Форма № 24

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Комп'ютерних наук і технологій
(повне найменування факультету)

Комп'ютерні системи та мережі
(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)
бакалаврський
(ступінь вищої освіти)

на тему: РОЗРОБКА КОМП'ЮТЕРНОЇ СИСТЕМИ АВТОСАЛОНУ

Виконав(ла): студент(ка) 4 курсу,
групи КНТз-513сп

Спеціальності
123 Комп'ютерна інженерія
(код і найменування спеціальності)

Освітня програма (спеціалізація)
Комп'ютерна інженерія
(назва освітньої програми (спеціалізації)

МАКАРЯН А.М.
(ПРІЗВИЩЕ та ініціали)

Керівник СКРУПСЬКИЙ С.Ю.
(ПРІЗВИЩЕ та ініціали)
Рецензент ХАРИТОНОВ О.Б.
(ПРІЗВИЩЕ та ініціали)

2026

Форма № 25

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій
Кафедра комп'ютерних систем та мереж
Ступінь вищої освіти бакалаврський
Спеціальність 123 комп'ютерна інженерія
(код і найменування)
Освітня програма (спеціалізація) комп'ютерна інженерія
(назва освітньої програми (спеціалізації)

ЗАТВЕРДЖУЮ
Завідувач кафедри КУДЕРМЕТОВ Р.К.
_____
« » _____ 2026 року

З А В Д А Н Н Я
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

МАКАРЯНА Альберта Мікаеловича
(ПРІЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Розробка комп'ютерної системи автосалону
керівник проєкту (роботи) к.т.н., доцент, СКРУПСЬКИЙ С.Ю.,
(науковий ступінь, вчене звання, ПРІЗВИЩЕ, ім'я, по батькові)
затверджені наказом закладу вищої освіти від « 14 » квітня 2026 року № 160
2. Строк подання студентом проєкту (роботи) 01.06.2026 р.
3. Вихідні дані до проєкту (роботи) опис предметної області, технології розробки клієнтської та серверної частини

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1) Аналіз технологій розробки клієнт-серверних систем;
2) Розробка серверної частини системи;
3) Розробка клієнтської частини системи;
4) Випробування системи.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів)
Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4	СКРУПСЬКИЙ С.Ю.		
Нормоконтроль	ДЬЯЧУК Т.С.		

7. Дата видачі завдання «14» квітня 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Аналіз предметної області та аналогів	до 18.04.2026	
2	Визначення та аналіз вимог до системи	до 21.04.2026	
3	Проєктування бази даних системи	до 25.04.2026	
4	Проєктування архітектури сервера	до 28.04.2026	
5	Розробка серверної частини	до 10.05.2026	
6	Проєктування клієнтської частини	до 14.05.2026	
7	Розробка користувацького інтерфейсу	до 22.05.2026	
8	Випробування комп’ютерної системи	до 24.05.2026	
9	Оформлення пояснювальної записки	до 26.05.2026	
10	Проходження нормоконтролю	до 30.05.2026	
11	Перевірка на наявність академічного плагіату	до 01.06.2026	
12	Проходження рецензування	до 02.06.2026	

Студент(ка) _____ Альберт МАКАРЯН
(підпис) (Ім’я ПРІЗВИЩЕ)

Керівник проєкту (роботи) _____ Степан СКРУПСЬКИЙ
(підпис) (Ім’я ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра: 91 с., 13 рис., 2 табл., 3 дод., 20 джерел.

РОЗПОДІЛЕНА КОМП’ЮТЕРНА СИСТЕМА, ANDROID, КАТАЛОГ АВТОМОБІЛІВ, JAVA, GLIDE, POSTGRESQL, SOCKET, THREADS

Предмет розробки – клієнт-серверна комп’ютерна система для перегляду асортименту автомобілів.

Мета роботи – підвищення ефективності доступу користувачів до інформації про автомобільні моделі та комплектації.

Новизна роботи – поєднання Android-клієнта на мові Java та серверного застосунку із сокетною взаємодією без використання важких веб-фреймворків для перегляду каталогу автомобілів.

Практична цінність роботи полягає у можливості використання отриманих рішень як основи для легких мобільних каталогів у сфері автомобільного ритейлу.

У межах роботи виконано аналіз предметної області та існуючих рішень, сформовано вимоги до системи й обґрунтовано вибір клієнт-серверної архітектури. Реалізовано серверну частину на мові Java із сокетною взаємодією та передаванням даних у форматі JSON, а також клієнтську частину у вигляді Android-застосунку з фрагментною навігацією, асинхронним завантаженням даних і підтримкою відкриття зовнішніх PDF-прайсів.

Проведено випробування користувацького інтерфейсу та аналіз продуктивності системи з використанням інструментів Android Studio. Отримані результати показали стабільну роботу застосунку, прийнятне споживання оперативної пам’яті і мережевого трафіку, а також швидке завантаження інформації через мобільне підключення LTE.

ЗМІСТ

ВСТУП

Сучасний етап розвитку інформаційних технологій характеризується активним впровадженням мобільних рішень у сфери, що раніше переважно орієнтувалися на веб-застосунки або офлайн-каталоги. Зростання обсягів даних, вимоги до швидкодії та очікування користувачів щодо простоти й інтуїтивності інтерфейсу зумовлюють необхідність розробки спеціалізованих мобільних клієнт-серверних систем, адаптованих до обмежених ресурсів мобільних пристроїв і нестабільних мережевих умов. Актуальність даної дипломної роботи зумовлена зростаючою потребою в мобільних комп'ютерних системах, що забезпечують швидкий та зручний доступ до структурованих даних у клієнт-серверному середовищі. У сфері автомобільного ритейлу мобільні застосунки дозволяють підвищити ефективність взаємодії з користувачами, скоротити час пошуку інформації про моделі та комплектації, а також зменшити залежність від важких веб-інтерфейсів.

Предметом розробки є клієнт-серверна комп'ютерна система для перегляду асортименту автомобілів. Метою роботи є підвищення ефективності доступу користувачів до інформації про автомобільні моделі та комплектації.

У першому розділі виконано аналіз предметної області та існуючих рішень, визначено функціональні та нефункціональні вимоги до системи. Розглянуто особливості клієнт-серверної взаємодії в мобільних застосунках, обґрунтовано вибір архітектури, технологій та інструментів розробки, а також сформовано загальні вимоги до структури даних і користувацького інтерфейсу.

У другому розділі описано розробку серверної частини системи. Наведено логіку обробки клієнтських запитів, організацію мережевої взаємодії на основі сокетів, формат передавання даних у вигляді JSON, а також структуру моделей даних. Окрему увагу приділено проектуванню бази даних комп'ютерної системи.

У третьому розділі розглянуто реалізацію клієнтської частини у вигляді Android-застосунку. Описано структуру застосунку, взаємодію між фрагментами,

механізм асинхронного завантаження даних з сервера та їх відображення в інтерфейсі користувача. Детально розглянуто роботу основних фрагментів: вітального екрану, каруселі моделей та екрану детальної інформації про автомобіль.

У четвертому розділі проведено випробування розробленої системи. Описано результати випробування користувацького інтерфейсу, перевірку коректності навігації та функціональності відкриття зовнішніх ресурсів. Також виконано аналіз продуктивності застосунку із використанням стандартних інструментів Android, наведено показники споживання ресурсів та зроблено висновки щодо ефективності й доцільності використання розробленого рішення.

Новизна роботи полягає у розробці клієнт-серверної системи, що поєднує Android-клієнт на мові Java та серверний застосунок із сокетною взаємодією без використання важких веб-фреймворків. Практична цінність роботи полягає у можливості використання отриманих рішень як основи для легких мобільних каталогів у сфері автомобільного ритейлу.

1 АНАЛІЗ ТЕХНОЛОГІЙ РОЗРОБКИ КЛІЄНТ-СЕРВЕРНИХ СИСТЕМ

1.1 Призначення та функціональні вимоги до системи

У сучасній діяльності автосалонів основним каналом надання інформації потенційним клієнтам залишаються офіційні вебсайти. Через них користувач отримує доступ до переліку моделей, технічних характеристик, описів комплектацій, зображень автомобілів та додаткових матеріалів. Практика використання вебсайтів як базового інструменту взаємодії є типовою для більшості відомих автомобільних брендів, зокрема Volkswagen, Skoda, Audi та Toyota [1–4]. Такі ресурси зазвичай реалізовані у вигляді багатосторінкових інформаційних порталів із складною структурою навігації, великою кількістю розділів і допоміжних сервісів.

Аналіз офіційних вебсайтів зазначених брендів показує, що вони орієнтовані на універсальну аудиторію та поєднують одразу кілька цілей: інформування про модельний ряд, маркетингове просування, підтримку дилерської мережі та залучення клієнтів до придбання автомобіля. У результаті користувач, який має просту мету - ознайомитися з асортиментом авто та їхніми комплектаціями, змушений виконувати значну кількість переходів між сторінками, адаптуватися до складної навігації та працювати з великим обсягом другорядної інформації.

Незважаючи на наявність адаптивного дизайну, мобільні версії вебсайтів автосалонів не завжди забезпечують оптимальний користувацький досвід. Це зумовлено обмеженнями браузерного середовища, залежністю від швидкості мережі, а також необхідністю відображення складних інтерфейсних елементів на екранах з невеликою діагоналлю. У результаті знижується зручність перегляду каталогу моделей, зображень і текстових описів, що є критичним саме на етапі первинного вибору автомобіля.

Ефективність інформаційної системи значною мірою визначається відповідністю архітектурних та інтерфейсних рішень реальним сценаріям

використання. Сучасні підходи до проєктування клієнтських систем підкреслюють доцільність створення спеціалізованих мобільних застосунків, які забезпечують швидкий доступ до ключового функціоналу, мінімізують навантаження на користувача та дозволяють ефективно працювати з мультимедійним контентом [5].

Мобільний застосунок у контексті автосалону може розглядатися як альтернатива або доповнення до вебсайту, орієнтоване виключно на ознайомлення з асортиментом автомобілів. На відміну від вебресурсу, мобільний застосунок дозволяє реалізувати спрощену та цільову навігацію, оптимізовану під сенсорне керування, а також забезпечує стабільніший користувацький досвід незалежно від особливостей браузера. Крім того, застосунок створює передумови для подальшого розширення функціональності без зміни архітектури системи [6].

Призначенням розроблюваної комп'ютерної системи є надання клієнтам автосалону зручного мобільного інструменту для перегляду асортименту автомобілів, їхніх основних характеристик і ознайомлення з детальними описами комплектацій у форматі PDF. Система орієнтована на інформаційне ознайомлення клієнта з асортиментом автосалону та не охоплює процеси продажу, сервісного обслуговування або взаємодії з менеджерами автосалону. Такий підхід дозволяє зосередитися на якості подання контенту та швидкості доступу до нього.

Функціональні вимоги до клієнтського мобільного застосунку. Клієнтська частина системи повинна бути реалізована у вигляді мобільного застосунку для портативних пристроїв та забезпечувати запуск і коректну роботу на типовому обладнанні користувача. Після запуску застосунку користувачеві повинен відображатися стартовий екран із інформаційним привітанням автосалону та елементом керування для переходу до перегляду асортименту автомобілів. Стартовий екран виконує роль початкової точки взаємодії з системою та не передбачає введення будь-яких персональних даних користувача. Перехід до наступного екрана ініціює отримання даних з серверної частини системи та підготовку інтерфейсу для відображення переліку моделей.

Функціональні вимоги до представлення асортименту автомобілів. Система

повинна забезпечувати відображення переліку моделей автомобілів однієї марки у зручному для користувача візуальному форматі. Для цього клієнтський застосунок повинен підтримувати карусельне представлення моделей із можливістю послідовного перегляду. Для кожної моделі мають відображатися зображення автомобіля, назва моделі та базова ціна у форматі «від N грн». Навігація між моделями повинна здійснюватися за допомогою зрозумілих елементів керування, що дозволяють перемикатися між позиціями без переходу на окремі сторінки. Всі дані для формування каруселі повинні надходити з серверної частини та не зберігатися у клієнті як постійний контент.

Функціональні вимоги до перегляду інформації про окрему модель. При виборі конкретної моделі автомобіля система повинна забезпечувати відкриття окремого екрана з розширеною інформацією. На цьому екрані користувач повинен мати можливість ознайомитися з детальним текстовим описом моделі, який включає загальну характеристику автомобіля, основні особливості та стислий опис комплектацій. Текстовий опис має завантажуватися з серверної частини системи та відображатися у зручному для читання форматі. Завантаження та відображення інформації повинні виконуватися таким чином, щоб не порушувати стабільність роботи інтерфейсу та не блокувати взаємодію користувача із застосунком.

Функціональні вимоги до роботи з додатковими матеріалами. Система повинна надавати користувачеві можливість отримати доступ до повної інформації про комплектації автомобіля у вигляді PDF-документа. Для кожної моделі автомобіля серверна частина повинна зберігати та передавати клієнтському застосунку посилання на відповідний PDF-документ, який розміщується на зовнішньому ресурсі офіційного імпортера відповідної марки автомобілів в Україну. Клієнтський застосунок повинен забезпечувати відкриття такого документа за посиланням із використанням стандартних засобів пристрою користувача. Завантаження та перегляд PDF-документів не повинні вимагати додаткових дій з боку адміністратора або користувача в межах самого застосунку.

Функціональні вимоги до клієнт-серверної взаємодії. Система повинна

забезпечувати обмін даними між клієнтською та серверною частинами у режимі мережевої взаємодії. При зверненні клієнта до сервера має формуватися запит на отримання повного набору інформації щодо моделей автомобілів однієї марки. Серверна частина повинна обробляти цей запит та повертати структурований набір даних, що включає назви моделей, ціни, текстові описи, посилання на зображення та PDF-документи. Архітектура взаємодії повинна бути спроєктована таким чином, щоб мінімізувати кількість мережевих звернень і спростити логіку клієнтського застосунку.

Функціональні вимоги до серверної частини системи. Серверна частина повинна забезпечувати централізоване зберігання та обробку інформації про асортимент автомобілів. Сервер повинен приймати запити від клієнтських застосунків, виконувати обробку отриманих даних та формувати відповіді відповідно до визначеного формату. Серверна логіка має бути організована таким чином, щоб підтримувати одночасну роботу з кількома клієнтами та забезпечувати коректне формування відповідей незалежно від кількості активних підключень. Усі зміни контенту повинні відбуватися виключно на серверній стороні.

Функціональні вимоги до адміністрування системи. Система повинна передбачати можливість адміністрування серверної частини без використання графічного інтерфейсу. Адміністратор повинен мати можливість змінювати текстові описи моделей автомобілів, базові ціни та посилання на додаткові матеріали. Адміністративні дії повинні виконуватися через спеціальний режим роботи серверного застосунку та не бути доступними для клієнтських користувачів. Такий підхід забезпечує розмежування ролей та зменшує ризик несанкціонованих змін даних.

Функціональні вимоги до розширюваності системи. Функціональна структура системи повинна передбачати можливість подальшого розширення без зміни базових сценаріїв взаємодії користувача з клієнтським застосунком. Зокрема, система має підтримувати потенційне додавання кількох марок автомобілів із збереженням принципу отримання повного переліку моделей однієї

марки за один запит. Це забезпечує адаптивність системи до майбутніх вимог і спрощує супровід та розвиток програмного рішення.

1.2 Нефункціональні вимоги до системи

Сформулюємо нефункціональні вимоги до клієнт-серверної системи автосалону. Вимоги до продуктивності та швидкодії. Система повинна забезпечувати прийнятний час відгуку при типовому сценарії використання, коли клієнтський мобільний застосунок виконує запит на отримання повного переліку моделей однієї марки з усіма необхідними атрибутами. Час формування відповіді серверною частиною має бути настільки малим, щоб користувач не сприймав взаємодію із системою як повільну або нестабільну. На рівні клієнта завантаження та обробка даних повинні виконуватися асинхронно, без блокування користувацького інтерфейсу, що відповідає сучасним рекомендаціям до побудови мобільних застосунків [7]. Архітектурні рішення мають враховувати потенційне зростання обсягу даних при додаванні нових марок і моделей без суттєвого погіршення швидкодії.

Вимоги до надійності та стабільності роботи. Система повинна зберігати працездатність у разі часткових збоїв, зокрема тимчасової недоступності мережі або зовнішніх ресурсів, на яких розміщені PDF-документи. У таких випадках клієнтський застосунок повинен коректно інформувати користувача про неможливість отримання даних, не завершуючи роботу аварійно. Серверна частина повинна забезпечувати коректну обробку помилкових або неповних запитів та не допускати пошкодження або втрати даних. Забезпечення стабільної поведінки системи розглядається як одна з ключових якісних характеристик комп’ютерних систем, що працюють у режимі постійної мережевої взаємодії [8].

Вимоги до безпеки та захисту даних. Система повинна забезпечувати розмежування доступу між клієнтською та адміністративною частинами.

Адміністративні операції з редагування даних про моделі автомобілів мають бути доступні лише авторизованим користувачам серверної частини. Облікові дані адміністраторів повинні зберігатися у захищеному вигляді з використанням криптографічних методів хешування. Клієнтський застосунок не повинен містити жодної інформації, що дозволяє виконувати адміністративні дії. Передавання даних між компонентами системи має здійснюватися у структурованому форматі, що спрощує перевірку коректності даних і знижує ризик помилок обробки [9].

Вимоги до масштабованості та розширюваності. Архітектура системи повинна передбачати можливість подальшого розширення функціоналу без необхідності повної переробки клієнтської або серверної частин. Зокрема, система має підтримувати додавання нових марок автомобілів, збільшення кількості моделей та розширення обсягу текстових описів і допоміжних матеріалів. При цьому базовий принцип взаємодії клієнта з сервером повинен залишатися незмінним. Такий підхід відповідає сучасним архітектурним принципам проєктування програмних систем і дозволяє зменшити вартість супроводу та розвитку рішення.

Вимоги до сумісності та переносимості. Клієнтський застосунок повинен коректно працювати на мобільних пристроях з різними технічними характеристиками та розмірами екранів. Інтерфейс користувача має адаптуватися до особливостей пристрою та забезпечувати однакову логіку взаємодії незалежно від апаратної платформи. Серверна частина повинна бути незалежною від конкретного середовища розгортання, що дозволяє запускати її на різних операційних системах без зміни програмного коду. Забезпечення переносимості є важливою нефункціональною вимогою для клієнт-серверних комп’ютерних систем [10].

Вимоги до зручності використання та підтримуваності. Інтерфейс мобільного застосунку повинен бути інтуїтивно зрозумілим і орієнтованим на сценарій швидкого ознайомлення з асортиментом автомобілів. Користувач повинен мати можливість виконувати основні дії з мінімальною кількістю кроків. Кодова база клієнтської та серверної частин має бути структурованою та

зрозумілою для подальшого супроводу і модифікації. Дотримання принципів модульності, розділення відповідальностей і чіткої архітектурної організації спрощує внесення змін і знижує ризик появи помилок у процесі розвитку системи [11].

1.3 Вибір технологій реалізації системи

Для реалізації клієнт-серверного мобільного застосунку автосалону було розглянуто декілька підходів до організації обміну даними між клієнтом та сервером, зокрема нативні сокети (Socket/ServerSocket) [12], віддалені виклики методів (RMI / RPC) та REST API [13] на основі HTTP/JSON [14]. Основною метою було забезпечити максимальну продуктивність, керованість системою, стабільність роботи, а також готовність до розширення і тривалої підтримки.

Використання нативних сокетів у Java дозволяє організувати прямий двонаправлений канал обміну байтами між клієнтом та сервером без накладних витрат на проміжні протоколи. Передача даних у вигляді структурованих JSON-об’єктів через Gson [15] забезпечує швидку серіалізацію та десеріалізацію. На сервері використання потоків (Threads) дозволяє обслуговувати одночасні підключення, а клієнтський мобільний застосунок може працювати асинхронно з Handler [16], що гарантує інтерактивність інтерфейсу навіть при обробці великих обсягів даних, таких як текстові описи моделей, URL-адреси зображень та посилання на PDF-документи. Використання PrintWriter та Scanner спрощує організацію обміну рядками, логування та тестування системи у консолі, що особливо зручно для мобільного клієнта з обмеженими ресурсами [12].

RMI та RPC дозволяють викликати методи на віддаленому сервері, автоматично серіалізуючи Java-об’єкти. Такий підхід зручний для внутрішніх Java-систем, але накладає додаткові затримки через обробку складних об’єктів, що може значно зменшити швидкодію при передачі переліків моделей з описами і

мультимедійними ресурсами. Крім того, RMI обмежений екосистемою Java, що ускладнює майбутнє розширення системи на інші платформи, зокрема мобільні пристрої з різними ОС. Налаштування RMI потребує додаткових кроків, таких як реєстрація об’єктів у RMI-Registry і обробка RemoteException, що підвищує складність супроводу та знижує керованість системою [12].

REST API на основі HTTP/JSON забезпечує платформонезалежну взаємодію та зручну інтеграцію з різними клієнтами, але має значні накладні витрати через відкриття HTTP-з’єднання, формування заголовків і обробку статус-кодів [13]. Для мобільного клієнта, який обмежений ресурсами та має вимоги до швидкодії, це може призводити до затримок у відображенні інформації. Крім того, REST-запити зазвичай обробляються синхронно, що ускладнює багатопотокову обробку даних без додаткових бібліотек і збільшує навантаження на мобільний пристрій.

Порівняння альтернативних технологій для реалізації системи наведено в таблиці 1.1.

Таблиця 1.1 – Порівнянні технологій для реалізації системи

Критерій	Socket/Threads/Gson	RMI/RPC	REST/HTTP API
Час відгуку	мінімальний	середній	вищий
Контроль потоків даних	повний	обмежений	обмежений
Простота розширення	висока	середня	висока
Залежність від технологій	низька	висока (тільки Java)	середня (HTTP)
Підтримка багатопотоковості	проста	складна	середня
Готовність до мобільного клієнта	висока	середня	середня
Гнучкість у форматах даних	повна	часткова	часткова
Підготовленість до майбутніх змін	висока	середня	висока

Час відгуку є критичною характеристикою для мобільного застосунку, оскільки користувач очікує швидкого завантаження моделей автомобілів і

відображення їхніх описів та мультимедійних ресурсів. У цьому аспекті сокети демонструють мінімальні затримки, оскільки обмін даними здійснюється без проміжних протоколів та додаткових накладних витрат на серіалізацію та десеріалізацію складних об’єктів. Використання RMI/RPC, хоча і забезпечує автоматизовану серіалізацію об’єктів, додає значні затримки при передачі масивів даних або великих об’єктів, що зменшує загальну швидкодію системи. REST API на основі HTTP/JSON має ще більші накладні витрати, оскільки кожен запит передбачає відкриття TCP-з’єднання, формування HTTP-заголовків та обробку статус-кодів, що робить відгук менш оперативним порівняно з сокетами.

Контроль потоків даних є ще одним важливим фактором. Сокети у поєднанні з потоками (Threads) забезпечують повний контроль над обробкою запитів: сервер може одночасно обслуговувати декілька клієнтів, а клієнтський мобільний застосунок може виконувати асинхронне завантаження даних без блокування користувацького інтерфейсу. RMI/RPC надають частковий контроль, але через автоматичну обробку викликів методів важко керувати паралельною обробкою та ресурсами сервера. REST API дозволяє багатопоточність лише за допомогою додаткових бібліотек або асинхронних обгорток, що ускладнює реалізацію для мобільного пристрою з обмеженими ресурсами.

Простота масштабування та розширюваність системи також впливає на вибір технології. Сокети забезпечують гнучкість: легко додати підтримку нових марок автомобілів, мультимедійних ресурсів або кількох одночасних клієнтів, просто організувавши додаткові потоки та структуровані повідомлення у вигляді JSON. RMI/RPC складніше масштабувати через необхідність реєстрації об’єктів у RMI-Registry і обмеження Java-екосистеми. REST API, хоча й добре підходить для веб-сервісів, для мобільного застосунку невеликого масштабу додає непотрібні накладні витрати та не дає повного контролю над потоками та обсягом переданих даних.

Залежність від технологій і гнучкість форматів даних також мають вирішальне значення. Сокети не прив’язані до конкретного протоколу або бібліотеки, а формат даних (JSON через Gson) можна легко адаптувати або

змінити для підтримки майбутніх потреб. RMI обмежений Java-клієнтами і створює складнощі при інтеграції з іншими платформами, зокрема мобільними ОС. REST API, будучи платформонезалежним, має більшу гнучкість для майбутніх розширень, але накладні витрати знижують його продуктивність.

Фактор мобільного клієнта є особливо критичним. Мобільні пристрої мають обмежені ресурси (пам’ять, процесор, енергоспоживання), тому мінімізація накладних витрат на передачу даних та обробку є ключовою вимогою. Сокети дозволяють реалізувати легкий, ефективний протокол обміну даними без зайвого навантаження, забезпечуючи швидку роботу застосунку навіть при великій кількості моделей та мультимедійного контенту. RMI та REST, навпаки, збільшують навантаження на процесор і мережу мобільного клієнта, що може призвести до помітних затримок у користувацькому інтерфейсі та підвищеного енергоспоживання.

