можуть містити методи, які обробляють запити, виконують розрахунки, маніпулюють даними або взаємодіють з іншими компонентами.

Для створення сервісів попередньо було створено відповідні репозиторії:

Репозиторії у Spring Boot є компонентами, які використовуються для взаємодії з базою даних. Вони виконують операції збереження, оновлення, видалення та запиту даних і забезпечують абстракцію доступу до даних у високорівневому рівні.

Розглянемо приклад створеного інтерфейсу AccountRepository (Рис. 18):

У Spring Framework анотація @Repository використовується для позначення інтерфейсу як репозиторію. Репозиторій відповідає за доступ до даних і виконання операцій збереження, оновлення, видалення та запиту даних. AccountRepository є інтерфейсом репозиторію, який розширяє інтерфейс

JpaRepository. JpaRepository є частиною модуля Spring Data JPA і надає загальні операції доступу до даних.

Додатково до методу findByLogin, AccountRepository буде мати доступ до різних інших методів, наслідуваних від JpaRepository, таких як save, delete, findAll та інші, які дозволяють виконувати різні операції з об'єктами Account в базі даних.

Метод findByLogin в AccountRepository використовує Spring Data JPA для автоматичної генерації SQL-запиту на основі імені методу. Коли ви викликаєте цей метод і передаєте йому значення для поля login, Spring Data JPA автоматично створює SQL-запит для пошуку об'єкта Account зі співпадінням поля login. Запит буде виконаний на базі даних, пов'язаній з AccountRepository, і результат буде повернутий як об'єкт Optional<Account>.

```
@Repository
public interface AccountRepository extends JpaRepository<Account, Long> {
    Optional<Account> findByLogin(String login);
```

Рис. 18. AccountRepository

Далі було створено репозиторії для кожної JPA-сутності схожим чином.

Після цього розглянемо реалізацію TestService як приклад (Рис. 19):

Анотація @Service в Spring Framework вказує, що клас TestService є сервісним компонентом. Сервіси виконують бізнес-логіку додатку і зазвичай використовуються для обробки операцій над даними, спілкування з репозиторіями і виконання різноманітних завдань.

У даному випадку, TestService має дві залежності: TestRepository і TestAccountRepository. За допомогою анотації @Autowired, ці залежності ін'єктуються в конструктор TestService, що реалізує принцип ін'єкції залежностей.

Метод createTest створює новий об'єкт Test з заданими питанням і відповіддю. Далі, цей об'єкт Test зберігається в базі даних за допомогою методу save репозиторію testRepository. Після успішного збереження, повертається ідентифікатор (id) збереженого об'єкту Test.

```
@Service
public class TestService {
    private final TestRepository testRepository;
    private final TestAccountRepository testAccountRepository;

    @Autowired
    public TestService(TestRepository testRepository, TestAccountRepository testAccountRepository) {
        this.testRepository = testRepository;
        this.testAccountRepository = testAccountRepository;
    }

    public Long createTest(String question, String answer) {
        Test test = new Test(question, answer);
        Test savedTest = testRepository.save(test);
        return savedTest.getId();
    }
```

Рис. 19. Test Service

Розглянемо деякий метод цього сервісу с більш складною логікою (Рис 20).

Цей метод дозволяє додавати тестові облікові записи до існуючих тестів у системі.

Спочатку метод отримує testId і accountId в якості параметрів. Використовуючи testRepository, виконується пошук тесту за його ідентифікатором testId за допомогою методу findById. Результат знаходиться в Optional<Test> testOptional.

Використовуючи testOptional, метод перевіряє, чи існує тест. Якщо тест не знайдено, генерується виключення TestNotFoundException.

Використовуючи testAccountRepository, виконується пошук об'єкту TestAccount за ключем TestAccountKey(testId, accountId) за допомогою методу findById. Результат знаходиться в Optional<TestAccount> testAccountOptional.

Використовуючи testAccountOptional, метод перевіряє, чи існує об'єкт TestAccount. Якщо об'єкт не знайдено, генерується виключення TestAccountNotFoundException.

Після успішного отримання тесту і облікового запису, метод викликає метод addTestAccount об'єкту Test, передаючи обліковий запис.