У підсумку, детальний аналіз порівняльних характеристик показує, що використання нативних сокетів є ефективним рішенням для даної клієнт-серверної системи. Сокети забезпечують максимальну продуктивність і мінімальний час відгуку, повний контроль над потоками даних, легкість масштабування та розширення функціоналу, низьку залежність від зовнішніх технологій і готовність до тривалої підтримки. Крім того, така архітектура є бажаною для мобільного клієнта з обмеженими ресурсами, дозволяючи забезпечити інтерактивний, стабільний та керований користувацький досвід.

1.4 Вибір бази даних

Для реалізації клієнт-серверної системи автосалону необхідно обрати систему керування базами даних, яка забезпечуватиме надійне зберігання структурованої інформації, підтримку транзакцій, цілісність даних та можливість подальшого розширення функціоналу. У загальному вигляді в базі даних

передбачається зберігання описів моделей автомобілів, цін у форматі «від N грн», а також посилань на PDF-документи з прайс-листами, що розміщені на зовнішніх ресурсах імпортера. Детальний склад і структура даних розглядатимуться у підрозділі 2.3, тому на цьому етапі доцільно зосередитися саме на виборі типу та конкретної СКБД.

На першому етапі доцільно розглянути вибір між реляційними та нереляційними базами даних. Нереляційні (NoSQL) СКБД орієнтовані переважно на зберігання неструктурованих або напівструктурованих даних, високу швидкість запису та горизонтальне масштабування. Такі системи ефективні у сценаріях із динамічною схемою або великими потоками подієвих даних. Проте для задачі автосалону характерна чітко визначена структура інформації, наявність логічних зв’язків між сутностями та вимоги до забезпечення цілісності й узгодженості даних. У цьому випадку реляційний підхід є більш доцільним, оскільки він дозволяє використовувати обмеження цілісності, транзакції та стандартизовану мову SQL для формування вибірок і агрегованих результатів [9].

Реляційні СКБД забезпечують механізми керування даними, контролю цілісності, беззбитковості та підтримку транзакцій з різними рівнями ізоляції, а також ефективну інтеграцію з серверними застосунками. Для клієнт-серверної системи, у якій сервер за один запит передає клієнту повний набір даних по моделях однієї марки, реляційна база даних дозволяє формувати цілісні результати та гарантувати коректність і узгодженість інформації. З огляду на це, для даної роботи було обрано саме клас реляційних систем керування базами даних.

Серед поширених реляційних СКБД доцільно розглянути MariaDB/MySQL, Microsoft SQL Server та PostgreSQL. MariaDB і MySQL є популярними рішеннями, які широко використовуються у веб-застосунках і відзначаються простотою розгортання та адміністрування. Вони добре підходять для типових операцій доступу до даних, однак мають певні обмеження щодо розширених можливостей роботи з транзакціями, типами даних та відповідності стандартам SQL у порівнянні з більш функціонально насиченими системами. Найкраще себе

розкривають у зв’язці з PHP [17].

Microsoft SQL Server є потужною промисловою СКБД з розвиненими інструментами адміністрування та оптимізації. Водночас її використання у кросплатформному Java-проєкті може бути обмежене ліцензійними умовами та залежністю від екосистеми Microsoft. Це знижує технологічну незалежність системи та ускладнює довгострокову підтримку без додаткових витрат, що є небажаним для навчального та демонстраційного проєкту.

PostgreSQL є повнофункціональною відкритою реляційною СКБД, яка поєднує високу надійність, відповідність стандартам SQL та широкі можливості розширення. Вона підтримує складні запити, транзакції, індексацію та різноманітні типи даних, що робить її придатною для зберігання структурованої інформації про моделі автомобілів, їх описи, ціни та пов’язані посилання. Важливою перевагою PostgreSQL є кросплатформність і відсутність ліцензійних обмежень, що дозволяє використовувати її як у навчальних, так і у промислових проєктах [18].

Для взаємодії серверного застосунку з обраною СКБД доцільно використати стандартний механізм доступу до баз даних у Java - JDBC (Java Database Connectivity). JDBC є універсальним API, який забезпечує виконання SQL-запитів, обробку результатів і керування транзакціями незалежно від конкретної реалізації СКБД. PostgreSQL надає офіційний JDBC-драйвер, що забезпечує стабільну і продуктивну взаємодію серверного застосунку з базою даних та добре інтегрується з класичною клієнт-серверною архітектурою [19].

З урахуванням вимог до надійності, цілісності даних, кросплатформності та технологічної незалежності, а також можливості стандартної взаємодії через JDBC, для реалізації серверної частини комп’ютерної системи автосалону доцільно обрати PostgreSQL як основну систему керування базами даних. Такий вибір створює стабільну основу для подальшого проєктування структури бази даних і реалізації серверної логіки.

1.5 Висновки по розділу

У першому розділі бакалаврської роботи було виконано комплексний аналіз підходів до розробки клієнт-серверних комп’ютерних систем та обґрунтовано основні архітектурні рішення, необхідні для реалізації мобільного застосунку автосалону. Розгляд сучасних підходів дозволив сформувати цілісне уявлення про вимоги до системи, її функціональне призначення та технологічну основу.

У результаті аналізу предметної області визначено, що основною задачею системи є надання клієнту зручного мобільного доступу до інформації про асортимент автомобілів без використання веббраузера. Обґрунтовано доцільність використання мобільного застосунку як альтернативи традиційним вебсайтам автосалонів, що дозволяє підвищити зручність користування, швидкодію доступу до контенту та створити основу для подальшого функціонального розширення системи.

На підставі аналізу технологій клієнт-серверної взаємодії обґрунтовано вибір прямої мережевої взаємодії між клієнтом і сервером. Використання нативних сокетів дозволяє мінімізувати накладні витрати на обмін даними, забезпечити повний контроль над потоками передачі інформації та досягти високої продуктивності, що є критичним для мобільного клієнта з обмеженими ресурсами. Обраний підхід також забезпечує технологічну незалежність системи та готовність до довгострокової підтримки.

У межах вибору засобів зберігання даних було обґрунтовано доцільність використання реляційної системи керування базами даних. Реляційна модель забезпечує цілісність і узгодженість структурованої інформації, підтримку транзакцій та можливість формування складних вибірок. Порівняльний аналіз поширених реляційних СКБД дозволив визначити PostgreSQL як оптимальне рішення з огляду на відкритість, кросплатформність, відповідність стандартам SQL та відсутність ліцензійних обмежень. Для взаємодії серверного застосунку з базою даних передбачено використання стандартного інтерфейсу JDBC, що

забезпечує уніфікований доступ до даних і спрощує супровід системи.

Таким чином, у першому розділі було сформовано концептуальну архітектуру клієнт-серверної системи автосалону (рис. 1.1), визначено ключові вимоги та обґрунтовано вибір основних технологій реалізації. Прийняті архітектурні рішення створюють основу для подальшого проєктування та реалізації серверної і клієнтської частин системи.

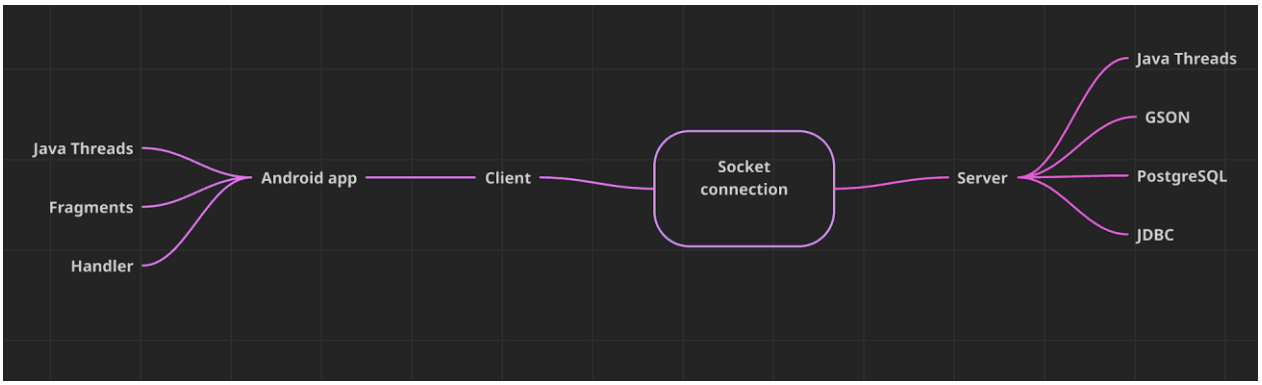


Рисунок 1.1 - Концептуальна архітектура клієнт-серверної системи автосалону

2 РОЗРОБКА СЕРВЕРНОЇ ЧАСТИНИ СИСТЕМИ

2.1 Проєктування серверної архітектури

Серверна частина системи проєктується як модульний консольний Java-застосунок із чітким розмежуванням відповідальностей між окремими класами. Такий підхід забезпечує керованість коду, спрощує супровід, тестування та подальше розширення функціональності без порушення існуючої логіки. Архітектура сервера орієнтована на одночасну роботу з декількома клієнтами, стабільну обробку мережевих з’єднань та централізований доступ до бази даних.

Серверний проєкт CarDealershipServer буде консольним Java-застосунком із двома пакетами: edu.cardealership.controller та edu.cardealership.model (рис. 2.1).

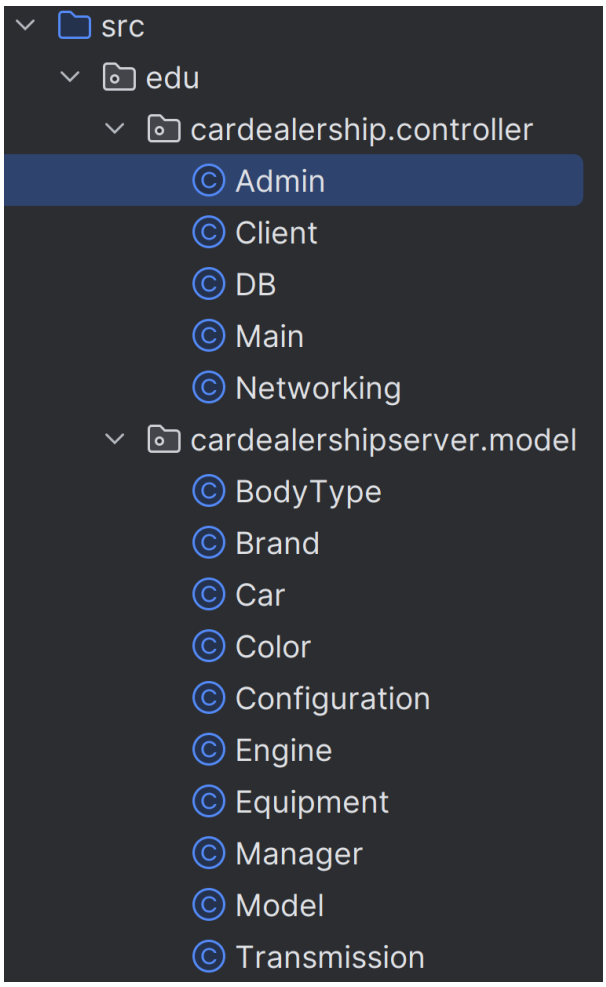


Рисунок 2.1 – Стуктура серверного проєкту

Пакет edu.cardealership.controller містить усі класи, що реалізують керування виконанням застосунку, мережеву взаємодію, доступ до бази даних та адміністративні операції.

Клас Main виконує роль точки входу в застосунок. У ньому ініціалізуються базові налаштування сервера: параметри мережевого з’єднання, шлях до конфігураційного файлу, створення екземпляра модуля доступу до бази даних і запуск мережевого прослуховування. Main координує запуск модуля Networking та, за необхідності, модуля Admin, а також відповідає за коректне завершення роботи сервера, звільнення ресурсів і логування критичних подій.

Клас Networking відповідає за організацію мережевого рівня. У ньому створюється ServerSocket, який слухає визначений порт і перебуває в циклі очікування підключень клієнтів. При кожному новому з’єднанні Networking створює екземпляр класу Client і запускає його в окремому потоці виконання.

Такий підхід дозволяє серверу обслуговувати декілька клієнтів паралельно, ізольовуючи їхню взаємодію та запобігаючи блокуванню всього сервера через одного клієнта.

Клас Client реалізує логіку обслуговування одного клієнтського підключення. Він відповідає за приймання запитів від мобільного застосунку, їх інтерпретацію та формування відповідей. Client викликає методи класу DB для отримання з бази даних інформації про автомобілі певної марки. Результатом таких викликів є колекції об’єктів класу Car. Отримані об’єкти серіалізуються у JSON та передаються клієнту по мережі. Client також контролює життєвий цикл сесії, обробляє помилки введення-виведення та коректно завершує з’єднання у разі розриву зв’язку.

Клас DB інкапсулює всю логіку взаємодії з базою даних. Він містить методи для вибірки даних про автомобілі, а також методи для оновлення цих даних при виконанні адміністративних операцій. DB повертає результати у вигляді об’єктів класу Car або колекцій таких об’єктів, не розкриваючи деталей SQL-запитів чи механізмів з’єднання з базою даних іншим модулям. Така ізоляція забезпечує слабке зв’язування між бізнес-логікою сервера та конкретною реалізацією доступу до даних.

Клас Admin реалізує адміністративний доступ до системи. Він працює в режимі консольної взаємодії і вимагає автентифікації шляхом введення пароля, хеш якого зберігається в базі даних. Після успішної перевірки Admin надає можливість змінювати дані автомобілів, зокрема мінімальні ціни, текстові описи та посилання на PDF-документи. Усі операції запису виконуються через методи класу DB, що гарантує єдиний контроль доступу до бази даних і цілісність даних.

Пакет edu.cardealership.model містить модельні класи Car, Brand, Model, Engine, Transmission, Color, Configuration, Equipment, BodyType та Manager. Вони будуть відображати сутності з бази даних в оперативній пам’яті сервера. Клас Car виступає агрегованою моделлю автомобіля в системі. На відміну від інших модельних класів, що безпосередньо відповідають окремим довідниковим сутностям, клас Car не відображає лише одну таблицю бази даних, а інкапсулює

сукупність взаємопов’язаних сутностей, формуючи цілісний опис автомобільної моделі у вигляді, зручному для передачі мобільному клієнтському застосунку.

Клас Car використовується як універсальний контейнер даних для клієнт-серверного обміну і є основною структурою, яка серіалізується у формат JSON та передається клієнту в межах одного мережевого запиту. Він об’єднує інформацію з кількох сутностей предметної області, зокрема містить посилання на об’єкт класу Model, що мість марку автомобіля, об’єкт Engine з характеристиками двигуна, об’єкт Transmission з інформацією про тип та кількість ступенів коробки передач, а також об’єкт BodyType, який описує тип кузова моделі. Крім того, клас Model включає колекцію доступних кольорів, представлених об’єктами класу Color. Car також містить посилання на комплектацію, що описується об’єктом класу Configuration з відповідним набором обладнання у вигляді об’єкту класу Equipment. Клас Model також зберігає бізнес-атрибути, необхідні для відображення інформації в клієнтському застосунку, зокрема назву моделі, мінімальну ціну у форматі «від», розгорнутий текстовий опис моделі та посилання на PDF-документ з детальною інформацією про комплектації. Це посилання передається клієнту і використовується для відкриття документа у зовнішньому браузері без зберігання PDF-файлів на сервері. Структура класу Model проєктується таким чином, щоб вона відповідала формату даних, які формуються сервером на основі результатів SQL-запитів і безпосередньо передаються по мережі у вигляді JSON.

Клас Car не містить бізнес-логіки, логіки доступу до бази даних або мережевої взаємодії. Його призначення полягає виключно в інкапсуляції та транспортуванні даних предметної області. Такий підхід спрощує процес серіалізації і десеріалізації, зменшує зв’язність між компонентами серверної частини та формує чіткий і стабільний контракт між сервером і клієнтським застосунком. Таким чином, це забезпечує логічну та масштабовану архітектуру серверного проєкту CarDealershipServer. Це рішення повністю відповідає фактичному функціоналу системи, орієнтованої на перегляд асортименту автомобілів та створює основу для подальшого додавання нових характеристик

моделей або підтримки кількох марок у межах одного програмного продукту.

2.2 Апаратні вимоги до сервера

Серверна частина системи автосалону повинна забезпечувати одночасну обробку сотень підключень від мобільних клієнтів, зберігання та обробку даних про моделі автомобілів, комплектації та прайс-листи, а також підтримку адміністрування бази даних. Сервер повинен мати багатоядерний процесор високої продуктивності, наприклад AMD Ryzen 9 7950X із 16 ядрами та 32 потоками або Intel Core i9-14900K із 24 ядрами і 32 потоками. Тактова частота процесора в базовому режимі має становити 4.0–4.5 ГГц із турбо-режимом до 5.7 ГГц, що дозволяє ефективно обслуговувати велику кількість одночасних клієнтських потоків, а багатопоточність забезпечує паралельну обробку запитів до бази даних. Обсяг оперативної пам’яті повинен бути не менше 32 ГБ DDR5 із можливістю розширення до 64 ГБ та тактовою частотою 5600–6000 МГц, що забезпечує достатній кеш для обробки одночасних запитів та швидке зчитування даних з бази. Для зберігання даних рекомендовано використовувати NVMe SSD об’ємом від 2 ТБ, наприклад Samsung 990 PRO або WD Black SN850X, що забезпечує високу швидкість читання та запису JSON-структурованих даних, а також додатковий диск для локальних бекапів об’ємом 4–8 ТБ. Мережева карта повинна підтримувати швидкість 10 Гбіт/с, TCP/IP та jumbo frames, щоб забезпечити стабільну передачу даних сотням одночасних клієнтських підключень.

Для забезпечення безперервної роботи сервера при перебоях електропостачання застосовується онлайн-UPS потужністю 1500–2000 ВА з автономною роботою мінімум 15–30 хвилин, що дає можливість коректно завершити всі запити та переключити систему на резервні джерела живлення. UPS підключається через USB або мережевий інтерфейс до сервера для автоматичного

завершення роботи або сповіщення адміністратора про перебої. Резервне копіювання реалізується багаторівнево. Локальне копіювання проводиться щодня на окремий SSD або HDD у корпусі або на локальний NAS, зберігаючи SQL dump або binary snapshot бази даних, а старі копії ротаційно видаляються через 7–14 днів. Віддалене резервне копіювання здійснюється щотижнево на хмарні сервіси або резервний сервер, з шифруванням даних AES-256 та передачею через захищений канал VPN або SFTP. Логи системи зберігаються ротаційно, наприклад за останні 30–60 днів, щоб економити дисковий простір та зберігати історію роботи сервера.

Для захисту серверної інфраструктури використовується апаратний firewall, який обмежує доступ лише на порти серверного застосунку та адміністративний порт, а віддалене адміністрування можливе лише через VPN із двофакторною автентифікацією. Додатково рекомендується встановити систему моніторингу стану сервера, яка відстежує завантаження CPU, RAM, диску та мережі, а у випадку збоїв надсилає сповіщення адміністратору. Надійність зберігання даних забезпечується використанням RAID-масиву, наприклад RAID 1 для дзеркалювання дисків або RAID 10 для одночасного підвищення швидкодії та захисту від втрати даних, а також регулярним тестуванням можливості відновлення бази з локальних і віддалених бекапів.

Сервер, що відповідає зазначеним характеристикам, здатний ефективно обслуговувати сотні одночасних підключень мобільних клієнтів, забезпечує високу швидкість обробки запитів, надійне резервне копіювання, безпечне адміністрування та масштабованість системи.

Програмна частина системи автосалону буде реалізована із використанням виключно кросплатформного програмного забезпечення, що забезпечує повну незалежність від конкретної операційної системи. Всі серверні модулі написані на Java, яка гарантує однакову роботу на Windows, Linux або інших платформах, підтримуючи стандартні механізми Socket/ServerSocket, Threads, PrintWriter, Scanner та роботу з JSON через Gson. Для взаємодії з базою даних обрана технологія JDBC, яка також є кросплатформною і дозволяє підключатися до

різних СКБД без змін у коді клієнтських модулів. Завдяки використанню лише кросплатформних компонентів немає принципової різниці, чи буде сервер запущено на Windows або Linux. Вибір конкретної ОС може визначатися адміністративними вподобаннями або наявною інфраструктурою, але на роботу системи це не впливає. Такий підхід забезпечує просте розгортання, легке перенесення на інший хост і довготривалу підтримку.

2.3 Проєктування бази даних

База даних клієнт-серверної системи автосалону (рис. 2.2) призначена для централізованого зберігання структурованої інформації про автомобілі, доступні для ознайомлення клієнтами через мобільний застосунок, а також для підтримки адміністративних операцій із редагування цього контенту. Модель даних побудована відповідно до реляційного підходу з дотриманням принципів нормалізації, що забезпечує цілісність даних, відсутність дублювання та можливість масштабування системи.

У базі даних зберігається ієрархія автомобільної номенклатури: від марок до моделей, а також пов’язані з моделями характеристики, такі як доступні комплектації, двигуни, трансмісії, типи кузовів і кольори. Окремо виділено сутності, що описують склад обладнання автомобіля у вигляді PR-кодів, що дозволяє формувати комплектації з великої кількості повторно використовуваних елементів. Для забезпечення керування даними передбачено сутність користувачів з адміністративними правами, які можуть змінювати вміст бази через серверний застосунок. Такий набір сутностей дозволяє реалізувати поточний функціонал мобільного застосунку, орієнтований на перегляд асортименту, і водночас створює основу для подальшого розширення системи без зміни структури бази даних.

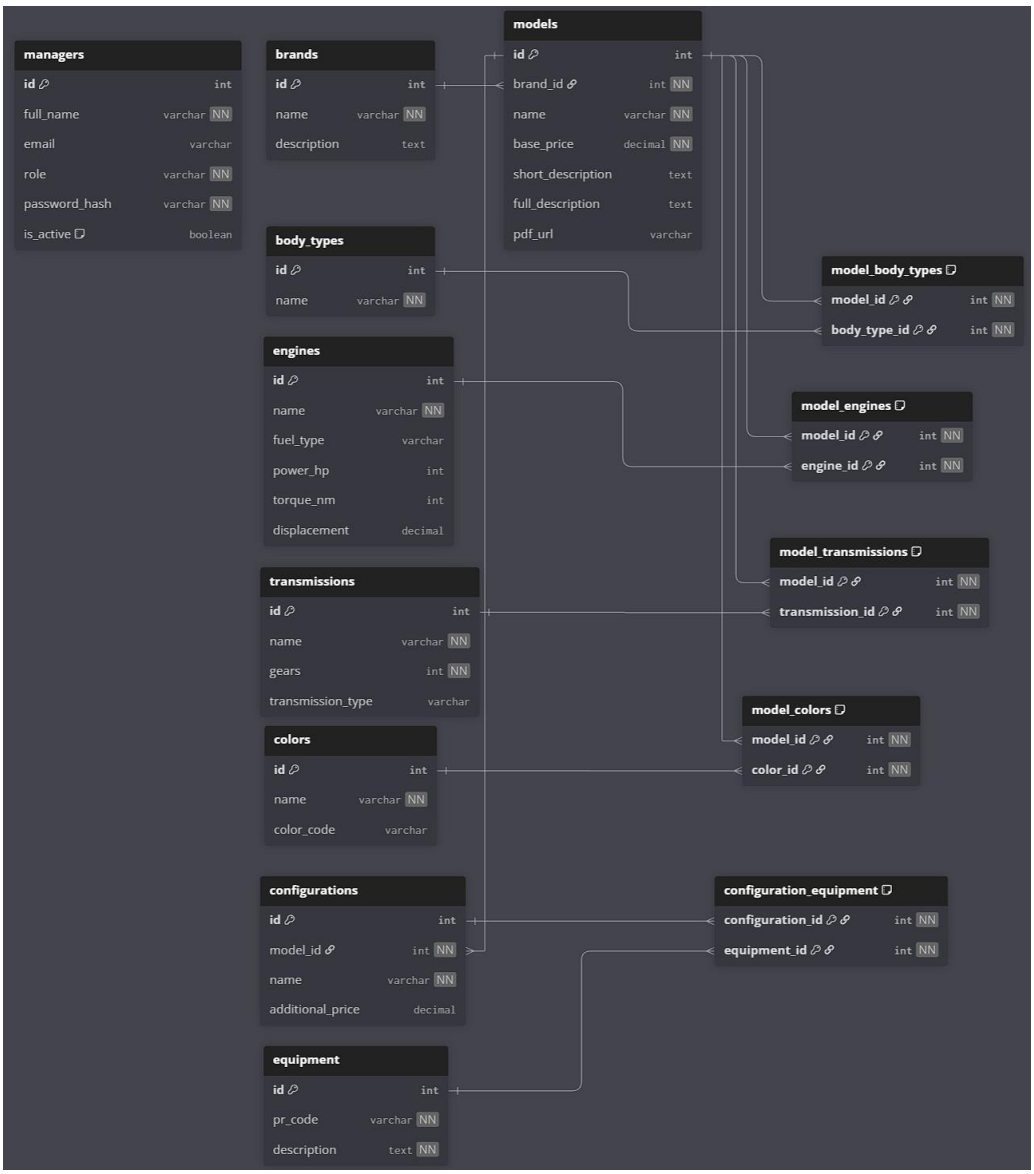


Рисунок 2.2 – Схема бази даних комп’ютерної системи автосалону

Сутність Brands (лістинг 2.1) призначена для зберігання інформації про автомобільні марки. Вона містить унікальну назву марки та, за потреби, її опис. Сутність використовується як верхній рівень ієрархії автомобільних даних і дозволяє групувати моделі за виробником.

Лістинг 2.1 – DBML-опис сутності Brands

```
Table brands {
id int [pk, increment]
name varchar [not null, unique]
description text
}
```

Сутність Models (лістинг 2.2) описує конкретні моделі автомобілів, що належать до певної марки. Вона містить посилання на марку, назву моделі, базову ціну у форматі «від», скорочений та розширений текстові описи, а також посилання на PDF-документ з детальним прайс-листом. Саме ця сутність є центральною для клієнтського застосунку, оскільки на її основі формується більшість відображуваного контенту.

Лістинг 2.2 – DBML-опис сутності Models

```
Table models {
id int [pk, increment]
brand_id int [not null, ref: > brands.id]
name varchar [not null]
base_price decimal [not null]
short_description text
full_description text
pdf_url varchar
}
```

Сутність BodyTypes (лістинг 2.3) призначена для зберігання типів кузовів автомобілів, таких як седан, універсал або хетчбек. Оскільки одна модель може мати декілька варіантів кузова, ця сутність використовується у зв’язці з моделями через таблицю зв’язку.

Лістинг 2.3 – DBML-опис сутності BodyTypes

```
Table body_types {
id int [pk, increment]
name varchar [not null, unique]
}
```

Сутність Engines (лістинг 2.4) зберігає інформацію про двигуни, доступні для моделей. До її атрибутів належать назва двигуна, тип пального, потужність, крутний момент і робочий об’єм. Один і той самий двигун може використовуватися в різних моделях, що обґрунтовує його винесення в окрему сутність.

Лістинг 2.4 – DBML-опис сутності Engines

```
Table engines {
id int [pk, increment]
name varchar [not null]
fuel_type varchar
power_hp int
torque_nm int
displacement decimal
}
```

Сутність Transmissions (лістинг 2.5) описує коробки передач, що можуть встановлюватися на автомобілі. Вона містить назву трансмісії, кількість ступенів і загальний тип. Така структура дозволяє коректно відображати різні варіанти трансмісій для однієї моделі.

Лістинг 2.5 – DBML-опис сутності Transmissions

```
Table transmissions {
id int [pk, increment]
name varchar [not null]
gears int [not null]
transmission_type varchar
}
```

Сутність Colors (лістинг 2.6) призначена для зберігання довідника доступних кольорів кузова. Вона містить назву кольору та, за необхідності, внутрішній або виробничий код. Окрема сутність кольорів дозволяє повторно використовувати однакові кольори для різних моделей без дублювання даних.

Лістинг 2.6 – DBML-опис сутності Colors

```
Table colors {
id int [pk, increment]
name varchar [not null]
color_code varchar
}
```

Сутність Configurations (лістинг 2.7) описує комплектації автомобілів, наприклад, Essense, Selection або Sport. Кожна комплектація пов’язана з конкретною моделлю і має власну додану вартість до базової ціни. Це дозволяє

формувати різні рівні оснащення в межах однієї моделі.

Лістинг 2.7 – DBML-опис сутності Configurations

```
Table configurations {
id int [pk, increment]
model_id int [not null, ref: > models.id]
name varchar [not null]
additional_price decimal
}
```

Сутність Equipment (лістинг 2.8) призначена для зберігання переліку елементів обладнання у вигляді PR-кодів та їхніх описів. PR-код – це унікальний для кожного виробника текстовий код обладнання. Наприклад, в концерні VAG асистент контролю смуг руху – Side Assist має PR-код 7Y1, задній та передній парктронік – 7X2. Таким чином, сутність виступає універсальним довідником оснащення, яке може входити до різних комплектацій.

Лістинг 2.8 – DBML-опис сутності Equipment

```
Table equipment {
id int [pk, increment]
pr_code varchar [not null, unique]
description text [not null]
}
```

Сутність ConfigurationEquipment (лістинг 2.9) реалізує зв’язок між комплектаціями та обладнанням. Вона дозволяє описати, які саме PR-коди входять до конкретної комплектації, та забезпечує гнучке формування наборів оснащення. Складений первинний ключ у цій сутності утворений двома зовнішніми ключами - configuration_id та equipment_id. Такий підхід означає, що унікальність кожного запису визначається не окремим полем, а саме комбінацією значень обох полів. Тобто одна й та сама пара «комплектація - елемент обладнання» може бути присутня в таблиці лише один раз.

Лістинг 2.9 – DBML-опис сутності ConfigurationEquipment

```
Table configuration_equipment {
```

```
configuration_id int [not null, ref: > configurations.id]
equipment_id int [not null, ref: > equipment.id]

indexes { (configuration_id, equipment_id) [pk]}
}
```

Сутності ModelBodyTypes, ModelEngines, ModelTransmissions і ModelColors (лістинг 2.10) реалізують зв’язки багато-до-багатьох між моделями та відповідними довідниковими сутностями. Вони дозволяють задавати допустимі комбінації характеристик для кожної моделі без порушення нормалізації даних.

Лістинг 2.10 – DBML-опис сутностей ModelBodyTypes, ModelEngines, ModelTransmissions і ModelColors

```
Table model_body_types {
model_id int [not null, ref: > models.id]
body_type_id int [not null, ref: > body_types.id]
indexes {(model_id, body_type_id) [pk]}
}

Table model_engines {
model_id int [not null, ref: > models.id]
engine_id int [not null, ref: > engines.id]
indexes {(model_id, engine_id) [pk]}
}

Table model_transmissions {
model_id int [not null, ref: > models.id]
transmission_id int [not null, ref: > transmissions.id]
indexes {(model_id, transmission_id) [pk]}
}
Table model_colors {
model_id int [not null, ref: > models.id]
color_id int [not null, ref: > colors.id]
indexes {(model_id, color_id) [pk]}
}
```

Використання складеного первинного ключа у зв’язкових таблицях обумовлене тим, що такі таблиці не представляють самостійні сутності предметної області, а виконують роль формалізованого опису зв’язків між іншими сутностями. У випадку таблиці configuration_equipment вона не зберігає власних бізнес-атрибутів, а лише фіксує факт входження певного PR-коду обладнання до конкретної комплектації автомобіля. Додавання окремого

ідентифікатора в такій ситуації не має сенсу.

Складений первинний ключ виконує одразу декілька важливих функцій. По-перше, він гарантує логічну цілісність даних, запобігаючи дублюванню зв’язків. Наприклад, неможливо двічі додати той самий PR-код до однієї комплектації, що виключає появу некоректних або надлишкових даних. По-друге, він автоматично створює індекс, який оптимізує виконання запитів, що фільтруються за configuration_id, а також запитів, які використовують обидва поля ключа.

З точки зору нормалізації бази даних, використання складених первинних ключів у таких таблицях відповідає вимогам третьої нормальної форми. Усі неключові залежності відсутні, а кожен запис таблиці однозначно ідентифікується мінімальним набором атрибутів. Це забезпечує прозору структуру даних і спрощує подальший супровід бази. Аналогічний підхід застосовується і в інших зв’язкових таблицях системи, таких як model_colors, model_engines, model_transmissions та model_body_types. У кожному з цих випадків складений первинний ключ відображає семантику зв’язку між двома сутностями і забезпечує унікальність допустимих комбінацій характеристик для конкретної моделі автомобіля.

Сутність Managers (лістинг 2.11) призначена для зберігання інформації про користувачів з адміністративними правами. Вона містить персональні дані, роль користувача та хеш пароля. Ця сутність використовується серверним застосунком для автентифікації адміністратора та контролю доступу до операцій редагування даних.

Лістинг 2.11 – DBML-опис сутності Managers

```
Table managers {
id int [pk, increment]
full_name varchar [not null]
email varchar [unique]
role varchar [not null]
password_hash varchar [not null]
is_active boolean [default: true]
}
```

У базі даних системи паролі адміністративних облікових записів не зберігаються у відкритому вигляді, а зберігаються лише їхні криптографічні хеші. Такий підхід є базовою вимогою інформаційної безпеки і спрямований на мінімізацію ризиків у разі несанкціонованого доступу. Хешування є одностороннім перетворенням, тобто за збереженим значенням неможливо відновити початковий пароль, що принципово відрізняє його від простого шифрування або збереження у відкритому вигляді [20]. Під час автентифікації адміністратора введений пароль не порівнюється безпосередньо з даними в базі. Натомість він проходить через алгоритм хешування, після чого отриманий хеш порівнюється зі значенням, збереженим у полі password_hash. Це дозволяє перевірити правильність пароля без його розкриття системі або зберігання у пам’яті у відкритому вигляді. Такий підхід відповідає сучасним практикам захисту облікових даних та знижує наслідки можливих витоків інформації.

У сукупності описані сутності формують цілісну, логічно узгоджену модель бази даних, яка відповідає задачам комп’ютерної системи автосалону та забезпечує баланс між поточними вимогами і можливістю подальшого розвитку.

Первинне наповнення бази даних відбувається з конфігураційного файлу DB.conf, фрагмент якого наведено в лістингу А.1.

В такий спосіб база даних отримає первинний контент, який клієнти зможуть одержувати на свої мобільні пристрої через сервер по мережі. Подальші зміни в базі можливі з адміністраторського облікового запису, що надає доступ в базу на запис.

2.4 Реалізація серверних модулів

Почнемо реалізацію з моделі. В якості прикладу наведемо лістинг 2.12 класу Model, що відображає сутність моделі автомобіля. Набір полів дозволяє клієнту отримувати ключові технічні характеристики. Таким чином, Model

посилається на інші класи, що описують технічні характеристики моделі. Це зроблено для того, щоб не дублювати дані, а використати принцип інкапсуляції об’єктно-орієнтованого програмування.

Лістинг 2.12 – Код класу Model

```
package edu.cardealershipserver.model;

public class Model {
private int id;
private Brand brand;
private String name;
private BodyType bodyType;
private Engine engine;
private Transmission transmission;
private Color color;
private double base_price;
private String description;
private String pdf_url;
public Model(int id, Brand brand, String name, BodyType bodyType,
Engine engine, Transmission transmission, Color color, double
base_price, String description, String pdf_url) {
this.id = id;
this.brand = brand;
this.name = name;
this.bodyType = bodyType;
this.engine = engine;
this.transmission = transmission;
this.color = color;
this.base_price = base_price;
this.description = description;
this.pdf_url = pdf_url;
}
public int getId() {
return id;
}
public void setId(int id) {
this.id = id;
}    //інші getters and setters, toString
}
```

В лістингу 2.13 показано код класу Car, що агрегує конкретну модель в доступній конфігурації. Таким чином, Car агрегує Model, що в свою чергу агрегує всі інші класи, які описують модель автомобіля. Також Car містить Configuration (комплектацію автомобіля), яка агрегує перелік (список) обладнання Equipment, що входить в цю комплектацію. Клас Manager потрібен для відображення

сутності адміністратора і не пов’язаний з класом Car. На рис. 2.3 показане співвідношення між класами моделі.

Лістинг 2.13 – Код класу Car

```
package edu.cardealershipserver.model;
public class Car {
private Model model;
private Configuration configuration;
public Car(Model model, Configuration configuration) {
this.model = model;
this.configuration = configuration;
}
public Model getModel() {
return model;
}
public void setModel(Model model) {
this.model = model;
}
public Configuration getConfiguration() {
return configuration;
}
public void setConfiguration(Configuration configuration) {
this.configuration = configuration;
}
@Override
public String toString() {
return "Car{" +
"model=" + model +
", configuration=" + configuration +
'}';    }}
```

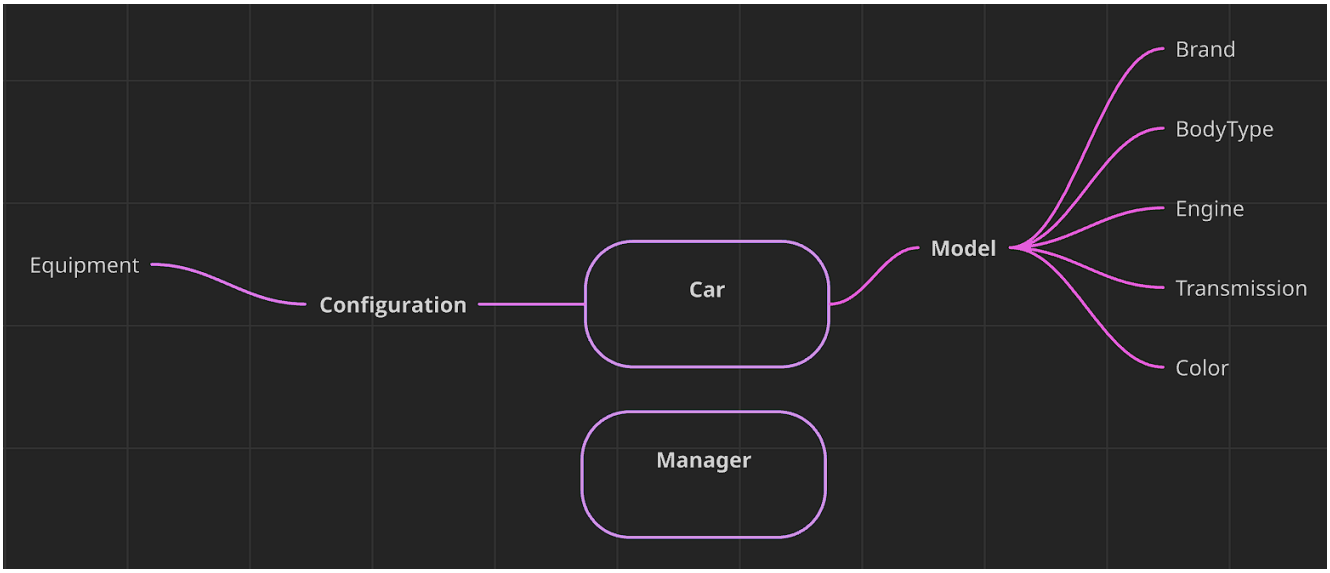


Рисунок 2.3 – Співвідношення між класами моделі

Поля всіх інших класів повністю відповідають полям відповідних сутностей і дають представлення даних в оперативній пам’яті сервера. Після виборки по базі даних, отримані дані будуть зберігатися в екземплярах відповідних класів моделі. Далі ці класи обгортаються в JSON формат та можуть бути відправлені клієнту для відображення на мобільному пристрої.

Далі реалізуємо контролери і почнемо з класу DB, що призначений для опрацювання роботи з базою даних. Він має надавати методи первинного створення та ініціалізації бази даних, методи наповнення бази відповідним контентом, методи вичитки даних для подальшої передачі клієнтам по мережі. Наведемо перелік ключових методів (табл. 2.1) у форматі: назва, аргументи, призначення. Цього набору методів достатньо для мінімально життєздатної реалізації серверної частини: клієнт може переглядати асортимент, деталі моделей і комплектацій, а адміністратор - виконувати базове керування даними в базі.

Таблиця 2.1 – Ключові методи контролера DB

Метод	Призначення
void connect()	Ініціалізує JDBC-з’єднання з БД на основі параметрів підключення (URL, user, password). Викликається під час старту сервера
void disconnect()	Закриває JDBC-з’єднання та пов’язані ресурси під час зупинки сервера
List<Brand> getAllBrands()	Повертає список усіх марок автомобілів з таблиці brands. Використовується клієнтом для початкового вибору марки.
List<Model> getModelsByBrandId(int brandId)	Повертає всі моделі заданої марки разом з базовими характеристиками (bodyType, engine, transmission, color, base_price, description, pdf_url). Сервер віддає всі моделі марки за один запит.
Model getModelById(int modelId)	Повертає одну модель з повним набором пов’язаних сутностей. Використовується при переході на детальний екран моделі
List<Configuration> getConfigurationsByModelId(int modelId)	Повертає всі комплектації для конкретної моделі. Кожна Configuration містить власну назву, додаткову ціну та список обладнання

Продовження таблиці 2.1

Метод	Призначення
List<Equipment> getEquipmentByConfigurationId(int configurationId)	Повертає перелік обладнання, що входить до заданої комплектації. Застосовується при побудові об’єкта Configuration.
Manager getManagerByLogin(String login)	Повертає обліковий запис адміністратора разом по логіну. Використовується виключно для автентифікації
boolean validateManagerCredentials(String login, String passwordHash)	Перевіряє відповідність переданих облікових даних запису в БД. Повертає true у разі успішної автентифікації
void updateModelPrice(int modelId, double newBasePrice)	Адміністративна операція для зміни базової ціни моделі по id без редагування інших параметрів.
void updateConfigurationAdditionalPrice (int configurationId, double additionalPrice)	Дозволяє встановити вартість заданої комплектації
Car buildCar(int modelId, int configurationId)	Метод агрегує Model і Configuration в один об’єкт Car. Використовується сервером перед відправкою даних клієнту

Покажемо код методу getModelByID класу DB в лістингу А.2.

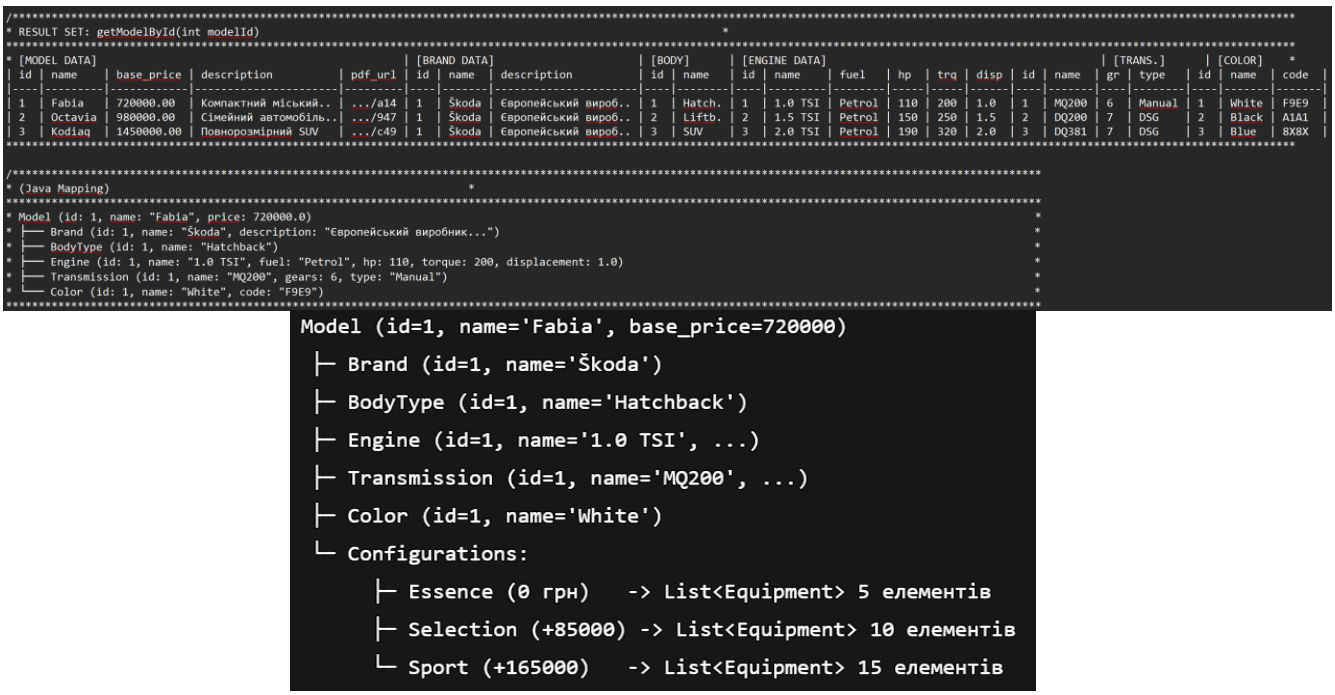
Метод getModelById реалізує повний цикл отримання однієї автомобільної моделі з бази даних та побудови відповідного об’єкта доменної моделі в оперативній пам’яті сервера. На вході метод приймає ідентифікатор моделі modelId, який однозначно визначає запис у таблиці model. Далі формується параметризований SQL-запит, у якому таблиця model виступає центральною, а всі пов’язані з нею сутності підтягуються через JOIN. Використання JOIN дозволяє за один запит отримати всі необхідні дані без додаткових звернень до бази. Запит явно вибирає поля кожної сутності з псевдонімами (AS ...). Це робиться для уникнення конфліктів імен колонок (id, name) та для чіткого зіставлення значень із параметрами конструкторів модельних класів. Наприклад, окремо вибираються ідентифікатор, назва та опис марки (brand_id, brand_name, brand_description), атрибути типу кузова, технічні характеристики двигуна, параметри трансмісії та дані кольору. Після підготовки запиту створюється PreparedStatement, у який передається значення modelId. Використання PreparedStatement забезпечує захист

від SQL-ін’єкцій та дозволяє СКБД повторно використовувати план виконання запиту. Далі виконується запит і отримується ResultSet. Якщо результат порожній, метод повертає null, що сигналізує викликаючому коду про відсутність моделі з таким ідентифікатором. Якщо запис знайдено, відбувається поетапне створення об’єктів модельного рівня. Спочатку створюються об’єкти базових довідникових сутностей - Brand, BodyType, Engine, Transmission та Color. Кожен з них ініціалізується безпосередньо через конструктор з повним набором параметрів, що відповідає структурі таблиць бази даних. Після цього створюється об’єкт Model, який виступає інкапсулюючою сутністю. У його конструктор передається ідентифікатор моделі, назва, базова ціна, текстовий опис та посилання на PDF-документ, а також раніше створені об’єкти Brand, BodyType, Engine, Transmission та Color. Таким чином формується повністю зв’язаний об’єктний граф, що відображає одну конкретну модель автомобіля в тому вигляді, в якому вона зберігається в базі даних. У результаті метод повертає ініціалізований екземпляр класу Model, який може бути без додаткової обробки використаний у серверній логіці, серіалізований у JSON та переданий клієнтському застосунку. Такий підхід відокремлює доступ до даних від бізнес-логіки та забезпечує однозначне відображення реляційної структури бази даних у об’єктній моделі сервера.

Продемонструємо результат виклику метода getModelById(1) з ідентифікатором моделі 1 у вигляді віртуальної таблиці на рис. 2.4.

Для роботи з базою використовується строка підключення (connection string): jdbc:postgresql://localhost:5432/cardealership. Цей формат строки призначено для з’єднання JDBC з PostgreSQL, порт з’єднання за замовчанням 5432, проте ми вказуємо його явно.

Класи Networking та Client виконують ключову роль у забезпеченні мережевої взаємодії між сервером та мобільними клієнтами. Клас Networking відповідає за організацію серверного сокета і прийом вхідних з’єднань від клієнтів. При запуску сервера створюється екземпляр ServerSocket, який прослуховує певний порт, очікуючи на підключення.



```
/*************************************************************************************************************************************************************************************************
* RESULT SET: getModelById(int modelId)                                                                                                                                                          *
*************************************************************************************************************************************************************************************************
* [MODEL DATA]                                                | [BRAND DATA]                                | [BODY]      | [ENGINE DATA]                                                                  | [TRANS.]      | [COLOR]    *
| id | name    | base_price | description          | pdf_url | id | name   | description          | id | name    | id | name     | fuel   | hp  | trq | disp | id | name  | gr | type   | id | name   | code |
|----|---------|------------|----------------------|---------|----|--------|----------------------|----|---------|----|----------|--------|-----|-----|------|----|-------|----|--------|----|--------|------|
| 1  | Fabia   | 720000.00  | Компактний міський.. | .../a14 | 1  | Škoda  | Європейський вироб.. | 1  | Hatch.  | 1  | 1.0 TSI  | Petrol | 110 | 200 | 1.0  | 1  | MQ200 | 6  | Manual | 1  | White  | F9E9 |
| 2  | Octavia | 980000.00  | Сімейний автомобіль.. | .../947 | 1  | Škoda  | Європейський вироб.. | 2  | Liftb.  | 2  | 1.5 TSI  | Petrol | 150 | 250 | 1.5  | 2  | DQ200 | 7  | DSG    | 2  | Black  | A1A1 |
| 3  | Kodiaq  | 1450000.00 | Повнорозмірний SUV   | .../c49 | 1  | Škoda  | Європейський вироб.. | 3  | SUV     | 3  | 2.0 TSI  | Petrol | 190 | 320 | 2.0  | 3  | DQ381 | 7  | DSG    | 3  | Blue   | 8X8X |
*************************************************************************************************************************************************************************************************

/*************************************************************************************************************************************************************************************************
* (Java Mapping)                                                                                                                                                                                 *
*************************************************************************************************************************************************************************************************
* Model (id: 1, name: "Fabia", price: 720000.0)                                                                                                                                                  *
* ├── Brand (id: 1, name: "Škoda", description: "Європейський виробник...")                                                                                                                      *
* ├── BodyType (id: 1, name: "Hatchback")                                                                                                                                                        *
* ├── Engine (id: 1, name: "1.0 TSI", fuel: "Petrol", hp: 110, torque: 200, displacement: 1.0)                                                                                                   *
* ├── Transmission (id: 1, name: "MQ200", gears: 6, type: "Manual")                                                                                                                              *
* └── Color (id: 1, name: "White", code: "F9E9")                                                                                                                                                 *
*************************************************************************************************************************************************************************************************
```

```
Model (id=1, name='Fabia', base_price=720000)
 ├─ Brand (id=1, name='Škoda')
 ├─ BodyType (id=1, name='Hatchback')
 ├─ Engine (id=1, name='1.0 TSI', ...)
 ├─ Transmission (id=1, name='MQ200', ...)
 ├─ Color (id=1, name='White')
 └─ Configurations:
      ├─ Essence (0 грн)   -> List<Equipment> 5 елементів
      ├─ Selection (+85000) -> List<Equipment> 10 елементів
      └─ Sport (+165000)   -> List<Equipment> 15 елементів
```

Рисунок 2.4 – ResultSet виклику метода getModelById(1)

Як тільки клієнт ініціює з’єднання, Networking приймає його та створює для обробки окремий екземпляр класу Client, запускаючи його у новому потоці (Thread). Таким чином забезпечується одночасна робота з багатьма клієнтами без блокування головного потоку сервера. Клас Client є індивідуальним обробником для кожного підключеного клієнта. Він отримує об’єкт сокета від Networking і керує всією взаємодією з цим клієнтом. Основним завданням Client є прийом запитів від клієнтського застосунку, їх обробка та формування відповідей. У контексті нашої системи запит клієнта включає інформацію про марку автомобілів, для якої необхідно отримати список моделей, доступні комплектації та обладнання. Client взаємодіє з класом DB, викликаючи методи для отримання даних із бази, після чого формує структурований об’єкт для передачі назад клієнту.