Зміни в об'єкті Test зберігаються у базі даних за допомогою методу save репозиторія testRepository.

```
public void addTestAccount(Long testId, Long accountId) {
    Optional<Test> testOptional = testRepository.findById(testId);
    Test test = testOptional.orElseThrow(() -> new TestNotFoundException("Unable to find test with id " + testId));

    Optional<TestAccount> testAccountOptional = testAccountRepository.findById(new TestAccountKey(testId, accountId));
    TestAccount testAccount = testAccountOptional.orElseThrow(() ->
            new TestAccountNotFoundException("Unable to find test account with id " + accountId));

    test.addTestAccount(testAccount);
    testRepository.save(test);
}
```

Рис 20 TestService method example

Інші методи сервісу було реалізовано схожим чином.

Далі було реалізовано інші сервіси для JPA- сутностей, коли усі потрібні сервіси реалізовано- можемо розглянути контролери.

Контролери у сервісах використовуються для створення точок доступу (API) до функціональності, яку надають сервіси. Контролери приймають HTTP-запити від клієнтів і взаємодіють з сервісами для обробки цих запитів і повернення відповідей.

Основні причини використання контролерів у сервісах:

Контролери відповідають за обробку HTTP-запитів і забезпечення правильної маршрутизації. Вони відокремлюють цю логіку від сервісів, які виконують бізнес-логіку. Це дозволяє підтримувати чистоту коду і покращує зрозумілість та обслуговування проекту.

Контролери приймають HTTP-запити і виконують перетворення параметрів запиту, аутентифікацію та авторизацію, якщо це необхідно. Вони передають вхідні дані до сервісів для подальшої обробки.

Контролери займаються формуванням відповідей на HTTP-запити. Вони можуть обробляти результати, отримані від сервісів, і створювати відповіді у відповідному форматі (наприклад, JSON, XML) для надсилання клієнтам.

DTO (Data Transfer Object) використовуються в контролерах з метою передачі даних між клієнтом і сервером. Вони допомагають забезпечити чітку відокремленість між представленням даних на клієнтській стороні і моделями даних на серверній стороні.

Розглянемо для прикладу початок реалізації TestAccountController (Рис 21)

Даний контролер TestAccountController є компонентом у Spring-додатку, що відповідає за обробку HTTP-запитів, пов'язаних з тестами і користувачами.

Основні елементи контролера:

Анотація @RestController: Ця анотація вказує, що клас є контролером, який повертає результати відповідей як JSON або інші формати. Він поєднує в собі анотації @Controller і @ResponseBody.

Залежність TestAccountService: Це сервісний клас, який надає функціональність для обробки тестів і користувачів. Він ін'єктується в контролер через конструктор за допомогою анотації @Autowired.

Анотація @PostMapping: Ця анотація вказує, що метод обробляє POST-запит на шляху /tests/{testId}/accounts/{accountId}/score. Вхідні дані для методу передаються у тілі запиту (@RequestBody TestAccountRequestDTO testAccountRequestDTO), який є об'єктом типу TestAccountRequestDTO.

Метод addScore: Цей метод викликає сервісний метод addScore з переданими параметрами з об'єкту testAccountRequestDTO. Результатом є ціле число - оцінка (score), яке повертається у відповідь клієнту.

Анотація @GetMapping: Ця анотація вказує, що метод обробляє GET-запит на шляху /tests/{testId}/accounts/{accountId}/score. Значення {testId} і {accountId} замінюються на відповідні значення з URL. Метод отримує ці значення через анотації @PathVariable.

Метод getScore: Цей метод викликає сервісний метод getScore з переданими значеннями testId і accountId. Результатом є ціле число - оцінка (score), яке повертається у відповідь клієнту.

```
@RestController
public class TestAccountController {
    private final TestAccountService testAccountService;

    @Autowired
    public TestAccountController(TestAccountService testAccountService) {
        this.testAccountService = testAccountService;
    }

    @PostMapping("/tests/{testId}/accounts/{accountId}/score")
    public Integer addScore(@RequestBody TestAccountRequestDTO testAccountRequestDTO) {
        testAccountService.addScore(testAccountRequestDTO.getTestId(), testAccountRequestDTO.getAccountId(),
                testAccountRequestDTO.getScore());
        return testAccountRequestDTO.getScore();
    }

    @GetMapping("/tests/{testId}/accounts/{accountId}/score")
    public Integer getScore(@PathVariable Long testId, @PathVariable Long accountId) {
        return testAccountService.getScore(testId, accountId);
    }
}
```

Рис 21 TestAccountController

Розглянемо для прикладу декілька інших методів іншого класу TestController (Рис 22).