Для передачі даних між сервером та клієнтом використовується двосторонній бінарний канал, організований через сокети. Усі об’єкти, що передаються, серіалізуються в формат JSON за допомогою бібліотеки Gson, що дозволяє легко декодувати їх на стороні Android-клієнта. Клас Client відповідає за упаковку списку моделей та відповідних конфігурацій у JSON і відправку його

через PrintWriter, а також за отримання запитів від клієнта через Scanner. Такий підхід забезпечує високу швидкодію, оскільки виключає накладні витрати на додаткові протоколи чи обробку HTTP-запитів.

Архітектура класів Networking і Client дозволяє досягти високої масштабованості системи. Завдяки створенню окремого потоку для кожного клієнта сервер може одночасно обслуговувати сотні підключень, не блокуючи роботу інших клієнтів. При цьому кожен екземпляр Client ізольований і управляє лише своїм з'єднанням, що зменшує ризик помилок і підвищує стабільність роботи системи. Додатково, централізована взаємодія з DB через ці класи забезпечує чітке розмежування відповідальностей між мережею та логікою доступу до даних. У результаті клас Networking виступає як керуючий компонент для підключень, а клас Client реалізує протокол обміну даними для конкретного клієнта. Така структура забезпечує зрозумілу і підтримувану архітектуру мережевої взаємодії, де серверна частина може легко масштабуватися, підтримувати багатопотокову обробку, а також інтегрувати нові функції, наприклад, фільтрування моделей, пошук за характеристиками або підтримку декількох марок автомобілів без зміни базової логіки обміну даними.

У Main сервер стартує і підключає базу даних (лістинг 2.14). Якщо користувач вводить логін та пароль, створюється екземпляр Admin, пароль хешується та перевіряється у базі. У разі успішної аутентифікації відкривається консоль адміністратора для перегляду моделей та комплектацій і зміни цін. Якщо дані авторизації не введено або вона не відбулася, тоді сервер стартує без можливості редагування даних та очікує підключення клієнтів через Networking.

Лістинг 2.14 – Код точки входу в серверний застосунок

```
public class Main {
public static void main(String[] args) throws SQLException {
DB db = new DB(); db.connect();
Scanner scanner = new Scanner(System.in);
System.out.println("Enter admin username (or Enter to skip)");
String username = scanner.nextLine().trim();
if (!username.isEmpty()) {
System.out.print("Enter admin password: ");
String password = scanner.nextLine();
```

```
Admin admin = new Admin(db, username);
if (admin.authenticate(password)) {
System.out.println("Authentication successful. You can now update
prices.");
admin.runConsoleInterface();
} else {
System.out.println("Authentication failed! Server will start without
admin access.");
}
} else {
System.out.println("No admin login provided. Server starting without
admin access.");
}
Networking networking = new Networking(5052);
networking.startServer();
}
```

2.5 Висновки по розділу

У цьому розділі було розроблено детальну серверну архітектуру системи клієнт-серверного застосунку автосалону. Було визначено основні модулі серверного проєкту CarDealershipServer, включаючи головний модуль Main для старту сервера та ініціалізації базових налаштувань, модуль Networking для організації мережевої взаємодії з клієнтами, модуль Client для обробки кожного підключення у окремому потоці, модуль DB для взаємодії з базою даних, а також модуль Admin для адміністративного керування контентом і зміни цін. Кожен модуль має чітко визначену відповідальність, що забезпечує розділення логіки та спрощує підтримку та розширення системи.

Особливу увагу було приділено моделі даних і побудові класів моделі в пакеті edu.cardealership.model. Створено класи Brand, Model, BodyType, Engine, Transmission, Color, Configuration, Equipment та Manager, а також клас Car, який інкапсулює конкретну комбінацію моделі та комплектації. Таке рішення дозволяє ефективно відображати дані з бази у пам’яті сервера, забезпечує просту серіалізацію в JSON та передачу клієнту, а також гарантує, що логіка роботи з базою та мережевою взаємодією не переплітається з моделлю даних.

Проєктування бази даних враховувало забезпечення нормалізації та логічних зв’язків між сутностями. Для багатозначних відносин (M-M) між моделями, кольорами, комплектаціями та обладнанням створено окремі таблиці зв’язків. Кожна сутність має первинний ключ, а зв’язки реалізовані через зовнішні ключі, що дозволяє уникнути дублювання даних і забезпечує цілісність бази. Також продумано безпечне зберігання облікових записів адміністраторів у вигляді хешів паролів для захисту від несанкціонованого доступу.

Розроблена серверна архітектура забезпечує роботу системи з можливістю одночасного обслуговування сотень підключень клієнтів. Використання кросплатформного програмного забезпечення гарантує сумісність як з Windows, так і з Linux. Продумана структура класів та модулів створює основу для додавання нових марок автомобілів, комплектацій, функцій адміністрування та масштабування апаратних ресурсів.

3 РОЗРОБКА КЛІЄНТСЬКОЇ ЧАСТИНИ СИСТЕМИ

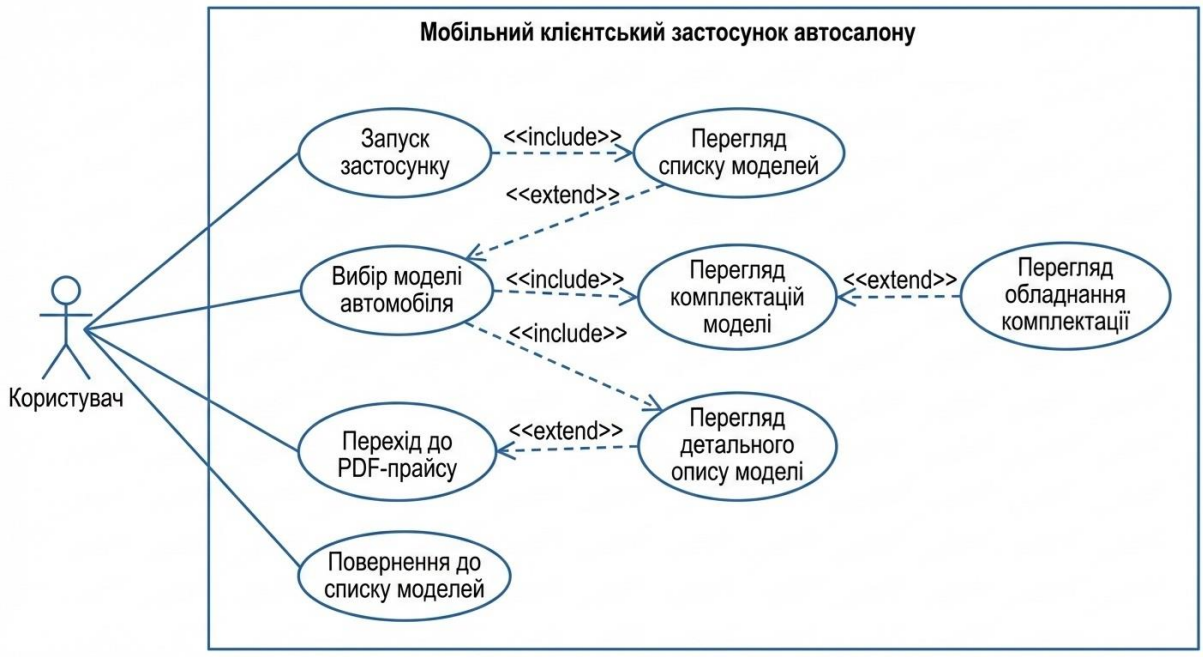
3.1 Проєктування клієнтської архітектури

Почнемо проєктування з діаграми варіантів використання клієнтського застосунку автосалону (рис. 3.1).

Користувач (потенційний покупець автомобіля), який взаємодіє з Android-клієнтом. Система: Мобільний клієнтський застосунок автосалону. Сервер виступає як зовнішня система-постачальник даних і не є актором діаграми. Користувач може використовувати застосунок в таких варіантах:

- запуск застосунку;
- перегляд списку моделей;
- вибір моделі автомобіля;
- перегляд комплектацій моделі;
- перегляд обладнання комплектації;

– перегляд детального опису моделі;
– перехід до PDF-прайсу;
– повернення до списку моделей.



Рисинок 3.1 – Use case діаграма клієнтського застосунку автосалону

При запуску застосунку користувач відкриває мобільний застосунок. Система ініціалізує інтерфейс, перевіряє доступність мережі та встановлює з’єднання із сервером. У разі успішного підключення відображається стартовий екран з коротким описом за стосунку і можливість перейти далі на перелік доступних моделей автомобілів.

При перегляді списку моделей користувач переглядає список автомобілів одного бренду. В подальшому система може бути розширена на мультибрендове використання. Система надсилає один запит до сервера для отримання повного переліку моделей із необхідною інформацією (назва, тип кузова, базова ціна, комплектації, посилання на прас-лист тощо). Отримані дані відображаються у вигляді каруселі.

При виборр моделі автомобіля користувач обирає конкретну модель з каруселі. Система відображає повну інформацію про неї, включаючи доступні

комплектації, опис та технічні характеристики.

При перегляді комплектацій моделі користувач переглядає перелік комплектацій обраної моделі. Для кожної комплектації система відображає назву, додаткову вартість та короткий опис. Комплектації впорядковані від базової до максимальної.

При перегляді обладнання комплектації система відображає список обладнання, що входить до неї, включаючи як базові елементи, так і додаткові опції, що додаються у старших комплектаціях.

При перегляді детального опису модел користувач відкриває екран з детальним описом моделі. Система відображає текстовий опис, технічні характеристики, а також посилання на офіційний PDF-прайс або каталог на сайті офіційного імпортера.

При переході до PDF-прайсу користувач натискає на посилання для перегляду прайсу. Система відкриває PDF-документ у зовнішньому браузері. Посилання на прайс для кожної моделі буде сталим, а вже імпортер буде оновлювати прайси за цими посиланнями.

При поверненні до списку моделей користувач повертається до попередніх екранів для вибору іншої моделі або комплектації. Система використовує вже отримані дані, які зберігаються в оперативній пам’яті без повторного запиту до сервера.

Use case «Запуск застосунку» включає use case «Перегляд списку моделей». Use case «Перегляд списку моделей» розширюється use case «Вибір моделі автомобіля». Use case «Вибір моделі автомобіля» включає «Перегляд комплектацій моделі» та «Перегляд детального опису моделі». Use case «Перегляд комплектацій моделі» розширюється use case «Перегляд обладнання комплектації». Use case «Перегляд детального опису моделі» розширюється use case «Перехід до PDF-прайсу».

Структура інтерфейсу застосунку (UI) будується навколо одного Activity та трьох фрагментів, що реалізує принцип Single Fragment Application, мінімізує кількість компонентів і спрощує навігацію та керування станом. Екрани в

застосунку будуть наступні:

- головний екран (WelcomeFragment);
- екран вибору моделі (CarouselFragment);
- екран детальної інформації про модель (DetailFragment).

Головний екран (WelcomeFragment) - перший фрагмент, який відображається одразу після запуску застосунку. Його призначення - коротко пояснити користувачу суть застосунку та ініціювати подальшу взаємодію. На екрані розміщується текст привітання і одна кнопка «Почати». Натискання кнопки не виконує мережевих операцій, а лише переводить користувача до наступного фрагмента. Це дозволяє чітко відокремити етап ініціалізації інтерфейсу від етапу завантаження даних.

Екран вибору моделі (CarouselFragment) - другий фрагмен, який є основним навігаційним екраном застосунку. На ньому реалізовано карусель моделей автомобілів. Кожен елемент каруселі містить зображення автомобіля, що завантажується з зовнішнього URL, назву моделі, наприклад, Fabia, Octavia, Kodiaq, базову ціну у форматі «від N грн», отриману із сервера. Навігація по каруселі здійснюється за допомогою кнопок «ліворуч» та «праворуч», при цьому перелік моделей циклічний. Під час першого відкриття фрагмента виконується один мережевий запит до сервера для отримання всіх моделей бренду разом із їх комплектаціями та цінами. Отримані дані зберігаються в оперативній пам’яті (у відповідному data holder), що дозволяє уникнути повторних запитів. Натискання на зображення або картку моделі відкриває третій екран.

Екран детальної інформації про модель (DetailFragment) - третій фрагмент, який відображає повну інформацію про обрану модель. На екрані виводяться назва моделі, текстовий опис, базові технічні характеристики, перелік доступних комплектацій із зазначенням додаткової вартості, кнопка «Відкрити прайс-лист». Дані для цього екрану не завантажуються повторно з сервера, а передаються з другого фрагмента у вигляді об’єкта моделі або ідентифікатора з доступом до вже збережених даних. Натискання кнопки відкриття прайс-листа запускає зовнішній браузер з використанням системного Intent та URL, отриманого із сервера. Сам

PDF-файл не зберігається і не обробляється всередині застосунку.

Wireframe клієнтського застосунку представлено на рис. 3.2, що дає розуміння загального вигляду основних екранів та елементів інтерфейсу.

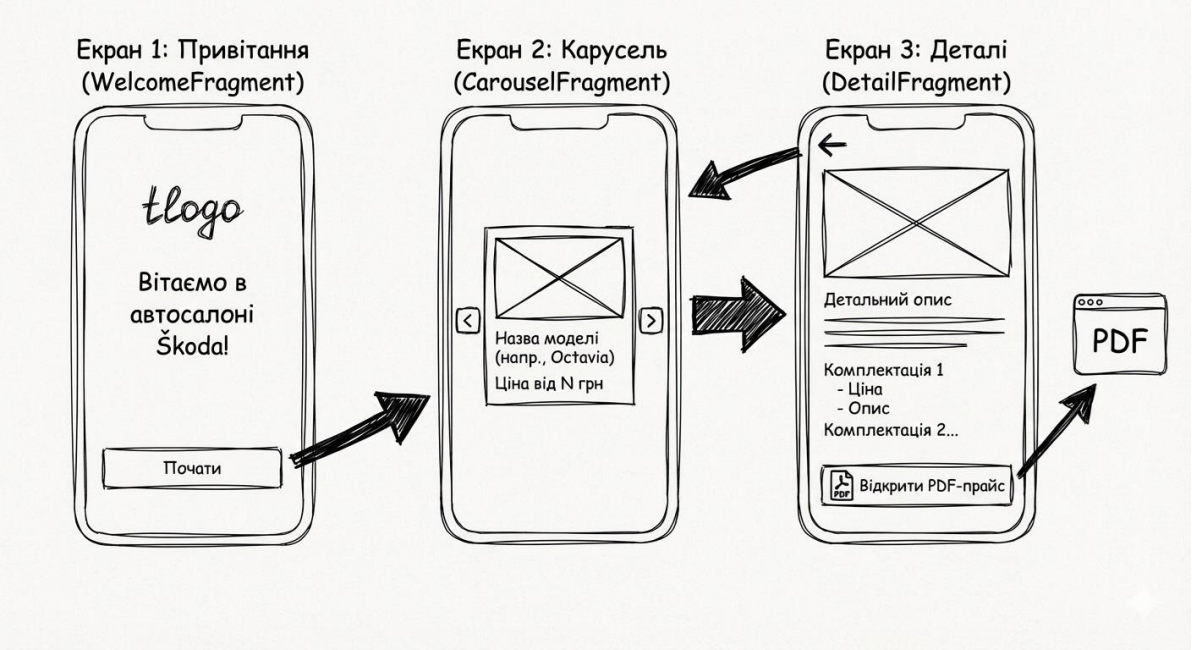


Рисунок 3.2 – Wireframe клієнтського застосунку

В основі архітектури застосунку лежить один Activity, який виконує роль контейнера для фрагментів і точки входу. MainActivity не містить бізнес-логіки та не виконує мережевих операцій. Його відповідальність обмежується ініціалізацією інтерфейсу, керуванням навігацією між фрагментами та збереженням спільного стану застосунку, зокрема кешованих даних, отриманих із сервера, які використовуються різними екранами без повторних запитів.

Першим екраном застосунку є WelcomeFragment, який відображається одразу після запуску. Цей фрагмент виконує виключно презентаційну функцію: він інформує користувача про призначення застосунку та ініціює подальшу взаємодію. На цьому етапі не виконується жодних мережевих запитів або обробки даних. Натискання кнопки «Почати» переводить користувача до наступного фрагмента. Основним робочим екраном є CarouselFragment. Саме в ньому реалізується мережева взаємодія з сервером у межах одного сеансу роботи застосунку. Під час першого відкриття фрагмента у фоновому потоці

викликається клас Networking мережевої взаємодії для отримання повного переліку моделей автомобілів одного бренду разом із пов’язаними з ними даними, такими як базова ціна, комплектації та посилання на прайс-листи. Отримані дані зберігаються в оперативній пам’яті. Сам фрагмент відповідає лише за ініціювання запиту, отримання результату через Handler та відображення даних у користувацькому інтерфейсі. Для відображення розширеної інформації про обрану модель використовується DetailFragment. Він отримує необхідні дані від CarouselFragment через спільне сховище стану в MainActivity. Повторні мережеві запити не виконуються, що забезпечує швидку навігацію та зменшення навантаження на сервер. У цьому фрагменті відображаються текстовий опис моделі, технічні характеристики, перелік доступних комплектацій та кнопка переходу до PDF-прайсу, яка відкриває зовнішній браузер. Окремим обов’язковим компонентом клієнтського застосунку є клас Networking, який інкапсулює всю логіку мережевої взаємодії з сервером. Саме в цьому класі реалізується встановлення socket-з’єднання, формування запитів, обмін повідомленнями та серіалізація і десеріалізація даних у форматі JSON. Такий підхід дозволяє повністю ізолювати мережеву логіку від інтерфейсних компонентів і зберегти простоту та зрозумілість фрагментів. Для представлення отриманих із сервера даних у клієнтському застосунку використовується набір модельних класів, структура яких узгоджена з серверною моделлю. Ці класи відображають сутності предметної області, такі як модель автомобіля, бренд, комплектація, обладнання, двигун, трансмісія, тип кузова та колір. Клієнтські модельні класи не містять бізнес-логіки і слугують виключно контейнерами даних для роботи інтерфейсу. Клас-зв’язка Car інкапсулює конкретну комбінацію моделі та комплектації, що дозволяє зручно оперувати кінцевим варіантом автомобіля в межах клієнтського застосунку. У сукупності така структура класів і фрагментів (рис. 3.3) формує мінімальну структуровану архітектуру клієнтського застосунку, яка відповідає вимогам MVP і логічно узгоджується з архітектурою серверної частини системи.

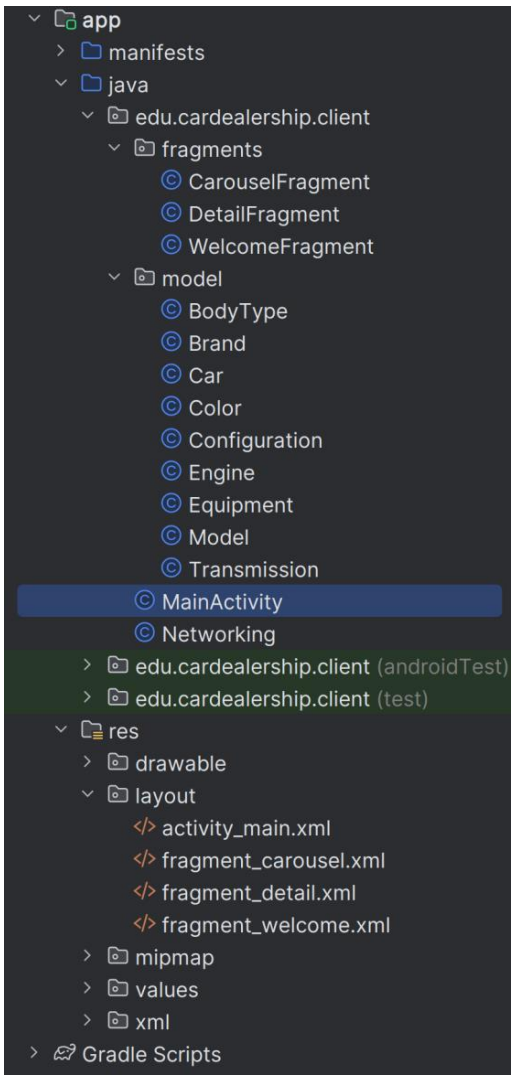


Рисунок 3.3 – Структура проєкту клієнтського застосунку

3.2 Програмно-апаратні вимоги до клієнта

Мінімальною підтримуваною версією операційної системи для клієнтського застосунку доцільно обрати Android 10 (API Level 29). Таке рішення є компромісом між покриттям аудиторії та можливістю використовувати сучасні механізми платформи. Android 10 і новіші версії охоплюють переважну більшість активних пристроїв на ринку і водночас надають стабільний доступ до актуальних API для роботи з мережею, багатопоточністю, фрагментами, системними Intent та керуванням життєвим циклом застосунку. Підтримка старіших версій Android

призвела б до необхідності використання застарілих API, додаткових бібліотек сумісності та складніших рішень для роботи з мережею і фоновими потоками, що ускладнило б реалізацію та супровід застосунку. Водночас орієнтація на новіші версії Android як мінімально необхідні – призвела б до втрати значної частини потенційних користувачів.

З точки зору апаратної платформи клієнтський застосунок не висуває високих вимог до обчислювальних ресурсів. Обробка даних обмежується парсингом JSON, збереженням об’єктів у пам’яті та відображенням інформації в інтерфейсі. Для коректної та плавної роботи достатньо сучасного мобільного процесора середнього рівня, наприклад SoC класу Qualcomm Snapdragon 600 серії або еквівалентних рішень від інших виробників. Використання таких процесорів забезпечує стабільну роботу UI, плавну анімацію каруселі моделей та виконання мережевих операцій у фонових потоках без негативного впливу на користувацький досвід.

Вимоги до оперативної пам’яті також є помірними. Мінімально рекомендований обсяг оперативної пам’яті становить 4 ГБ. Цього вистачить для зберігання в оперативній пам’яті отриманого з сервера переліку моделей, комплектацій та обладнання без повторних запитів, а також для коректної роботи одного Activity з кількома фрагментами. На пристроях із меншим обсягом пам’яті можливі часті перезапуски компонентів системою, що негативно впливатиме на швидкість навігації між екранами та стабільність застосунку.

Клієнтський застосунок орієнтований на постійну мережеву взаємодію з сервером під час першого запуску основного екрану, тому наявність стабільного мережевого з’єднання є обов’язковою умовою коректної роботи. Підтримується як підключення через Wi-Fi, так і через мобільні мережі стандартів LTE та новіших. Обсяг переданих даних є відносно невеликим, оскільки сервер надсилає структуровану текстову інформацію у форматі JSON, а зображення автомобілів та PDF-прайси завантажуються з зовнішніх ресурсів за прямими URL. Це дозволяє використовувати застосунок навіть у мобільних мережах без значного споживання трафіку. В лістингу 3.1 наведено фрагмент JSON-структури, яку

сервер відправляє клієнту. Відповідно, клієнт її парсить та відображає на екрані пристрою.

Лістинг 3.1 – Фрагмент JSON-структури даних, яку сервер відправляє клієнту по мережі


```
"cars": [
{   "model": {
"id": 1,
"name": "Fabia",
"brand": {
"id": 1,
"name": "Škoda"},
"bodyType": {
"id": 1,
"name": "Hatchback"},
"engine": {
"id": 1,
"name": "1.0 TSI",
"fuel_type": "Petrol",
"horsepower": 110,
"torque": 200,
"displacement": 1.0},
"transmission": {
"id": 1,
"name": "MQ200",
"gears": 6,
"type": "Manual"},
"color": {
"id": 1,
"name": "White",
"code": "F9E9"},
"base_price": 720000,
"description": "Компактний міський автомобіль",
"pdf_url":   "https://www.skoda-auto.ua/_doc/a1443d06-d3c9-4ea8-92cd-
9ac0527f487c"},
"configuration": {
"id": 1,
"name": "Essence",
"additional_price": 0,
"equipment": [
{  "id":  1,  "pr_code":  "1AT",  "description":  "Система  стабілізації
ESP" },
{ "id": 2, "pr_code": "7K1", "description": "Контроль тиску в шинах"
},
{ "id": 3, "pr_code": "8T0", "description": "Круїз-контроль" },
{ "id": 4, "pr_code": "8W1", "description": "Денні ходові вогні" },
{  "id":  5,  "pr_code":  "9WT",  "description":  "Базова  мультимедійна
система" }
]  } },
```

Ключові архітектурні моменти:

– сервер не передає сирі таблиці, а формує об’єкти предметної області Car, що повністю відповідають моделі клієнта;

– кожен Car є завершеною одиницею даних для UI, без необхідності додаткових мережевих звернень;

– поле pdf_url передається один раз і безпосередньо використовується клієнтом для відкриття прайсу у браузері;

– вкладені об’єкти (Brand, Engine, Configuration, Equipment) спрощують серіалізацію, роблять JSON читабельним і стабільним як контракт між клієнтом і сервером;

– така структура зручна для одного мережевого з’єднання та локального кешування даних;

– для передачі такої JSON-структури вистачить каналу зв’язку кілька Мб/с.

3.3 Реалізація клієнтських модулів

Класи моделі в клієнтському застосунку будуть аналогічними класам моделі на сервері. Це робиться для того, щоб клієнт міг аналогічно серверу розпарсити отримані JSON-структури даних з автомобілями.

MainActivity є головною Activity клієнтського Android-застосунку. Вона виконує роль центрального координатора між фрагментами користувацького інтерфейсу та фоновою логікою завантаження даних із сервера. У класі реалізовано керування життєвим циклом фрагментів, ініціацію мережевої взаємодії та передачу отриманих даних у UI-потік. Фрагмент основного коду MainActivity наведено в лістингу 3.2.

Лістинг 3.2 – Фрагмент коду основної Activity клієнтського застосунку

```
public class MainActivity extends AppCompatActivity {
public static final int MSG_DATA_LOADED = 1;
```

```
private WelcomeFragment welcomeFragment;
private CarouselFragment carouselFragment;
private DetailFragment detailFragment;
private List<Car> cars = new ArrayList<>();
private Handler uiHandler = new Handler(Looper.getMainLooper(), msg
-> {
if (msg.what == MSG_DATA_LOADED) {
cars = (List<Car>) msg.obj;
carouselFragment.setCars(cars);
getSupportFragmentManager().beginTransaction()
.replace(R.id.fragment_container, carouselFragment)
.commit();
return true;
}
return false;
});
@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
welcomeFragment = new WelcomeFragment();
carouselFragment = new CarouselFragment();
detailFragment = new DetailFragment();
getSupportFragmentManager()
.beginTransaction()
.replace(R.id.fragment_container, welcomeFragment)
.commit();
}
public void onStartButtonClicked() {
new Thread(new Networking(this, uiHandler)).start();}
public void openDetails(Car car) {
detailFragment.setCar(car);
getSupportFragmentManager()
.beginTransaction()
.replace(R.id.fragment_container, detailFragment)
.addToBackStack(null)
.commit();}
//…}
```

MSG_DATA_LOADED константа використовується як ідентифікатор повідомлення в механізмі Handler. Вона дозволяє однозначно визначити тип події: у даному випадку успішне завершення завантаження даних із сервера. Такий підхід є типовим для обміну повідомленнями між фоновими потоками та головним UI-потоком в Android.

MainActivity зберігає посилання на всі фрагменти застосунку:

- WelcomeFragment - початковий екран з кнопкою старту;
- CarouselFragment - екран перегляду списку автомобілів у вигляді

каруселі;

- DetailFragment - екран детального перегляду вибраного автомобіля.

Централізоване керування фрагментами в Activity спрощує навігацію та дозволяє уникнути дублювання логіки переходів між екранами.

Поле cars зберігає колекцію об’єктів типу Car, отриманих із сервера. Кожен об’єкт Car є агрегованою моделлю, що містить дані про автомобіль та його комплектацію. Цей список використовується для наповнення UI-компонентів у CarouselFragment.

Handler ініціалізується з використанням Looper.getMainLooper(), що гарантує обробку повідомлень у головному UI-потоці. Це є критично важливим, оскільки будь-які операції з інтерфейсом користувача в Android дозволені лише в основному потоці. Після отримання повідомлення з кодом MSG_DATA_LOADED з повідомлення витягується список автомобілів, дані передаються до CarouselFragment через метод setCars, виконується заміна поточного фрагмента на CarouselFragment. Таким чином реалізовано асинхронне завантаження даних без блокування інтерфейсу користувача.

У методі onCreate виконується ініціалізація макета Activity, створення екземплярів усіх фрагментів, відображення початкового екрана (WelcomeFragment). Це дозволяє чітко визначити стартовий стан застосунку одразу після його запуску.

Метод onStartButtonClicked викликається після натискання користувачем кнопки старту у WelcomeFragment. У ньому створюється окремий фоновий потік, у якому виконується мережева взаємодія з сервером. Об’єкт Networking отримує посилання на Handler, через який після завершення роботи передає результат у UI-потік.

Метод openDetails використовується для переходу до екрана детального перегляду автомобіля. Перед навігацією вибраний об’єкт Car передається у DetailFragment. Додавання транзакції менеджера фрагментів до стеку зворотнього виклику дозволяє користувачеві повернутися до попереднього екрана з вибором автомобіля стандартною кнопкою «Назад».

Таким чином, MainActivity реалізує шаблон «Single Page Application», тобто виступає в якості навігаційного контролера: керує переходами між фрагментами, ініціює асинхронне завантаження даних та забезпечує коректну синхронізацію між фоновими потоками та UI.

Клас Networking відповідає за реалізацію клієнтської частини мережевої взаємодії з сервером. Він інкапсулює логіку встановлення TCP-з'єднання, обміну даними з сервером та перетворення отриманої відповіді у внутрішні об'єкти предметної області. Клас реалізує інтерфейс Runnable, що дозволяє виконувати мережеві операції у фоновому потоці без блокування головного UI-потоку. Фрагмент класу Networking наведено в лістингу 3.3.

Поля класу визначають адресу та порт сервера, з яким встановлюється TCP-з'єднання. Handler використовується для передачі результатів роботи з фонового потоку у головний UI-потік. Це необхідно для безпечного оновлення елементів інтерфейсу користувача після завершення мережевого запиту. Context передається для можливого доступу до ресурсів або системних сервісів Android. Метод run є точкою входу фонового потоку. Саме в ньому виконується вся мережева логіка. Використовується TCP-сокет для прямої взаємодії з сервером. Потоки введення та виведення обгорнуті у BufferedReader та PrintWriter відповідно, що спрощує роботу з текстовими даними. Конструкція try-with-resources гарантує автоматичне закриття сокета та потоків після завершення роботи або у випадку помилки.

Клієнт надсилає серверу текстову команду з назвою бренду. Сервер інтерпретує цей запит як команду повернути всі автомобілі зазначеного бренду. Такий протокол є простим і достатнім для MVP-реалізації клієнт-серверної взаємодії. Сервер повертає відповідь у форматі JSON одним рядком. Отриманий JSON містить масив об'єктів Car з вкладеними структурами Model та Configuration.

Лістинг 3.3 – Фрагмент класу Networking

```
public class Networking implements Runnable {
private static final String HOST = "16.138.5.8";
private static final int PORT = 5052;
```

```
private final Handler handler;
private Context context;
public Networking(Context context, Handler handler) {
this.context = context;
this.handler = handler;
}
//…
@Override
public void run() {
try (
Socket socket = new Socket(HOST, PORT);
BufferedReader in = new BufferedReader(
new InputStreamReader(socket.getInputStream()));
PrintWriter out =
new PrintWriter(socket.getOutputStream(), true)
)
{
out.println("Skoda"); //Витягнути всі моделі заданого бренду
String json = in.readLine();
Gson gson = new Gson();
Type type = new TypeToken<List<Car>>() {
}.getType();
List<Car> cars = gson.fromJson(json, type);
Message msg = Message.obtain();
msg.what = MainActivity.MSG_DATA_LOADED;
msg.obj = cars;
handler.sendMessage(msg);
} catch (Exception e) {
e.printStackTrace();}
}}
```

Бібліотека Gson використовується для перетворення JSON-рядка у список об’єктів типу Car. Використання TypeToken необхідне через стирання типів у Java, оскільки десеріалізується параметризована колекція.

Після успішного отримання та обробки даних формується повідомлення для Handler. У полі what задається тип події, а в obj передається список автомобілів. Handler у MainActivity приймає це повідомлення та оновлює інтерфейс користувача.

Усі exceptions, що можуть виникнути під час встановлення з’єднання, читання даних або парсингу JSON, перехоплюються. У поточній реалізації помилки лише логуються. В подальшому можна додати візуальні обробники виняткових ситуацій.

Для коректної роботи застосунку з інтернетом потрібно дописати

залежність в маніфесті AndroidManifest.xml: <uses-permission android:name="android.permission.INTERNET" />. Цей дозвіл є базовим системним дозволом, який надає застосунку можливість встановлювати мережеві з’єднання через протоколи TCP/IP, зокрема здійснювати HTTP та HTTPS-запити, відкривати сокет-з’єднання та обмінюватися даними з віддаленими серверами. У контексті цього проєкту цей дозвіл є обов’язковим, оскільки клієнтський Android-застосунок отримує дані про автомобілі з серверного Java-застосунку через мережу, використовуючи сокети та передаючи інформацію у форматі JSON.

Без явного зазначення дозволу android.permission.INTERNET у маніфесті будь-які спроби виконати мережеві операції завершаться помилками на рівні системи, наприклад винятками типу NetworkOnMainThreadException або помилками встановлення з’єднання. При цьому дозвіл INTERNET належить до категорії normal permissions, тому він автоматично надається під час встановлення застосунку і не потребує додаткового підтвердження користувачем під час виконання. Це спрощує розгортання застосунку і не ускладнює користувацький досвід.

Клас CarouselFragment призначений для відображення переліку автомобілів у вигляді каруселі з можливістю циклічного перегляду моделей. Він відповідає виключно за рівень представлення даних і не містить логіки отримання інформації з сервера. Дані передаються у цей фрагмент з головної активності після завершення мережевого запиту. Основний код класу CarouselFragment наведено в лістингу А.3.

У класі зберігається список об’єктів Car, який містить автомобілі одного бренду, а також індекс, що визначає поточну позицію в цьому списку. Індекс використовується для навігації між моделями та забезпечує циклічний перехід при досягненні початку або кінця списку. Також у класі оголошені посилання на елементи інтерфейсу користувача: ImageView для відображення зображення автомобіля та два TextView для виведення назви моделі і її базової ціни.

Метод setCars використовується для передачі списку автомобілів у фрагмент після його отримання з сервера. Метод зберігає список у внутрішньому

полі класу та, у випадку якщо представлення фрагмента вже створене, одразу ініціює оновлення інтерфейсу. Такий підхід дозволяє коректно працювати з асинхронним завантаженням даних і не залежати від життєвого циклу фрагмента.

Метод show відповідає за оновлення вмісту інтерфейсу відповідно до поточного автомобіля. Перед виконанням перевіряється, чи список автомобілів не є порожнім. Далі з об’єкта Car отримується вкладений об’єкт Model, з якого береться назва моделі та її базова ціна. Ці значення відображаються у відповідних текстових полях. Для зображення формується URL на основі назви моделі, після чого використовується бібліотека Glide для асинхронного завантаження зображення з мережі без блокування головного потоку.

Метод getImageUrl виконує зіставлення назви моделі з відповідним мережевим ресурсом зображення. Для кожної підтримуваної моделі повертається заздалегідь визначений URL. Якщо назва моделі не відповідає жодному з відомих варіантів, повертається порожній рядок, що запобігає спробі завантаження некоректного ресурсу.

Загалом CarouselFragment реалізує логіку перегляду списку автомобілів у наочному та інтерактивному вигляді, із чітким розмежуванням відповідальностей між інтерфейсом користувача, активністю та мережевим рівнем застосунку. Така структура є простою, зрозумілою та добре підходить для демонстрації клієнтської частини клієнт-серверної системи у дипломній роботі.

Для коректної роботи бібліотеки Glide а також Google-реалізації парсера JSON потрібно у блоці dependencies файлу build.gradle дописати відповідні залежності (лістинг 3.4).

Лістинг 3.4 – Додаткові залежності в файлі build.gradle

```
dependencies {
implementation 'com.google.code.gson:gson:2.10.1'
implementation 'com.github.bumptech.glide:glide:4.15.1'
annotationProcessor 'com.github.bumptech.glide:compiler:4.15.1'
}
```

Залежність com.google.code.gson:gson:2.10.1 підключає бібліотеку Gson, яка

призначена для серіалізації та десеріалізації даних у форматі JSON. У межах проєкту вона використовується для перетворення JSON-рядка, отриманого від серверного застосунку, у відповідні Java-об’єкти предметної області, зокрема списки об’єктів Car, Model та Configuration. Використання Gson дозволяє уникнути ручного парсингу JSON, спрощує код та зменшує кількість потенційних помилок при обробці структурованих даних.

Залежність com.github.bumptech.glide:glide:4.15.1 підключає бібліотеку Glide, яка відповідає за асинхронне завантаження, кешування та відображення зображень. У даному застосунку Glide використовується для отримання зображень автомобілів з мережі за URL-адресами та їх відображення в елементах ImageView. Бібліотека автоматично керує потоками, кешем пам’яті та диску, що забезпечує високу продуктивність і стабільну роботу інтерфейсу без блокування головного потоку.

Залежність annotationProcessor 'com.github.bumptech.glide:compiler:4.15.1' підключає компілятор анотацій Glide. Вона необхідна для генерації допоміжного коду на етапі компіляції, зокрема для підтримки розширених можливостей бібліотеки, таких як автоматично згенеровані API та робота з кешуванням. Без цього процесора анотацій Glide не зможе повноцінно функціонувати у проєкті.

DetailFragment є фрагментом інтерфейсу користувача, призначеним для відображення детальної інформації про вибраний автомобіль, зокрема опис моделі, комплектацію, перелік обладнання та посилання на офіційний PDF-прайс. Фрагмент використовується як екран другого рівня навігації та відкривається після вибору конкретного автомобіля у фрагменті каруселі. Основний код DetailFragment наведено в лістингу А.4.

У класі оголошується поле car, яке зберігає об’єкт типу Car, переданий з MainActivity. Метод setCar використовується для встановлення цього об’єкта перед відображенням фрагмента. Такий підхід дозволяє розділити логіку навігації та передачі даних від логіки відображення. Метод onCreateView відповідає за створення та ініціалізацію інтерфейсу фрагмента. На першому етапі відбувається inflate XML-макета fragment_detail. Якщо об’єкт car ще не встановлений, метод

повертає створене представлення без заповнення даних, що запобігає виникненню помилок доступу до null-значень. Далі з об’єкта Car отримується вкладений об’єкт Model, який містить загальну інформацію про автомобіль. За допомогою методу findViewById ініціалізуються елементи інтерфейсу: зображення автомобіля, назва моделі, текстовий опис та текстове поле для відображення комплектації. Назва моделі та опис безпосередньо встановлюються у відповідні TextView, що забезпечує базове інформування користувача.

Завантаження зображення моделі реалізовано за допомогою бібліотеки Glide. URL зображення визначається на основі назви моделі через допоміжний метод getImageUrl. Якщо URL не є порожнім, Glide асинхронно завантажує зображення з мережі та відображає його в ImageView. Такий підхід гарантує плавну роботу інтерфейсу без блокування головного потоку.

Фрагмент обробляє інформацію про комплектацію автомобіля. З об’єкта Car отримується об’єкт Configuration, який містить назву комплектації, додаткову вартість та список обладнання. За допомогою StringBuilder формується структурований текстовий опис, який включає назву комплектації, суму доплати та перелік елементів обладнання. Кожен елемент обладнання додається в окремому рядку, що підвищує читабельність та зручність сприйняття інформації користувачем. Сформований текст встановлюється у TextView configurationsView, який відображає повну інформацію про комплектацію автомобіля в межах одного екрану без використання складних спискових компонентів, що є доцільним для демонстраційного та навчального проєкту.

Кнопка відкриття PDF-прайсу реалізована через стандартний механізм Android Intent. При натисканні створюється неявний Intent з дією ACTION_VIEW, якому передається URL PDF-документа з моделі. Це дозволяє відкрити документ у зовнішньому браузері або PDF-переглядачі, встановленому на пристрої, без необхідності реалізації власного механізму перегляду PDF у застосунку.

Клієнт-серверна взаємодія будується за класичною моделлю запит-відповідь із використанням сокетного з’єднання поверх TCP. Після ініціалізації користувачем дії завантаження даних мобільний клієнт формує текстовий запит,

який містить ідентифікатор бренду автомобілів, та відкриває мережеве з’єднання з сервером за заданою IP-адресою і портом. Запит передається серверу через сокет за допомогою потоків введення-виведення. Серверна частина, отримавши запит, інтерпретує його як команду на вибірку моделей певного бренду, після чого звертається до бази даних для отримання пов’язаних сутностей, зокрема моделей автомобілів, їх комплектацій, цін, описів та посилань на зовнішні ресурси. Отримані з бази дані структуруються у вигляді об’єктів предметної області, що відповідають внутрішнім моделям системи.

Після формування повного набору даних сервер серіалізує об’єкти у формат JSON з використанням бібліотеки Gson та передає результат клієнту в одному повідомленні у відповідь на запит. На стороні клієнта отриманий JSON-рядок зчитується з сокета в окремому фоновому потоці, після чого виконується десеріалізація даних у відповідні Java-об’єкти. Результати передаються в головний потік застосунку через механізм Handler, що забезпечує потокобезпечну взаємодію з інтерфейсом користувача. Після цього дані відображаються на екрані у відповідних фрагментах: спочатку у вигляді списку моделей у каруселі, а при виборі конкретного елемента - на екрані детальної інформації з текстовим описом, зображеннями та можливістю переходу до PDF-прайс-листа. Схема клієнт-серверної взаємодії показана на рис. 3.4.

Лістинги верстки екранів клієнтського застосунку подані у додатку Б.

3.4 Висновки по розділу

У цьому розділі було реалізовано клієнтську частину клієнт-серверної комп’ютерної системи у вигляді Android-застосунку. Застосунок забезпечує взаємодію користувача з даними, що надходять із серверної частини, та реалізує повний цикл отримання, обробки і відображення інформації про автомобілі.

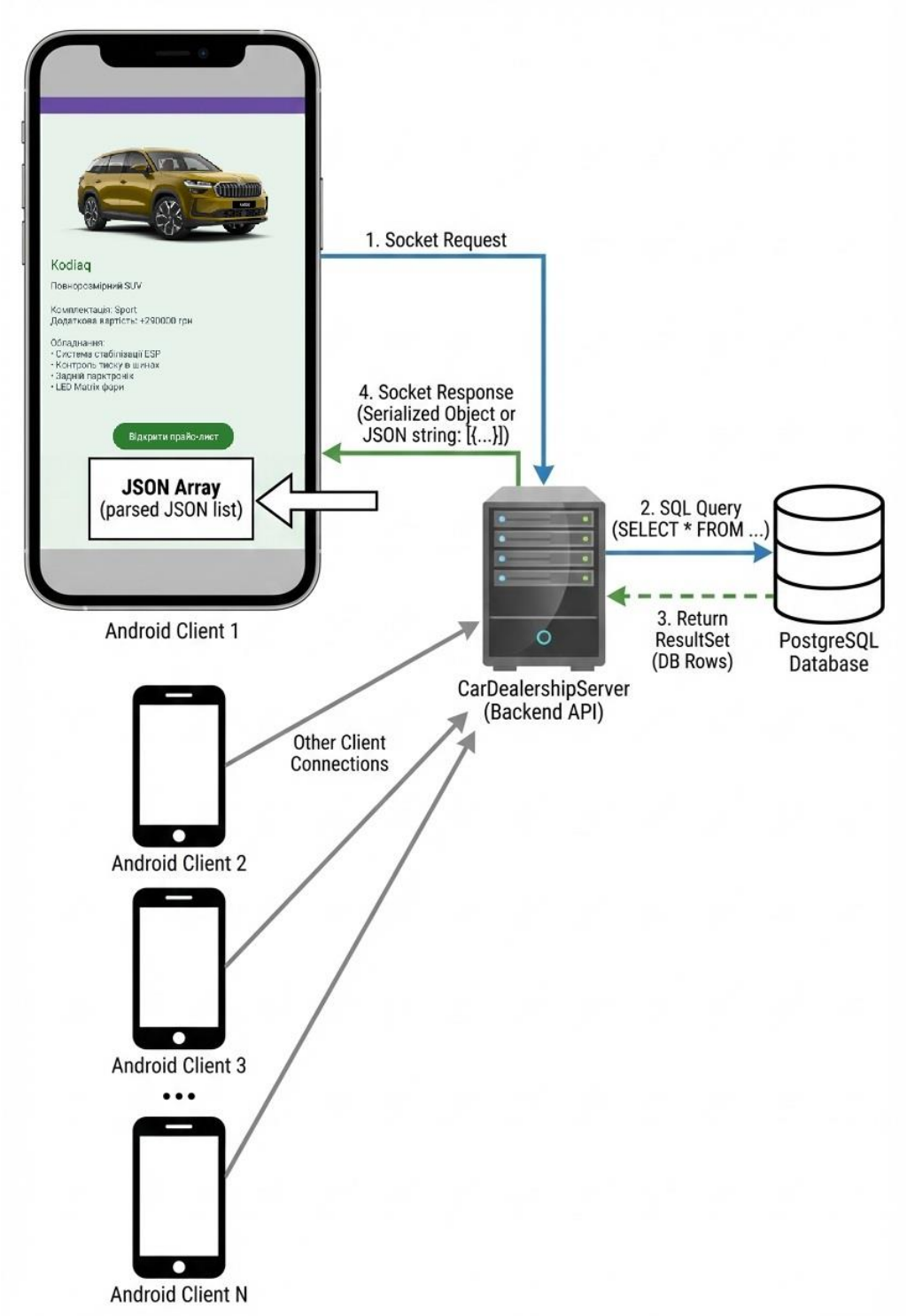


Рисунок 3.4 – Схема клієнт-серверної взаємодії

У процесі розробки було обґрунтовано вибір платформи Android та мови програмування Java як стабільного і поширеного середовища для створення мобільних клієнтів. Архітектура клієнта побудована на основі Activity та Fragment, що дозволило розділити логіку навігації, керування станом і

відображення інтерфейсу, підвищивши модульність та керованість коду.

Реалізовано асинхронну взаємодію з сервером за допомогою окремого потоку та механізму Handler, що забезпечує коректну роботу з мережею без блокування головного потоку інтерфейсу користувача. Для серіалізації та десеріалізації даних використано бібліотеку Gson, що дозволило зручно та надійно перетворювати JSON-структури, отримані з сервера, у об’єкти предметної області.

Інтерфейс користувача реалізовано у вигляді послідовності фрагментів: екрану привітання, каруселі моделей автомобілів та екрану детального перегляду. Карусель забезпечує зручну навігацію між моделями одного бренду, а детальний екран надає повну інформацію про вибрану модель та її комплектацію, включно з переліком обладнання і доступом до офіційного PDF-прайсу. Для завантаження та відображення зображень з мережі використано бібліотеку Glide, що підвищує продуктивність та стабільність роботи інтерфейсу.

Також було враховано системні вимоги платформи Android, зокрема налаштування маніфесту застосунку та використання дозволу INTERNET для коректної мережевої взаємодії. Усі ключові компоненти клієнтської частини інтегровані між собою та взаємодіють відповідно до обраної клієнт-серверної архітектури.

Таким чином, у третьому розділі досягнуто поставленої мети з розробки функціонального мобільного клієнта, який коректно взаємодіє з сервером та забезпечує зручний інтерфейс користувача.

4 ВИПРОБУВАННЯ СИСТЕМИ

4.1 Випробування користувацького інтерфейсу

Проведемо випробування користувацького інтерфейсу мобільного клієнта для перевірки коректності відображення даних, навігації між екранами та доступу

до додаткових ресурсів. Для оцінки роботи інтерфейсу було виконано випробування всіх трьох основних фрагментів застосунку, мережевої взаємодії та функціоналу відкриття PDF-прайсу через браузер.

На етапі запуску застосунку користувачу відображається екран привітання, який реалізовано у вигляді фрагменту WelcomeFragment. Цей екран містить кнопку початку роботи з програмою, що ініціює завантаження даних з серверу та переходить до каруселі моделей автомобілів. Для ілюстрації роботи цього фрагменту наводиться рис. 4.1.

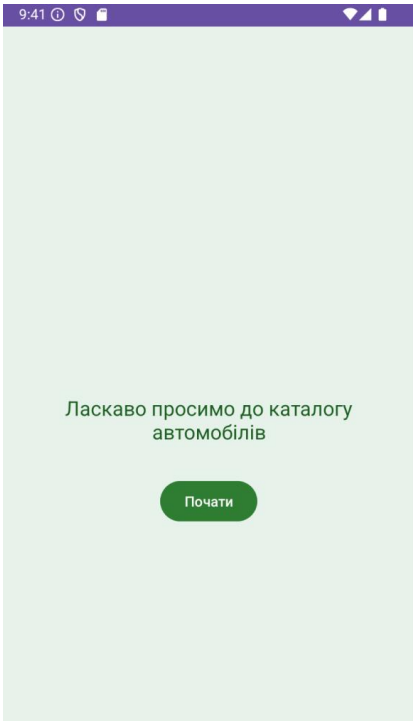


Рисунок 4.1 – Екран привітання WelcomeFragment

Після натискання кнопки «Почати» відбувається завантаження даних з серверу і користувач переходить до фрагменту CarouselFragment, де реалізовано послідовний перегляд автомобілів за допомогою кнопок навігації «Next» та «Prev». Кожна модель відображається із зображенням, назвою та базовою ціною. Навігація між елементами перевірена на коректність циклічного переходу при досягненні першого та останнього елементу. Робота каруселі показана на рис. 4.2.