Метод updateTest є PUT-методом і має шлях /tests/{testId}. Він приймає параметр testId з певним значенням testId, яке передається в URL, і об'єкт TestRequestDTO з тілом запиту, що містить нові дані тесту для оновлення. Контролер передає ці параметри до сервісу TestService, який відповідає за оновлення тесту. Сервіс виконує логіку оновлення тесту з використанням переданих параметрів і зберігає оновлений тест у базі даних.

Метод deleteTestById є DELETE-методом і має шлях /tests/{testId}. Він приймає параметр testId з певним значенням testId, яке передається в URL. Контролер передає цей параметр до сервісу TestService, який відповідає за

видалення тесту за його ідентифікатором. Сервіс виконує відповідну логіку видалення тесту з використанням переданого ідентифікатора.

```
@PutMapping("/tests/{testId}")
public void updateTest(@PathVariable Long testId, @RequestBody TestRequestDTO testRequestDTO) {
    testService.updateTest(testId, testRequestDTO.getQuestion(), testRequestDTO.getAnswer());
}

@DeleteMapping("/tests/{testId}")
public void deleteTestById(@PathVariable Long testId) {
    testService.deleteById(testId);
}
```

Рис 22 TestController several methods

Інші методи для контролерів було реалізовано схожим чином.

Далі реалізовано контролери для решти сервісів.

3.4 Взаємодія моделі та контролера

Взаємодія моделі та контролера у Spring Boot може бути реалізована за допомогою наступних підходів:

Анотації @Controller і @RestController: Використовуючи ці анотації, можна позначити клас контролера або його методи, що відповідають за обробку HTTP-запитів. Для позначення контролерів було прийнято рішення використовувати саме анотацію @RestController.

Анотація @Service: Якщо модель має складну логіку або вимагає більшої обробки, було створено відповідні сервісні класи, в якому реалізована логіка моделі. Контролер може взаємодіяти з цим сервісом, викликаючи його методи.

Використання шаблону проектування "Репозиторій": Можна визначити інтерфейс репозиторію, що надає методи для доступу до даних моделі, та реалізувати його в класі, який взаємодіє з джерелом даних. Сервіси використовують відповідні репозиторії для реалізації логіки та операцій з моделлю.

Контролери реалізують у собі відповідні сервіси для операцій з моделлю та обробки HTTP-запитів.

При отриманні HTTP-запиту, контролер отримує дані від клієнта. Якщо запит містить тіло (наприклад, метод POST або PUT), контролер отримує ці дані з тіла запиту.

Для передачі даних від клієнта до контролера, використовується клас RequestDTO. Цей клас містить поля, що відповідають даним, які передаються в запиті. Контролер отримує об'єкт RequestDTO із вхідними даними,

використовуючи анотацію @RequestBody, яка вказує, що дані мають бути зчитані з тіла запиту і прив'язані до об'єкта RequestDTO.

Після отримання RequestDTO контролер передає його в сервісний шар, де відбувається обробка бізнес-логіки.

В сервісному шарі можуть бути виконані різні операції з отриманими даними, такі як пошук, створення, оновлення або видалення. Відповідно до виконаних операцій, сервіс може використовувати репозиторій для взаємодії з базою даних.

Після обробки бізнес-логіки, сервіс повертає результат у вигляді моделі даних.

Контролер отримує результат від сервісу, який може бути представлений у вигляді моделі даних. Для передачі даних клієнту, контролер використовує клас ResponseDTO. Він створює об'єкт ResponseDTO, заповнює його даними з отриманої моделі, а потім повертає об'єкт ResponseDTO як відповідь на запит.

Сутність (Entity): Сутність представляє об'єкт або концепцію, яка має важливе значення для додатку або системи. Вона може мати атрибути (поля), які описують її властивості, та може мати взаємозв'язки з іншими сутностями. Наприклад, сутність "Користувач" може мати атрибути, такі як ідентифікатор, ім'я, електронна пошта та пароль.

DTO для відповіді (ResponseDTO): DTO (Data Transfer Object) для відповіді використовується для передачі даних від сервера до клієнта. Це об'єкт, який містить лише ту інформацію, яка необхідна клієнту для відображення або виконання деяких дій. Наприклад, для сутності "Користувач" DTO для відповіді може містити ідентифікатор, ім'я та електронну пошту користувача та інше.

DTO для запиту (RequestDTO): DTO для запиту використовується для передачі даних від клієнта до сервера. Він містить інформацію, яка необхідна серверу для виконання певної дії або операції. Наприклад, для створення нового користувача DTO для запиту може містити ім'я, електронну пошту та пароль, які введені користувачем.