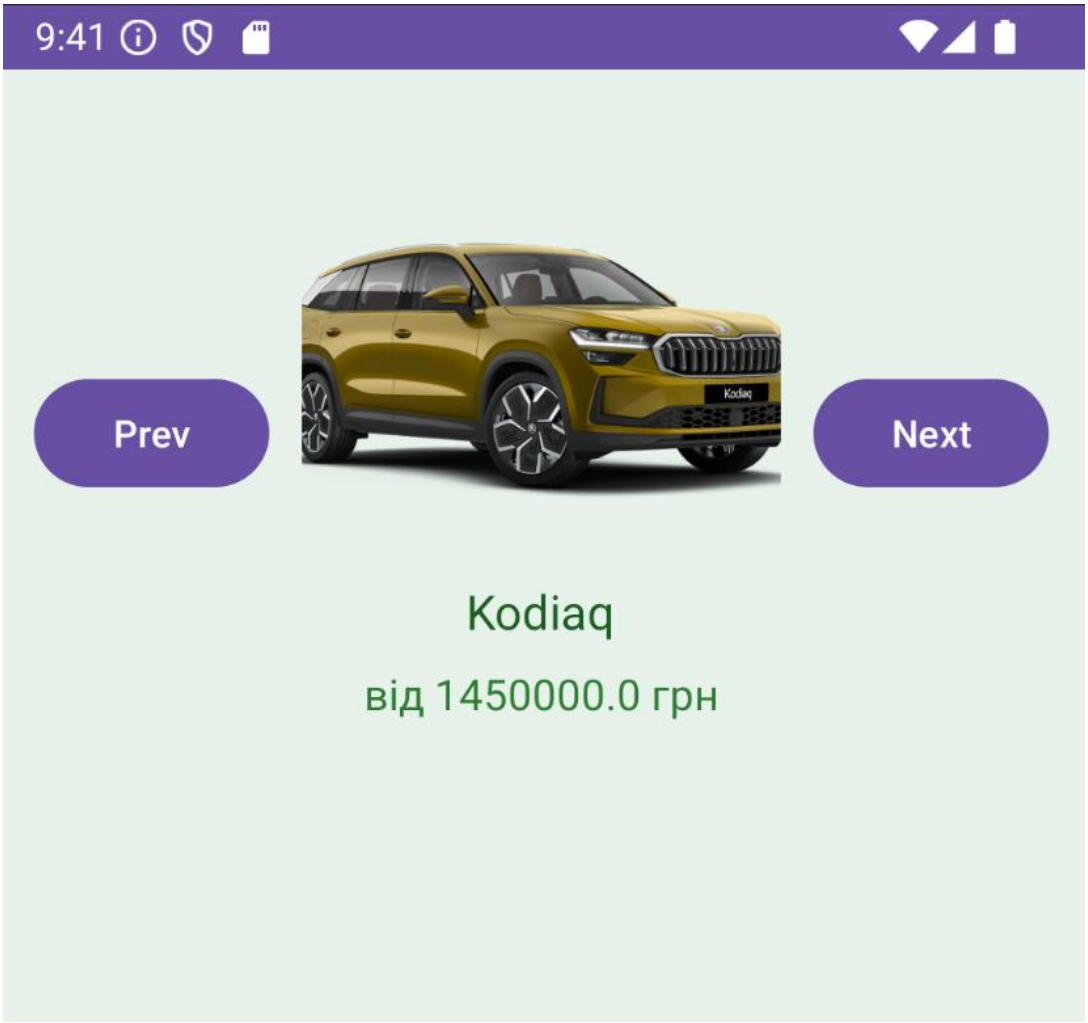


Рисунок 4.2 – Екран вибору моделі CarouselFragment

При натисканні на зображення моделі користувач переходить до фрагменту DetailFragment, де відображаються деталі обраного автомобіля: назва, опис, зображення, інформація про комплектацію, додаткову вартість та перелік обладнання. Інтерфейс цього фрагменту забезпечує зручне та наочне відображення всіх атрибутів моделі та комплектацій. Для ілюстрації роботи екрану деталей наводиться рис. 4.3.

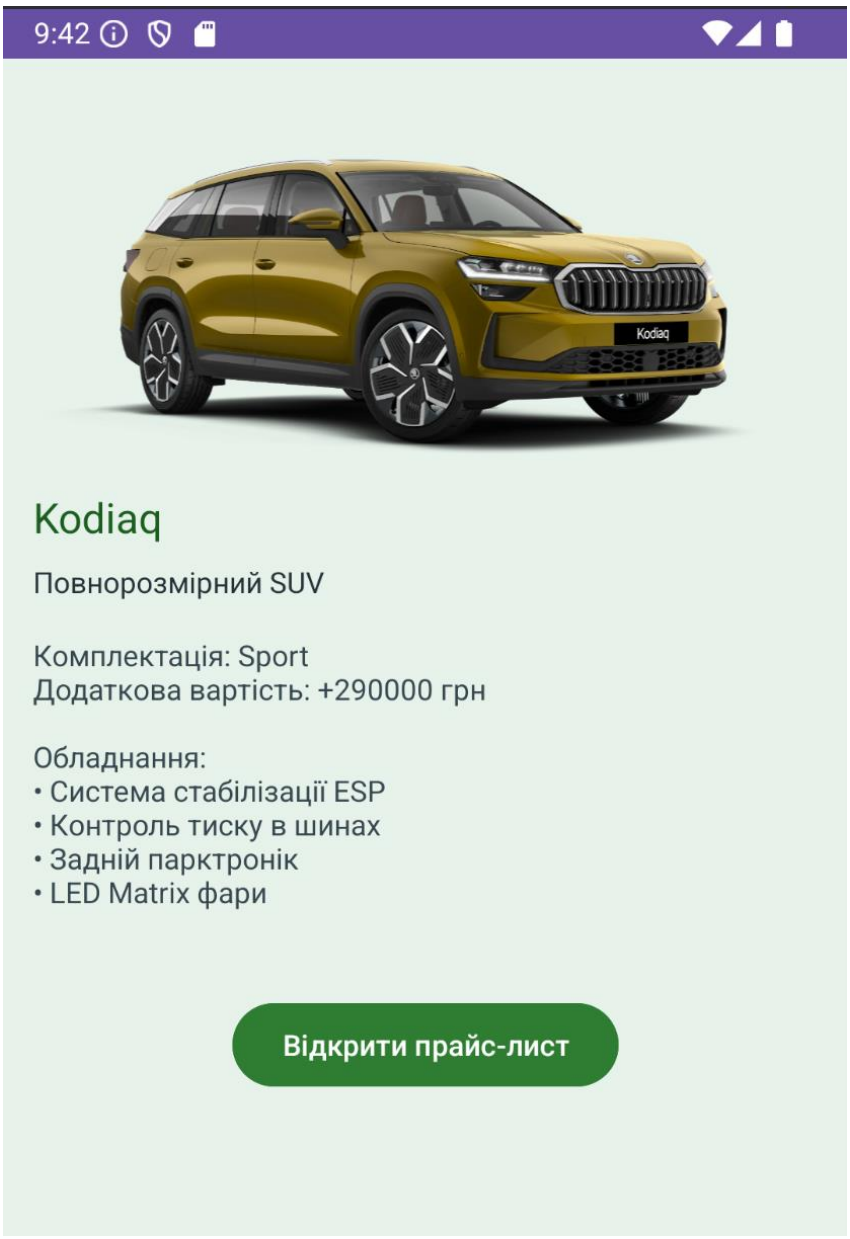


Рисунок 4.3 – Екран деталей моделі DetailFragment

Функціонал відкриття PDF-прайсу реалізовано через кнопку «Відкрити прайс-лист», яка відкриває документ з сайту офіційного імпортера у веб-браузері. Цей елемент перевірено на коректність завантаження та відображення актуальної інформації на прикладі Skoda Kodiaq. Відповідна демонстрація роботи функції наведена на рис. 4.4.

Додатково проведено випробування стандартної кнопки «Назад» на пристрої, яка завдяки додаванню фрагменту деталей у стек зворотніх викликів забезпечує повернення користувача з екрану DetailFragment на екран CarouselFragment. Це дозволяє користувачу швидко повернутися до перегляду

списку моделей без втрати контексту або необхідності повторного завантаження даних, що підтверджує коректну реалізацію навігації між фрагментами.

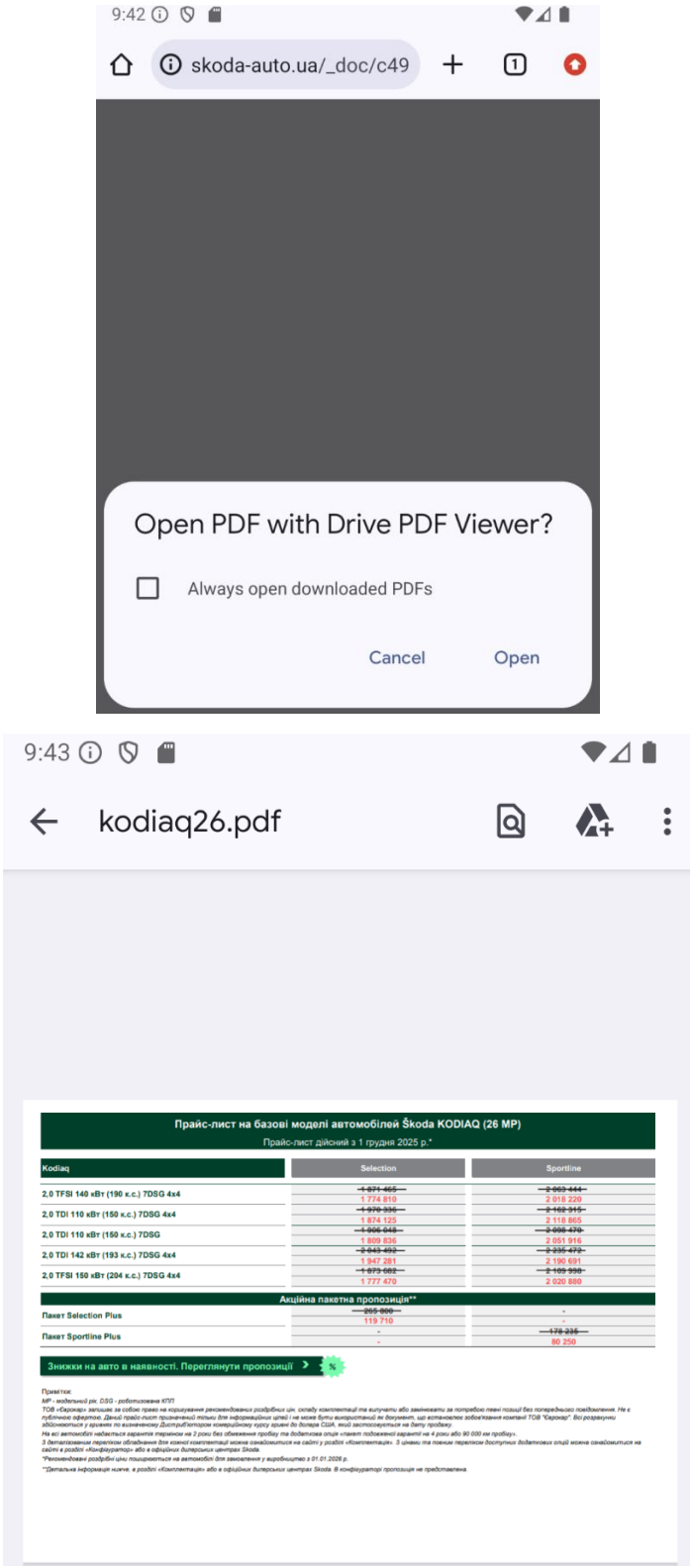


Рисунок 4.4 – Екран завантаження pdf-прайс листу з сайту імпортера

Проведене випробування користувацького інтерфейсу підтвердило коректність відображення інформації, стабільну роботу навігації між фрагментами та доступ до зовнішніх ресурсів, що підтверджує готовність застосунку до експлуатації.

4.2 Аналіз продуктивності системи

Для аналізу продуктивності мобільного клієнта було проведено вимірювання ключових показників з використанням стандартних інструментів Android Studio та вбудованих засобів профілювання. Основні метрики включали споживання оперативної пам’яті, мережевий трафік та час завантаження даних з серверу. Вимірювання проводилися на емуляторі з Android 10 (API 29) із конфігурацією 4 ГБ оперативної пам’яті та підключенням через LTE. Для отримання даних використовували Android Profiler у складі Android Studio, який дозволяє відстежувати використання CPU, пам’яті, мережеві запити та FPS рендерингу.

За результатами профілювання, при завантаженні всього набору моделей (9 об’єктів Car для трьох брендів з трьома комплектаціями кожен) середнє споживання оперативної пам’яті складало приблизно 180–200 МБ, причому більшу частину займали об’єкти Car та кешовані зображення, завантажені через Glide. Після завершення роботи користувача із фрагментом CarouselFragment і DetailFragment споживання пам’яті стабілізувалося на рівні 120–130 МБ.

Час завантаження даних з сервера при підключенні через LTE (швидкість завантаження ~15–20 Мбіт/с) складав у середньому 700–900 мс для отримання JSON з усіма моделями. Завантаження зображень через Glide займало додатково 400–600 мс на зображення при першому зверненні; повторні перегляди використовували кеш, що зменшувало час завантаження до 100–150 мс на зображення.

Мережевий трафік, спожитий клієнтом при завантаженні всіх моделей та зображень, становив приблизно 2,5–2,8 МБ: близько 1,0 МБ припадало на JSON-дані, решта – на графічні ресурси. Передача PDF-прайсу через браузер не впливала на основну продуктивність клієнта, оскільки документ відкривався в зовнішньому застосунку.

Для оцінки продуктивності рендерингу інтерфейсу були використані інструменти GPU Profiler і Layout Inspector, що показали стабільну частоту кадрів на рівні 50–55 FPS під час перемикання моделей у CarouselFragment, без помітних лагів чи підвисань.

Порівняння отриманих показників продуктивності з типовими значеннями для подібних мобільних застосунків показує, що наш клієнт працює в очікуваних межах. Згідно з даними Android Profiler типовий мобільний застосунок такого типу при завантаженні відповідних зображень споживає від 150 до 250 МБ оперативної пам’яті, витрачає на мережевий трафік 2–4 МБ і завантажує дані за 0,8–1,2 с при LTE. Наші показники (180–200 МБ ОЗП, 2,5–2,8 МБ трафіку, 0,7–0,9 с завантаження JSON) знаходяться в цьому діапазоні, що підтверджує ефективність реалізації та правильне використання кешування зображень через Glide.

Стандартні веб-сторінки з каталогами автомобілів зазвичай завантажують весь HTML, CSS, JavaScript та медіа-файли, що призводить до споживання 250–400 МБ оперативної пам’яті та мережевого трафіку 5–8 МБ для тих самих 9 моделей, а час повного завантаження сторінки при LTE складав би 1,5–2,0 с. Це порівняння демонструє, що наш застосунок забезпечує значно більш економне використання ресурсів пристрою і швидший доступ до даних, що покращує користувацький досвід, особливо на пристроях з обмеженою пам’яттю та при мобільному підключенні. При тому наш застосунок дає значно кращий користувацький досвід, що не перевантажений зайвою інформацією.

На рис. 4.5 наведено порівняльний аналіз показників продуктивності запропонованої системи з типовою мобільною системою та з типовою веб-системою. Веб-системи працюють значно повільніше та використовують значно

більше пам’яті через представлення великого обсягу інформації та реклами, який потрібно завантажити браузеру.

Таким чином, проведений аналіз підтвердив, що клієнтський застосунок працює стабільно, з прийнятним споживанням пам’яті та мережевого трафіку, забезпечує швидке завантаження даних і плавну навігацію між фрагментами.

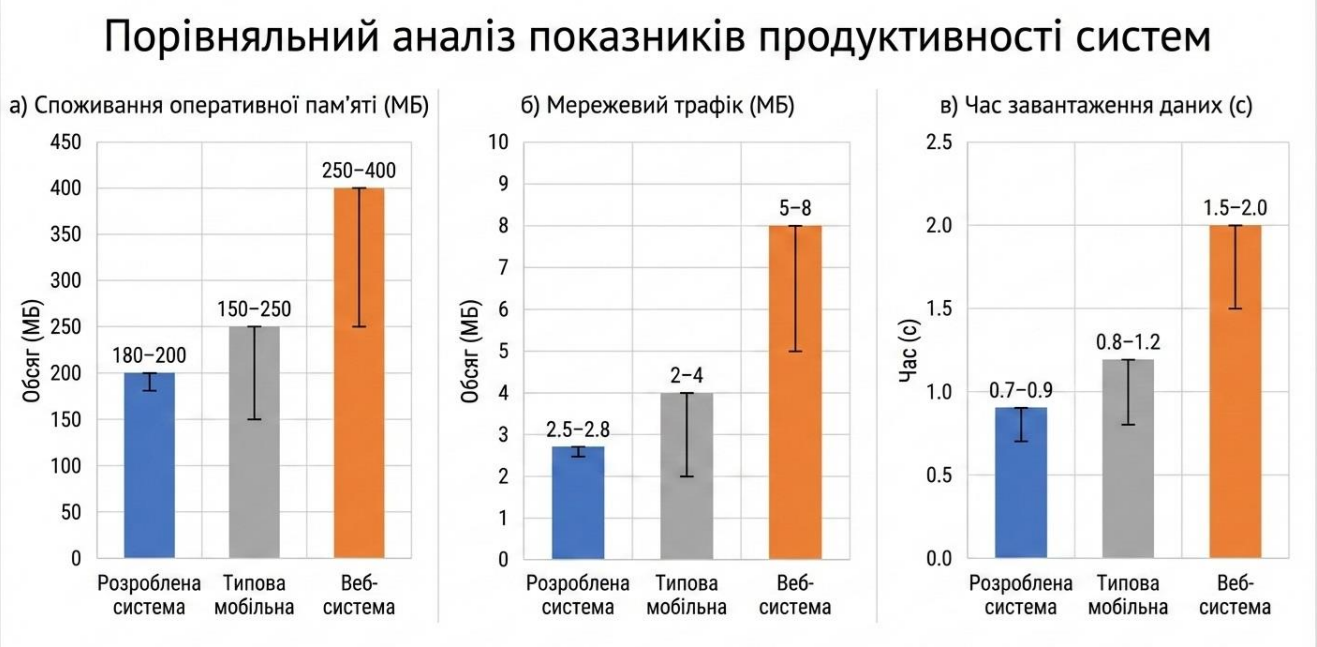


Рисунок 4.5 – Порівняльний аналіз показників продуктивності запропонованої системи

4.3 Висновки по розділу

За результатами проведених випробувань користувацького інтерфейсу та аналізу продуктивності системи можна зробити такі висновки. Інтерфейс застосунку є зручним та інтуїтивно зрозумілим: користувач може швидко ознайомитися з доступними моделями автомобілів, переглянути деталі комплектацій та відкрити прайс-листи у форматі PDF. Використання стеку фрагментів забезпечує коректну навігацію між екранами та стандартну роботу кнопки «Назад», що повертає користувача з екрану деталей на екран вибору

моделі.

Продуктивність застосунку відповідає сучасним вимогам для мобільних Android-застосунків з подібним функціоналом. Споживання оперативної пам’яті, мережевого трафіку та час завантаження даних знаходяться у межах типових значень, демонструючи ефективне використання ресурсів. Порівняння з браузерним доступом до того ж контенту підтверджує, що застосунок є забезпечує швидший та ресурсозберігаючий доступ до даних. Система працює стабільно, без збоїв та значних затримок, що підтверджує її готовність до практичного використання.

ВИСНОВКИ

У межах дипломної роботи було реалізовано клієнт-серверну комп’ютерну систему автосалону для перегляду асортименту автомобілів, орієнтовану на ефективний доступ користувачів до структурованих даних у мобільному середовищі. У процесі виконання роботи досягнуто поставленої мети підвищення ефективності доступу до інформації про автомобільні моделі та комплектації за рахунок використання спеціалізованого Android-застосунку замість універсальних веб-інтерфейсів.

У ході роботи проведено аналіз предметної області та існуючих рішень, на основі якого сформовано функціональні та нефункціональні вимоги до системи. Обґрунтовано вибір клієнт-серверної архітектури з централізованим сервером і мобільним клієнтом, визначено доцільність використання мови Java для обох компонентів, а також сокетної взаємодії та формату JSON для передавання даних. Це дозволило мінімізувати складність серверної частини та зменшити накладні витрати на мережеву взаємодію.

Реалізовано серверну частину у вигляді консольного Java-застосунку, який забезпечує обробку клієнтських запитів, роботу з базою даних та передавання агрегованих даних про автомобільні моделі у вигляді об’єктів Car. Сервер надає за один запит повний перелік моделей обраного бренду з описами, цінами та URL графічних і PDF-ресурсів, що дозволяє скоротити кількість мережевих звернень і спростити логіку клієнта. Спроєктовано структуру бази даних, яка забезпечує збереження брендів, моделей, комплектацій, цін і посилань на прайс-листи.

Розроблено клієнтську частину у вигляді Android-застосунку з використанням фрагментів і керуванням навігацією через стек зворотних викликів. Реалізовано вітальний екран, екран вибору моделей у вигляді каруселі та екран детальної інформації про автомобіль. Забезпечено асинхронне завантаження даних із сервера, коректну роботу стандартної кнопки «Назад», а також відкриття PDF-прайсів офіційного імпортера у зовнішньому браузері. Для

завантаження та кешування зображень використано бібліотеку Glide, що позитивно вплинуло на швидкодію та стабільність інтерфейсу.

У четвертому розділі виконано випробування користувацького інтерфейсу та аналіз продуктивності системи. За результатами профілювання встановлено, що при завантаженні повного набору з 9 об’єктів Car середнє споживання оперативної пам’яті становило 180–200 МБ, а після стабілізації роботи з фрагментами зменшувалося до 120–130 МБ. Час завантаження JSON-даних із сервера складав у середньому 700–900 мс, а мережевий трафік - 2,5–2,8 МБ, з яких близько 1,0 МБ припадало на структуровані дані. Продуктивність рендерингу інтерфейсу перебувала на рівні 50–55 FPS без помітних затримок.

Порівняння отриманих показників з результатами вимірів для веб-каталогів показало, що розроблена система є більш економною щодо використання оперативної пам’яті та мережевого трафіку.

Таким чином, у ході дипломної роботи розроблено працездатну клієнт-серверну систему, проведено її випробування та підтверджено ефективність обраних архітектурних. Отримані результати можуть бути використані як основа у сфері автомобільного ритейлу.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Volkswagen AG. Volkswagen official website. URL: https://www.volkswagen.ua/ (дата звернення: 07.01.2026).

2. Škoda Auto. Škoda official website. URL: https://www.skoda-auto.ua/ (дата звернення: 07.01.2026).

3. Audi AG. Audi official website. URL: https://www.audi.ua/ (дата звернення: 07.01.2026).

4. Toyota Motor Corporation. Toyota official website. URL: https://www.toyota.ua/ (дата звернення: 07.01.2026).

5. Farley D. Modern Software Engineering: Doing What Works to Build Better Software Faster. Boston : Addison-Wesley, 2021. 256 p.

6. Richards M., Ford N. Fundamentals of Software Architecture: An Engineering Approach. Sebastopol : O'Reilly Media, 2020. 419 p.

7. Sommerville I. Software Engineering. 11th ed. Harlow : Pearson Education Limited, 2020. 832 p.

8. Bass L., Clements P., Kazman R. Software Architecture in Practice. 4th ed. Boston : Addison-Wesley, 2021. 448 p.

9. Kleppmann M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. 1st ed. Sebastopol : O'Reilly Media, 2017. 616 p.

10. Android Developers. Guide to App Architecture. URL: https://developer.android.com/topic/architecture (дата звернення: 06.01.2026).

11. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Boston : Prentice Hall, 2017. 432 p.

12. Horstmann C. S. Core Java. Volume I: Fundamentals. 12th ed. Hoboken : Pearson Education, 2022. 912 p.

13. Geewax J. API Design Patterns. Shelter Island : Manning Publications, 2021. 480 p.

14. Prasad R., Sarma S. Beginning Spring Boot 3: Build Cloud-Native Stack: Web, Database, and Messenger Applications. 3rd ed. New York : Apress, 2023. 604 p.

15. Google. Gson. URL: https://github.com/google/gson (дата звернення: 07.01.2026).

16. Android Developers. Handler. URL: https://developer.android.com/reference/android/os/Handler (дата звернення: 07.01.2026).

17. MariaDB Foundation. MariaDB Server Documentation. URL: https://mariadb.org (дата звернення: 07.01.2026).

18. PostgreSQL Global Development Group. PostgreSQL Documentation. URL: https://www.postgresql.org/docs/ (дата звернення: 07.01.2026).

19. Oracle. JDBC Basics. URL: https://docs.oracle.com/javase/tutorial/jdbc/ (дата звернення: 07.01.2026).

20. Wong D. Real-World Cryptography. Shelter Island : Manning Publications, 2021. 392 p.

ДОДАТОК А
ЛІСТИНГИ

Лістинг А.1 – Фрагмент конфігураційного файлу первинного контенту

```
INSERT INTO brands (id, name, description) VALUES
(1, 'Škoda', 'Європейський виробник легкових автомобілів, що входить до групи Volkswagen');
INSERT INTO body_types (id, name) VALUES (1, 'Hatchback'), (2, 'Liftback'),(3, 'SUV');
INSERT INTO engines (id, name, fuel_type, horsepower, torque, displacement) VALUES (1, '1.0 TSI', 'Petrol', 110, 200, 1.0), (2, '1.5 TSI', 'Petrol', 150, 250, 1.5), (3, '2.0 TSI', 'Petrol', 190, 320, 2.0);
INSERT INTO transmissions (id, name, gears, type) VALUES (1, 'MQ200', 6, 'Manual'), (2, 'DQ200', 7, 'DSG'), (3, 'DQ381', 7, 'DSG');
INSERT INTO colors (id, name, code) VALUES (1, 'White', 'F9E9'), (2, 'Black', 'A1A1'), (3, 'Blue', '8X8X'), (4, 'Silver', 'C9C9');
INSERT INTO models (id, brand_id, name, body_type_id, engine_id, transmission_id, color_id, base_price, description, pdf_url) VALUES
(1, 1, 'Fabia', 1, 1, 1, 1, 720000, 'Компактний міський автомобіль', 'https://www.skoda-auto.ua/_doc/a1443d06-d3c9-4ea8-92cd-9ac0527f487c'),
(2, 1, 'Octavia', 2, 2, 2, 2, 980000, 'Сімейний автомобіль середнього класу', 'https://www.skoda-auto.ua/_doc/947bf628-936c-4bee-884d-27163a49056a'),
(3, 1, 'Kodiaq', 3, 3, 3, 3, 1450000, 'Повнорозмірний SUV', 'https://www.skoda-auto.ua/_doc/c49cd971-dab5-4c6c-af42-197ff09711ec');
INSERT INTO configurations (id, model_id, name, additional_price) VALUES -- Fabia
(1, 1, 'Essence', 0), (2, 1, 'Selection', 85000),
(3, 1, 'Sport', 165000), -- Octavia
(4, 2, 'Essence', 0), (5, 2, 'Selection', 110000),
(6, 2, 'Sport', 220000), -- Kodiaq
(7, 3, 'Essence', 0), (8, 3, 'Selection', 140000),
(9, 3, 'Sport', 290000);
INSERT INTO equipment (id, pr_code, description) VALUES
(1, '1AT', 'Система стабілізації ESP'),
(2, '7K1', 'Контроль тиску в шинах'),
(3, '8T0', 'Круїз-контроль'),
(4, '8W1', 'Денні ходові вогні'),
(5, '9WT', 'Базова мультимедійна система');
INSERT INTO equipment (id, pr_code, description) VALUES
(6, '7X1', 'Задній парктронік'),
(7, '6XK', 'Електроскладання дзеркал'),
(8, '9AK', 'Кліматконтроль Climatronic'),
(9, '8RM', 'Аудіосистема 8 динаміків'),
```

```
(10,'7Y1', 'Асистент утримання смуги');
INSERT INTO equipment (id, pr_code, description) VALUES
(11,'7Y8', 'Контроль сліпих зон'),
(12,'8R0', 'Навігаційна система'),
(13,'6Q2', 'Шкіряне кермо'),
(14,'8IT', 'LED Matrix фари'),
(15,'2PF', 'Спортивне шасі');
INSERT INTO configuration_equipment (configuration_id, equipment_id)
VALUES (1,1),(1,2),(1,3),(1,4),(1,5), (4,1),(4,2),(4,3),(4,4),(4,5),
(7,1),(7,2),(7,3),(7,4),(7,5);
INSERT INTO configuration_equipment (configuration_id, equipment_id)
VALUES (2,1),(2,2),(2,3),(2,4),(2,5),(2,6),(2,7),(2,8),(2,9),(2,10),
(5,1),(5,2),(5,3),(5,4),(5,5),(5,6),(5,7),(5,8),(5,9),(5,10),
(8,1),(8,2),(8,3),(8,4),(8,5),(8,6),(8,7),(8,8),(8,9),(8,10);
INSERT INTO configuration_equipment (configuration_id, equipment_id)
VALUES (3,1),(3,2),(3,3),(3,4),(3,5),(3,6),(3,7),(3,8),(3,9),(3,10),
(3,11),(3,12),(3,13),(3,14),(3,15),
(6,1),(6,2),(6,3),(6,4),(6,5),(6,6),(6,7),(6,8),(6,9),(6,10),
(6,11),(6,12),(6,13),(6,14),(6,15),
(9,1),(9,2),(9,3),(9,4),(9,5),(9,6),(9,7),(9,8),(9,9),(9,10),
(9,11),(9,12),(9,13),(9,14),(9,15);
```

Лістинг А.2 – Код методу getModelByID класу DB

```
public Model getModelById(int modelId) throws SQLException {
String sql = """SELECT
m.id AS model_id, m.name AS model_name, m.base_price, m.description
AS model_description, m.pdf_url, b.id AS brand_id, b.name AS
brand_name, b.description AS brand_description, bt.id AS
body_type_id, bt.name AS body_type_name, e.id AS engine_id, e.name
AS engine_name, e.fuel_type, e.horsepower, e.torque, e.displacement,
t.id AS transmission_id, t.name AS transmission_name,
t.gears, t.type AS transmission_type, c.id AS color_id,
c.name AS color_name, c.code AS color_code
FROM models m JOIN brands b ON m.brand_id = b.id
-- Зв'язок через проміжні таблиці:
LEFT JOIN model_body_types mbt ON m.id = mbt.model_id
LEFT JOIN body_types bt ON mbt.body_type_id = bt.id
LEFT JOIN model_engines me ON m.id = me.model_id
LEFT JOIN engines e ON me.engine_id = e.id
LEFT JOIN model_transmissions mt ON m.id = mt.model_id
LEFT JOIN transmissions t ON mt.transmission_id = t.id
LEFT JOIN model_colors mc ON m.id = mc.model_id
LEFT JOIN colors c ON mc.color_id = c.id
WHERE m.id = ?""";
try(PreparedStatement ps = connection.prepareStatement(sql)) {
ps.setInt(1, modelId);
try (ResultSet rs = ps.executeQuery()) {
if (!rs.next()) { return null;}
Brand brand = new Brand(
rs.getInt("brand_id"),
rs.getString("brand_name"),
rs.getString("brand_description"));
```

```
BodyType bodyType = new BodyType(
rs.getInt("body_type_id"),
rs.getString("body_type_name"));
Engine engine = new Engine(
rs.getInt("engine_id"),
rs.getString("engine_name"),
rs.getString("fuel_type"),
rs.getInt("power_hp"),
rs.getInt("torque_nm"),
rs.getDouble("displacement"));
Transmission transmission = new Transmission(
rs.getInt("transmission_id"),
rs.getString("transmission_name"),
rs.getInt("gears"),
rs.getString("transmission_type"));
Color color = new Color(rs.getInt("color_id"),
rs.getString("color_name"),
rs.getString("color_code"));
return new Model(
rs.getInt("model_id"), brand,
rs.getString("model_name"),
bodyType, engine,
transmission, color,
rs.getDouble("base_price"),
rs.getString("model_description"),
rs.getString("pdf_url"));}}}
```

Лістинг А.3 – Основний код класу CarouselFragment

```
public class CarouselFragment extends Fragment {
private List<Car> cars = new ArrayList<>();
private int index = 0;
private ImageView imageModel;
private TextView textName;
private TextView textPrice;
public void setCars(List<Car> cars) {
this.cars = cars;
if (getView() != null) {
show();
}
}
//…onCreateView
private void show() {
if (cars.isEmpty()) return;
Model model = cars.get(index).getModel();
textName.setText(model.getName());
textPrice.setText("від " + model.getBase_price() + " грн");
String url = getImageUrl(model.getName());
if (!url.isEmpty()) {
Glide.with(this)
.load(url)
.into(imageModel);
}
```

```
}
private String getImageUrl(String modelName) {
switch (modelName) {
case "Fabia":
return "https://cdn.skoda-
auto.com/images/external/b06897e0d7df0c208855b5853c632df7/Derivative
IntroductionModule/9991858888551adc2e23c7b2ad2a8a3853f5cd04de3d16320
a15b71b2c9e83c8/Default_bp1200_1.webp";
case "Octavia":
return "https://cdn.skoda-
auto.com/images/external/2aadb6009d86d0e3c2aa26d82837b2e0/Derivative
IntroductionModule/9991858888551adc2e23c7b2ad2a8a3853f5cd04de3d16320
a15b71b2c9e83c8/Default_bp1200_1.webp";
case "Kodiaq":
return "https://cdn.skoda-
auto.com/images/external/9e0ad990f55ee219467ed87ace308414/Derivative
IntroductionModule/9991858888551adc2e23c7b2ad2a8a3853f5cd04de3d16320
a15b71b2c9e83c8/Default_bp1200_1.webp";
default:
return "";
}}}
```

Лістинг А.4 – Основний код DetailFragment

```
public class DetailFragment extends Fragment {
private Car car;
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup
container, Bundle savedInstanceState) {
View view = inflater.inflate(R.layout.fragment_detail, container,
false);
if (car == null) return view;
Model model = car.getModel();
ImageView image = view.findViewById(R.id.image_detail_model);
TextView title = view.findViewById(R.id.text_detail_name);
TextView description =
view.findViewById(R.id.text_detail_description);
TextView configurationsView =
view.findViewById(R.id.text_configurations);
title.setText(model.getName());
description.setText(model.getDescription());
String url = getImageUrl(model.getName());
if (!url.isEmpty()) {
Glide.with(this)
.load(url)
.into(image);}
Configuration configuration = car.getConfiguration();
StringBuilder builder = new StringBuilder();
builder.append("Комплектація: ")
.append(configuration.getName()).append("\n");
builder.append("Додаткова вартість: +")
.append((int) configuration.getAdditionalPrice())
.append(" грн\n\n");
```

```
builder.append("Обладнання:\n");
for (Equipment eq : configuration.getEquipment()) {
builder.append("• ")
.append(eq.getDescription())
.append("\n");
}
configurationsView.setText(builder.toString());
view.findViewById(R.id.button_open_pdf).setOnClickListener(v -> {
Intent intent = new Intent(Intent.ACTION_VIEW,
Uri.parse(model.getPdf_url()));
startActivity(intent);
});
return view;}
}
```

ДОДАТОК Б
ЛІСТИНГИ ВЕРСТКИ ЕКРАНІВ КЛІЄНТСЬКОГО ЗАСТОСУНКУ

Лістинг Б.1 – Файл activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
xmlns:android="http://schemas.android.com/apk/res/android"
android:id="@+id/fragment_container"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:background="#E6F2EA" />
```

Лістинг Б.2 – Файл fragment_welcome.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
android:id="@+id/welcome_root"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:orientation="vertical"
android:gravity="center"
android:padding="24dp"
android:background="#E6F2EA">
<TextView
android:id="@+id/text_welcome"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="Ласкаво просимо до каталогу автомобілів"
android:textSize="20sp"
android:textColor="#1B5E20"
android:gravity="center"
android:layout_marginBottom="32dp" />
<Button
android:id="@+id/button_start"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="Почати"
android:backgroundTint="#2E7D32"
android:textColor="#FFFFFF" />
</LinearLayout>
```

Лістинг Б.3 – Файл fragment_carousel.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
```

```
android:id="@+id/carousel_root"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:orientation="vertical"
android:padding="16dp"
android:background="#E6F2EA">
<LinearLayout
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:orientation="horizontal"
android:gravity="center_vertical">
<Button
android:id="@+id/button_prev"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="Prev" />
<LinearLayout
android:layout_width="0dp"
android:layout_height="wrap_content"
android:layout_weight="1"
android:orientation="vertical"
android:gravity="center"
android:padding="12dp">
<ImageView
android:id="@+id/image_model"
android:layout_width="250dp"
android:layout_height="150dp"
android:scaleType="fitCenter"
android:contentDescription="Зображення моделі" />
<TextView
android:id="@+id/text_model_name"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:textSize="18sp"
android:textColor="#1B5E20"
android:layout_marginTop="12dp" />
<TextView
android:id="@+id/text_model_price"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:textSize="16sp"
android:textColor="#2E7D32"
android:layout_marginTop="8dp" />
</LinearLayout>
<Button
android:id="@+id/button_next"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="Next" />
</LinearLayout>
</LinearLayout>
```

Лістинг Б.4 – Файл fragment_detail.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
android:id="@+id/detail_root"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:orientation="vertical"
android:padding="16dp"
android:background="#E6F2EA">
<ImageView
android:id="@+id/image_detail_model"
android:layout_width="match_parent"
android:layout_height="180dp"
android:scaleType="fitCenter"
android:contentDescription="Зображення моделі" />
<TextView
android:id="@+id/text_detail_name"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:textSize="20sp"
android:textColor="#1B5E20"
android:layout_marginTop="12dp" />
<TextView
android:id="@+id/text_detail_description"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:textSize="14sp"
android:textColor="#263238"
android:layout_marginTop="8dp" />
<TextView
android:id="@+id/text_configurations"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:textSize="14sp"
android:textColor="#37474F"
android:layout_marginTop="16dp" />
<Button
android:id="@+id/button_open_pdf"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="Відкрити прайс-лист"
android:backgroundTint="#2E7D32"
android:textColor="#FFFFFF"
android:layout_marginTop="24dp"
android:layout_gravity="center"/>
</LinearLayout>
```