Ці сутності та DTO-класи використовуються для зручного представлення та передачі даних між компонентами програми, такими як контролери, сервіси та репозиторії. Вони допомагають забезпечити чітку структуру та розділення відповідальностей у системі.

3.5 Майбутні перспективи розвитку проекту

Проект розробки додатку для автомобілістів має значний потенціал для майбутнього розвитку та вдосконалення. Деякі перспективи розвитку проекту включають:

Розширення функціональності: Постійне розширення функціональних можливостей додатку дозволить користувачам отримувати більше корисної інформації та послуг. Наприклад, можна додати функції моніторингу та аналізу витрат палива, інтеграцію з системами навігації, побудову оптимальних маршрутів, додаткові сервіси підтримки та інше.

Розвиток спільноти користувачів: Створення активної спільноти користувачів додатку може сприяти обміну досвідом, порадами та рекомендаціями між автомобілістами. Можна впровадити функцію форумів, де користувачі зможуть обговорювати різні теми, задавати запитання та ділитися своїми знаннями.

Інтеграція зі сторонніми сервісами: Взаємодія з іншими сервісами та платформами, такими як онлайн-магазини запчастин, сервіси доставки, системи технічного обслуговування, може покращити зручність та функціонал додатку. Це дозволить користувачам замовляти запчастини, записуватися на сервісне обслуговування та використовувати інші пов'язані послуги безпосередньо з додатку.

Використання аналітики даних: Використання аналітики даних може забезпечити більш точні прогнози та рекомендації для користувачів.

Удосконалення безпеки та взаємодії: Додаток може продовжувати розвиватись у напрямку поліпшення безпеки автомобілістів та їх взаємодії з

дорожнім середовищем. Наприклад, додаток може надавати рекомендації щодо безпечної їзди, інформувати про дорожні пригоди або небезпечні ділянки доріг, а також попереджати про непередбачувані ситуації

Розробка додатку для автомобілістів, який надає інформацію про стан автомобіля, рекомендації щодо обслуговування, пошук сервісів, інформацію про відгуки, новини та взаємодію з іншими водіями, має значний потенціал і актуальність у сучасному автомобільному світі. Постійний розвиток технологій, зростання інтересу до електромобілів та автономного руху, а також збільшення кількості автомобілів на дорогах створюють нові виклики та можливості для таких додатків. Отже, розробка та подальший розвиток додатку для автомобілістів має великий потенціал і актуальність, забезпечуючи зручність, безпеку та ефективність використання автомобілів. Шляхи розширення платформи, покращення функціональності та інтерфейсу, а також удосконалення безпеки та взаємодії з іншими системами відкривають нові перспективи розвитку проекту.

ВИСНОВКИ

Результатом моєї дипломної роботи стало розроблення мобільного додатку для автомобілістів. Постановка задачі, пояснення актуальності теми у сьогоденні та аналітика майбутніх перспектив проекту. Проектування необхідної архітектури для створення додатку. Реалізація серверної частини додатку мовою програмування Java, за допомогою фреймворку Spring Boot, та СУБД PostgreSQL. Детальне пояснення використаних паттернів та шаблонів. Покрокова реалізація з поясненням JPA- сутностей, інтерфейсів, сервісів та контролерів. Демонстрація роботи серверної частини додатку та правильної обробки даних.

В ході розробки було проведено роботу з паттерном веб програмування, та шаблоном передачі даних. Також з фреймворком Spring Boot та створенням необхідних компонентів за його допомогою. Продемонстрована взаємодія СУБД PostgreSQL з додатком, та правильність відображання та отримання інформації.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Посібник із проектування реляційних баз даних: веб-сайт. URL: https://metanit.com/sql/tutorial/ (дата звернення 02.03.23)
2. Документація до PostgreSQL 12.15: веб-сайт. URL: https://postgrespro.ru/docs/postgresql/12 (дата звернення 04.03.23)
3. Посібник з мови програмування Java. URL: https://metanit.com/java/tutorial/ (дата звернення 05.03.23)
4. JDK 10 Documentation: веб-сайт. URL: https://docs.oracle.com/javase/10/ (дата звернення 08.03.23)
5. Паттерни для новачків: MVC vs MVP vs MVVM: веб-сайт. URL: https://habr.com/ru/articles/215605/ (дата звернення 14.03.23)
6. Побудова REST сервісів за допомогою Spring: веб-сайт. URL: https://spring-projects.ru/guides/tutorials-bookmarks/ (дата звернення 25.04.23)