ДОДАТОК В
СЛАЙДИ ПРЕЗЕНТАЦІЇ

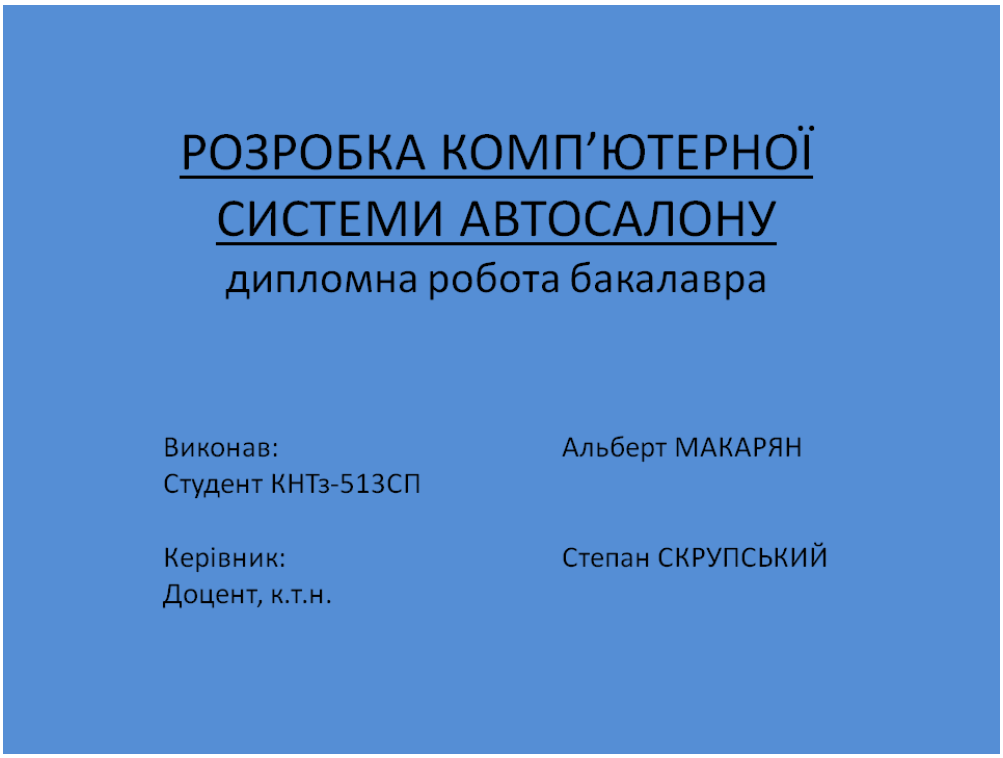


Рисунок В.1 – Слайд 1

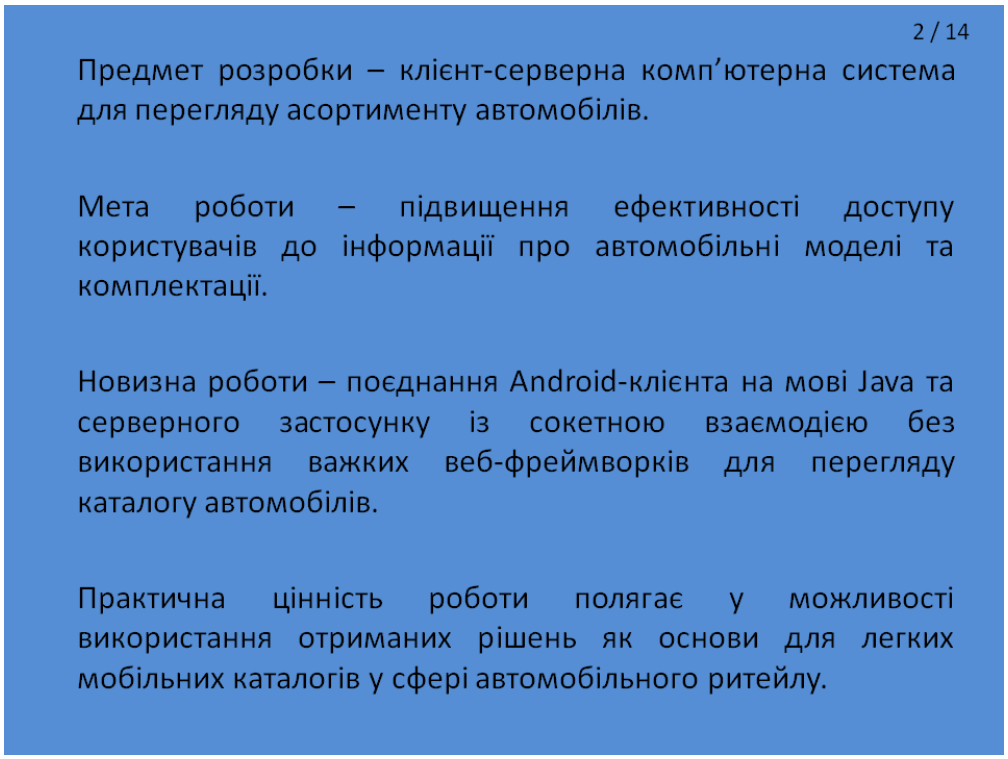


Рисунок В.2 – Слайд 2

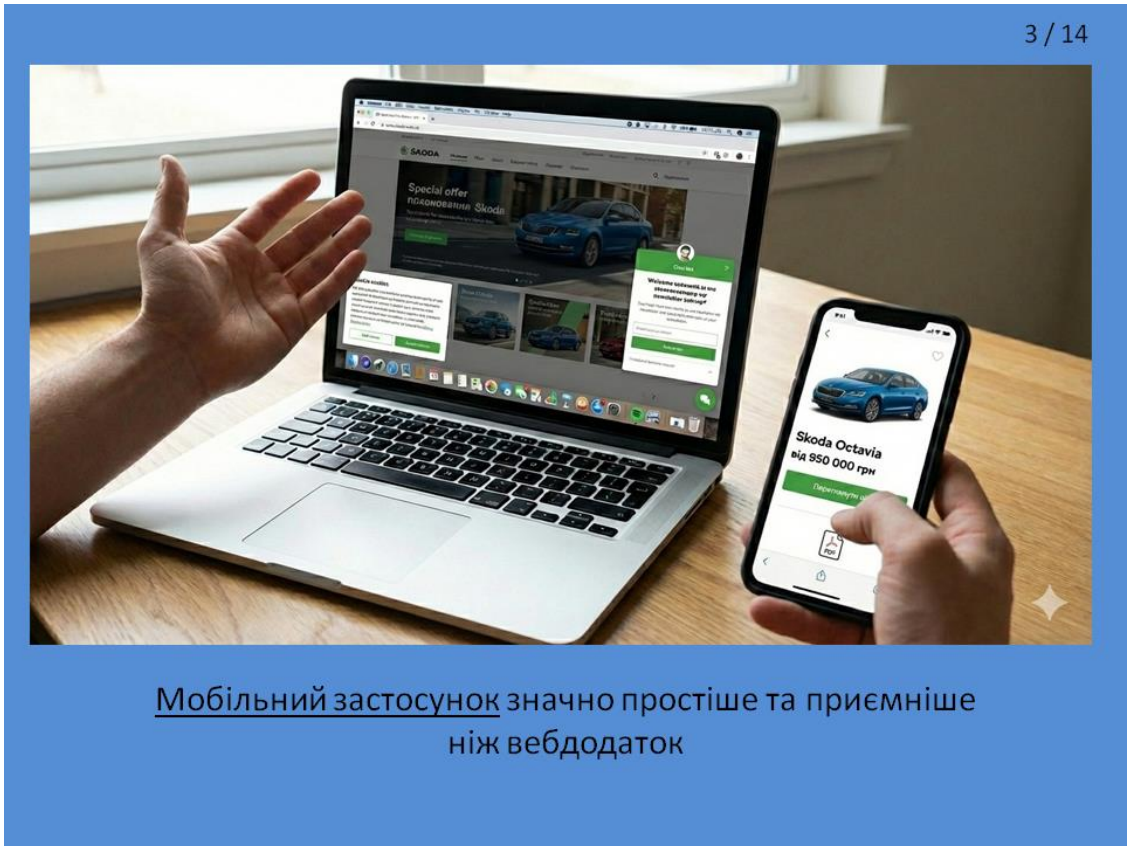


Рисунок В.3 – Слайд 3

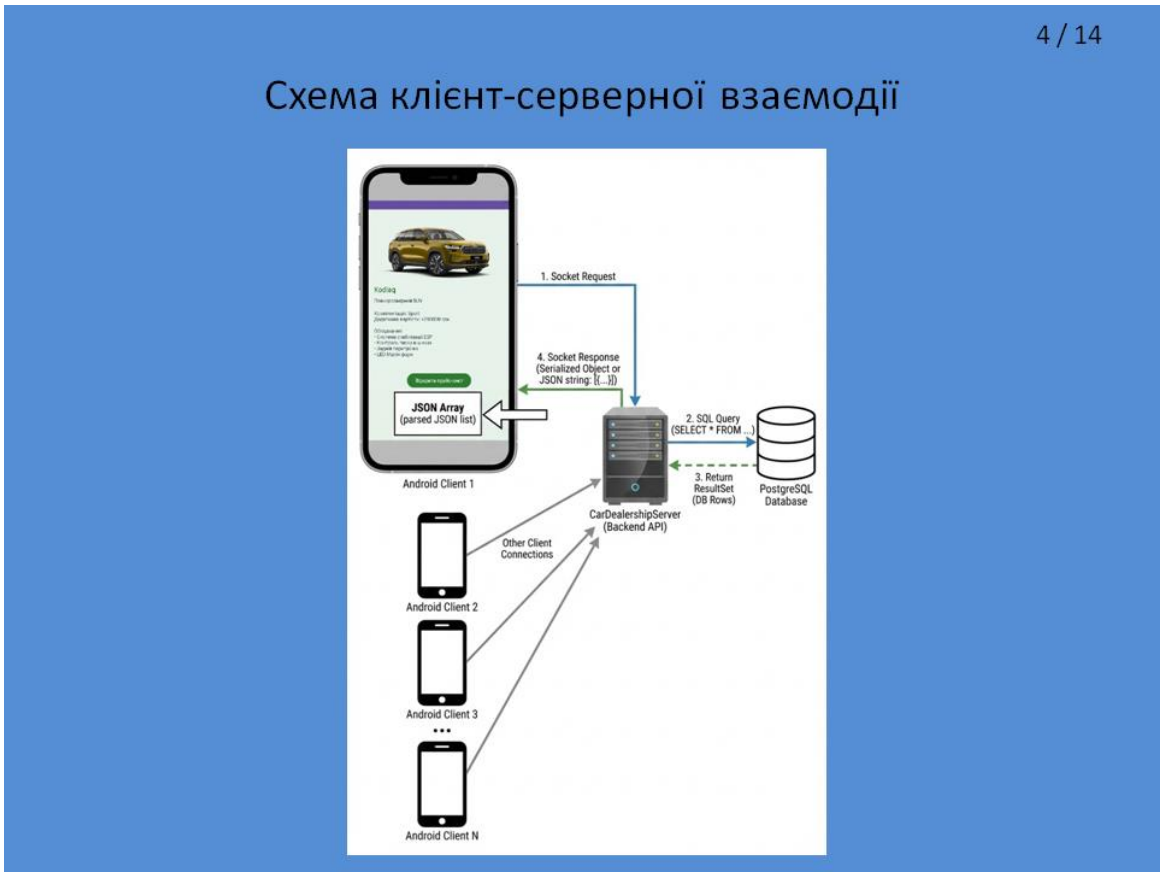


Рисунок В.4 – Слайд 4

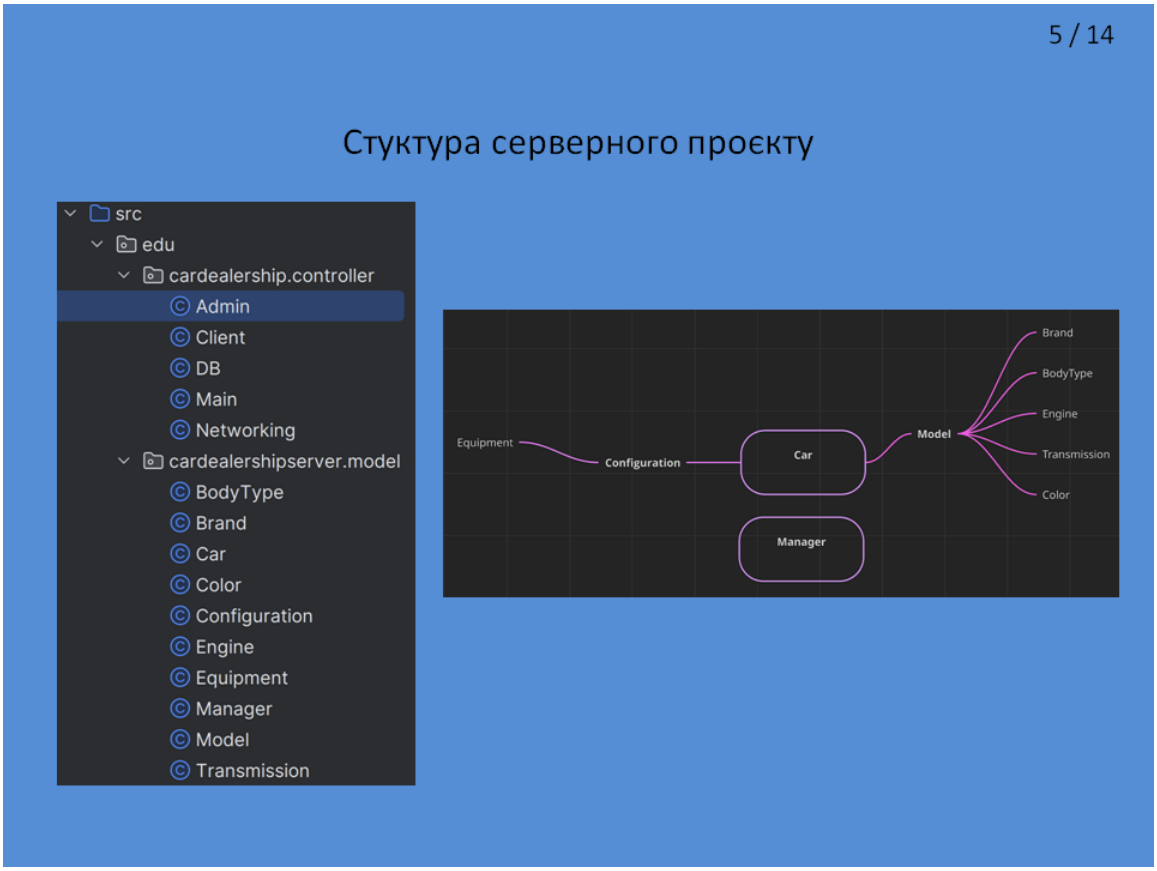


Рисунок В.5 – Слайд 5

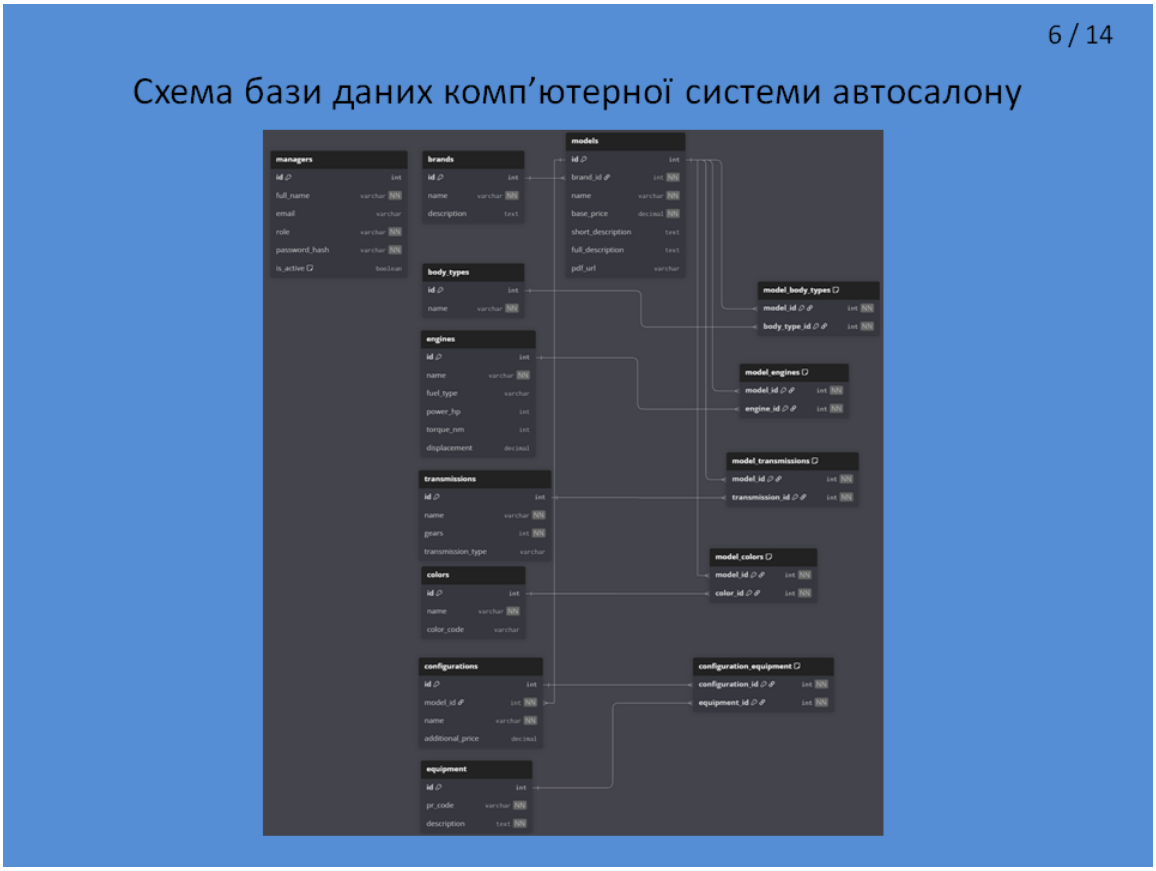


Рисунок В.6 – Слайд 6

Рисунок В.7 – Слайд 7

Рисунок В.8 – Слайд 8

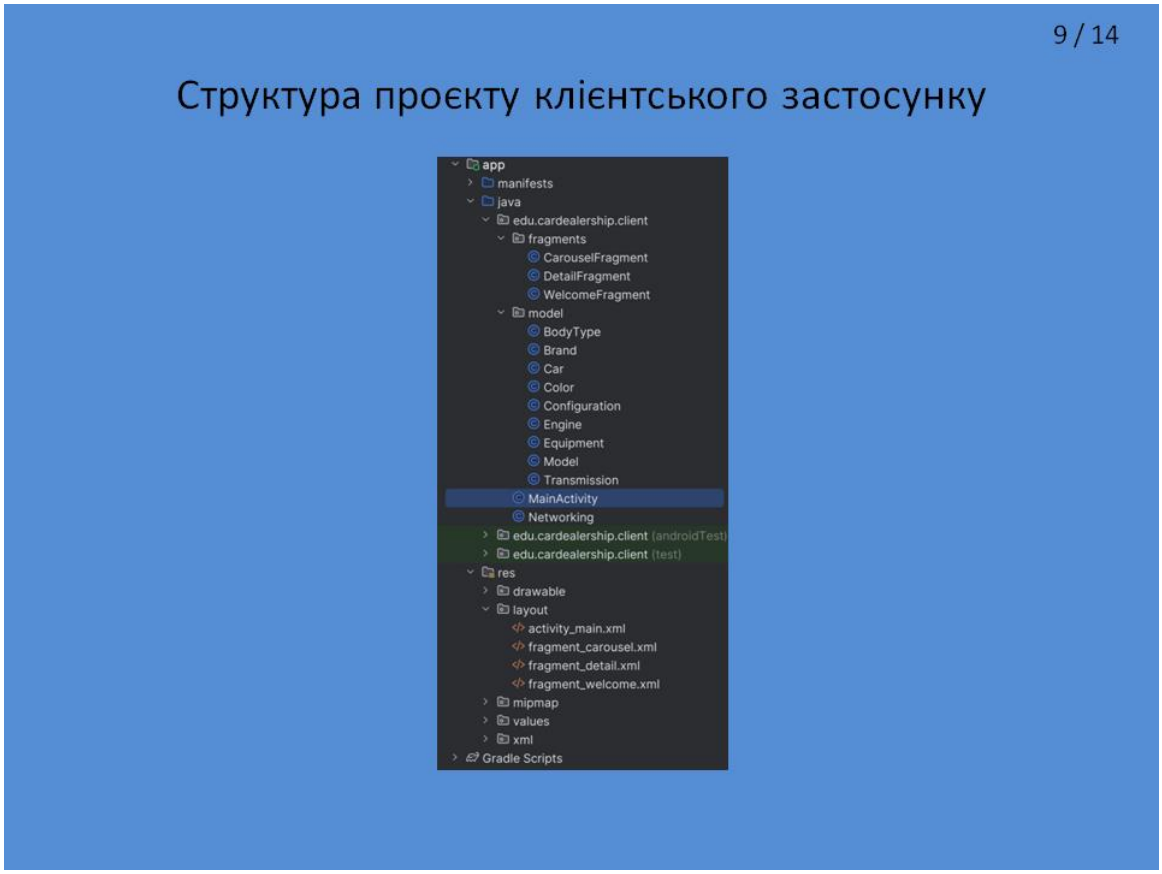


Рисунок В.9 – Слайд 9

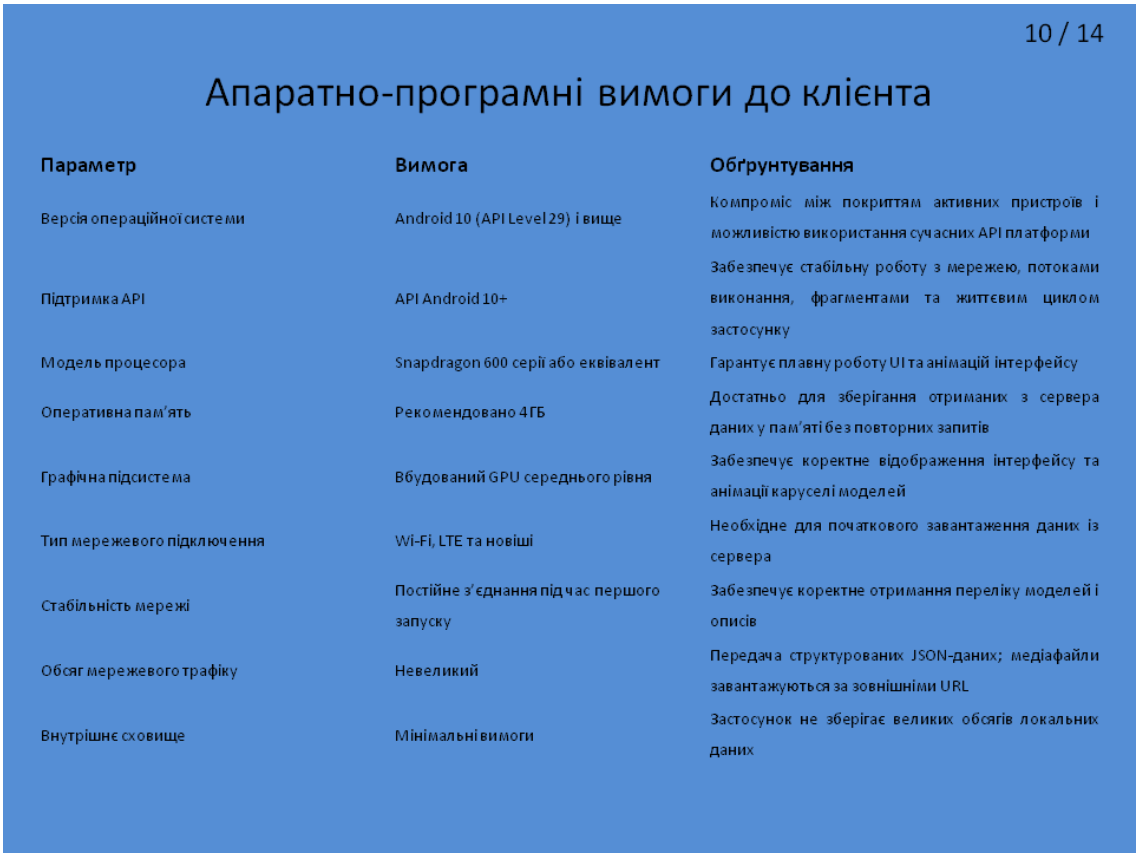


Рисунок В.10 – Слайд 10

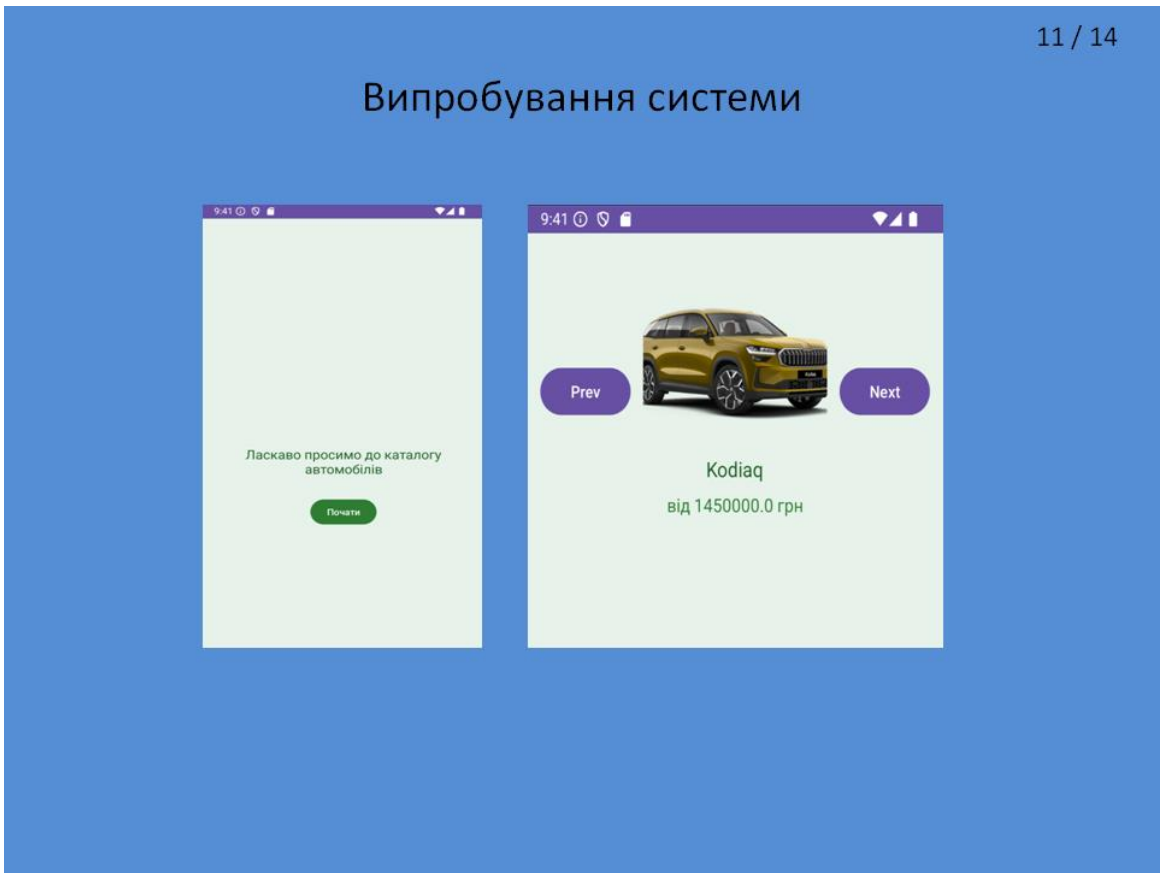


Рисунок В.11 – Слайд 11

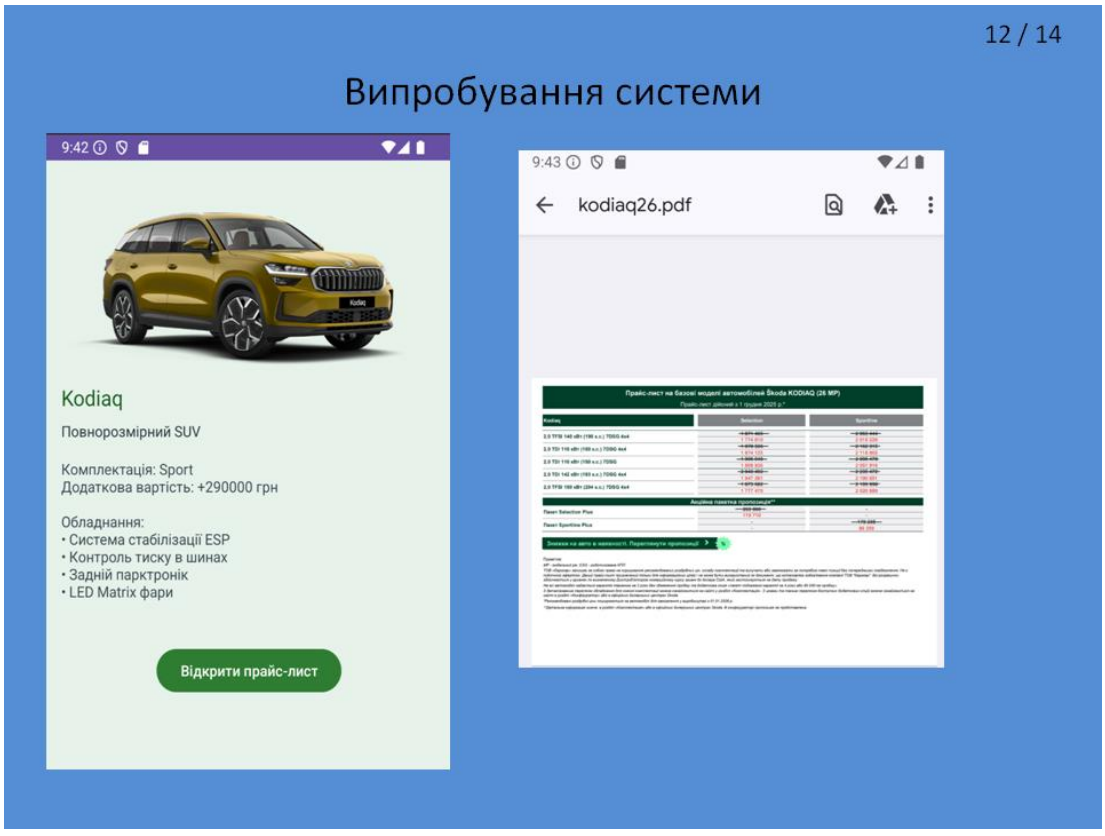


Рисунок В.12 – Слайд 12

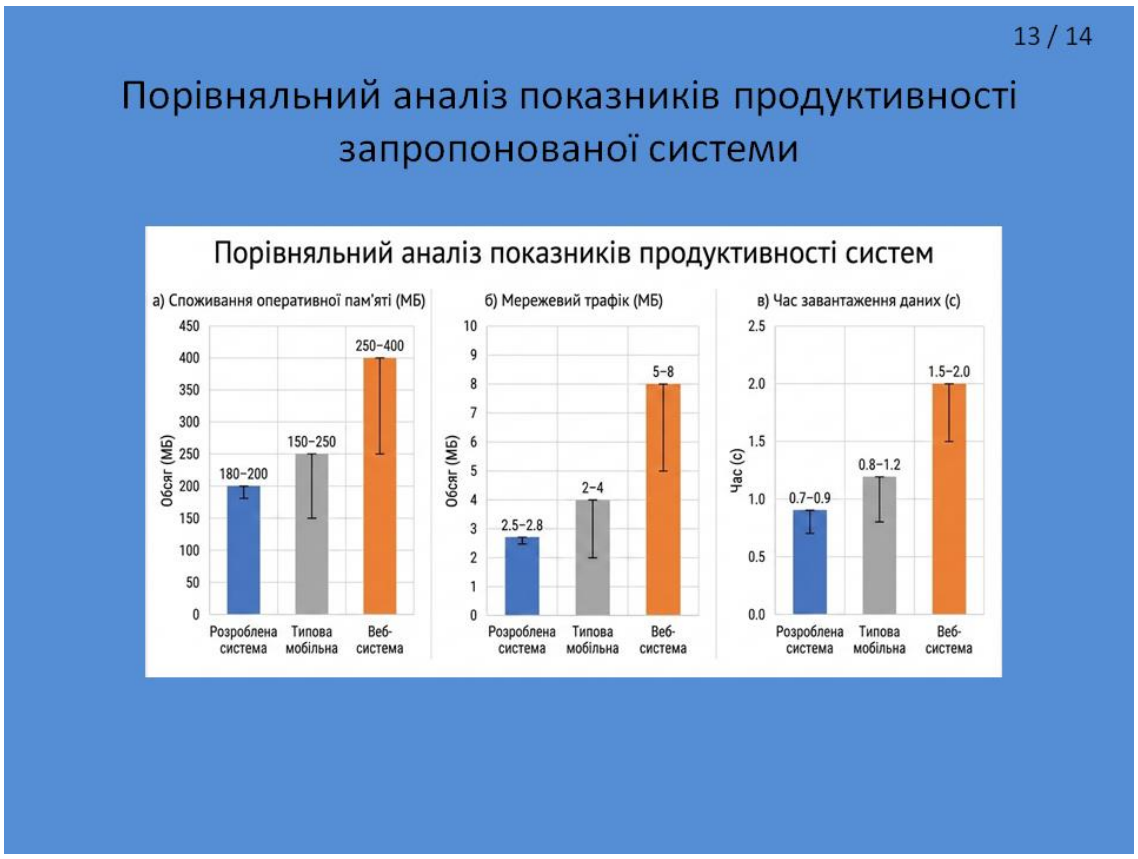


Рисунок В.13 – Слайд 13

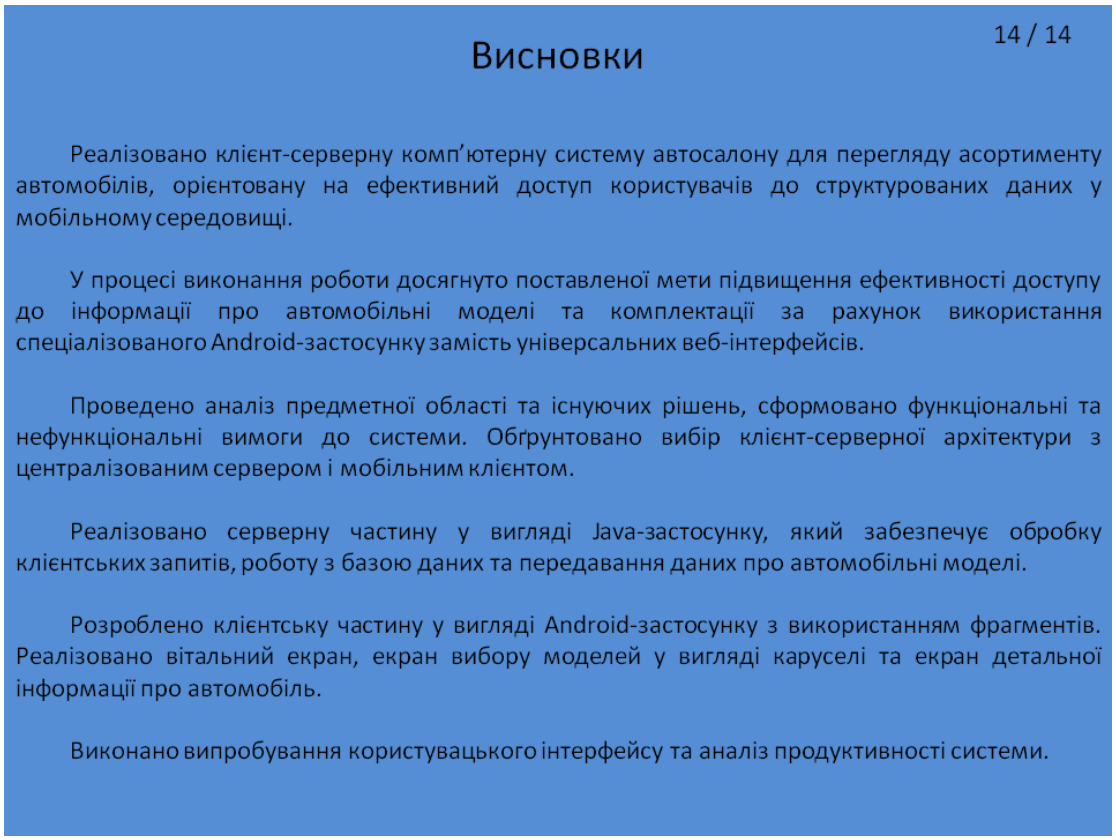


Рисунок В.14 – Слайд 14