ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка застосунку для організації спільних поїздок мешканців житлового комплексу»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Олександр ЗАЗГАРСЬКИЙ
(підпис)

Виконав: здобувач вищої освіти групи ПД-43
Олександр ЗАЗГАРСЬКИЙ

Керівник: Юрій ЗАДОНЦЕВ
к.т.н.

Рецензент: _____

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення
Ступінь вищої освіти Бакалавр
Спеціальність 121 Інженерія програмного забезпечення
Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
« ____ » _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Зазгарському Олександру Віталійовичу

1.Тема кваліфікаційної роботи: «Розробка застосунку для організації спільних поїздок мешканців житлового комплексу»
керівник кваліфікаційної роботи к.т.н. Юрій ЗАДОНЦЕВ,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «24» лютого 2025 р. № 56.

2. Строк подання кваліфікаційної роботи «02» червня 2025 р.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про принципи ride-sharing і цифрову взаємодію в локальних спільнотах; результати аналізу сучасних веб-сервісів для організації спільних поїздок; емпіричні дані, зібрані шляхом анкетування мешканців житлового комплексу; технічна документація до використаних технологій (Flask, JavaScript, MySQL); приклади реалізації подібних систем; вимоги користувачів до функціоналу застосунку.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Аналіз сервісів ride-sharing та локальних цифрових платформ у контексті організації спільних поїздок.

2. Проєктування клієнт-серверної архітектури веб-застосунку для координації поїздок мешканців житлового комплексу.

3 Програмна реалізація та опис функціонування основних модулів: реєстрація, створення маршрутів, перегляд поїздок, особисті профілі.
4 Тестування застосунку: модульне, інтеграційне, UX-тестування.

5. Перелік графічного матеріалу: *презентація*

1. Аналіз аналогів.
2. Вимоги до програмного забезпечення.
3. Програмні засоби реалізації.
4. Системна архітектура веб-додатку.
5. Екранні форми
6. Демонстрація роботи програми
7. Апробація результатів дослідження

6. Дата видачі завдання «25» лютого 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	25.02.-04.03.2025	
2	Аналіз та дослідження існуючих аналогів	05.03-13.03.2025	
3	Формування функціональних і нефункціональних вимог до системи організації спільних поїздок	14.03–21.03.2025	
4	Проєктування клієнт-серверної архітектури застосунку та структури бази даних	22.03–03.04.2025	
5	Реалізація основних модулів системи: реєстрація, створення маршрутів, профілі користувачів	04.04–15.04.2025	
6	Проведення модульного, інтеграційного та UX-тестування веб-застосунку	16.04–28.04.2025	
7	Оформлення роботи: вступ, висновки, реферат	29.04-11.05.2025	
8	Розробка демонстраційних матеріалів	12.05-18.05.2025	
9	Попередній захист роботи	19.05-30.05.2025	

Здобувач вищої освіти _____ Олександр ЗАЗГАРСЬКИЙ
(підпис)

Керівник кваліфікаційної роботи _____ Юрій ЗАДОНЦЕВ
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 93 стор., 3 табл., 3 рис., 35 джерел.

Мета роботи – розробка програмного забезпечення для покращення взаємодії мешканців житлового комплексу шляхом використання принципів клієнт-серверної архітектури, модульного програмування та REST API. Також для спрощення організації спільних поїздок, покращення логістики та взаємодії між користувачами.

Об'єкт дослідження – процес організації та координації поїздок серед мешканців житлового комплексу з використанням цифрових засобів. Досліджується спосіб побудови системи, яка забезпечує створення, пошук і управління спільними поїздкама.

Предмет дослідження – вебтехнології, зокрема клієнт-серверна архітектура, REST API, HTML/CSS/JS, взаємодія з базою даних, авторизація через JWT, а також засоби побудови інтерфейсу користувача.

Короткий зміст роботи: У роботі розглянуто проблему організації спільних поїздок мешканців житлових комплексів із метою підвищення зручності щоденних пересувань та зменшення витрат. Проведено аналіз існуючих сервісів ride-sharing, виявлено їх переваги й недоліки, а також сформульовано вимоги до функціоналу локального застосунку. Реалізовано клієнт-серверну архітектуру на основі Flask, JavaScript і MySQL, яка охоплює реєстрацію користувачів, створення поїздок, фільтрацію за параметрами, перегляд водіїв та комунікацію між учасниками. Застосунок має зручний інтерфейс і передбачає базову модерацію вмісту. Проведено тестування основних модулів системи, що підтвердило стабільність її роботи та відповідність поставленим вимогам.

Сферою використання застосунку є локальні спільноти мешканців житлових комплексів, гуртожитків або приватних районів, які зацікавлені в оптимізації щоденних поїздок шляхом координації спільних маршрутів.

КЛЮЧОВІ СЛОВА: СПІЛЬНІ ПОЇЗДКИ, RIDE-SHARING, ВЕБ-ЗАСТОСУНОК, ЛОКАЛЬНА СПІЛЬНОТА, FLASK, MYSQL, JAVASCRIPT, МАРШРУТИ, КОРИСТУВАЧ, ІНТЕРФЕЙС.

ЗМІСТ

ВСТУП

У сучасних умовах активного розвитку цифрових технологій та урбанізації значна увага приділяється оптимізації повсякденних процесів життєдіяльності мешканців міст. Зростання кількості автомобілів, розширення житлових комплексів і підвищення мобільності населення призводить до потреби в раціональнішому використанні транспортних ресурсів та покращенні логістики на локальному рівні. Особливо актуальним стає створення інструментів, які сприяють не лише ефективному пересуванню, а й формуванню локальної спільноти та взаємодії між мешканцями. Саме у цьому контексті розробка цифрових сервісів для організації спільних поїздок (так званих ride-sharing рішень) набуває особливої значущості. Це явище стає частиною ширшої концепції сталого транспорту, який передбачає не лише екологічну безпеку, але й економічну доцільність, зниження навантаження на інфраструктуру та соціальну згуртованість.

Актуальність теми дослідження зумовлена кількома ключовими чинниками. По-перше, в Україні досі спостерігається недостатня кількість локалізованих IT-продуктів, спрямованих на конкретні мікроспільноти, зокрема мешканців окремих житлових комплексів. По-друге, відсутність гнучких та доступних засобів організації спільних поїздок створює щоденні незручності: нераціональне використання приватного транспорту, затори, надмірні витрати пального та часу. По-третє, в контексті сучасних викликів — як-от енергетична нестабільність, безпекові обмеження в умовах воєнного стану чи карантинних заходів — подібні сервіси стають важливими не лише для зручності, а й для виживання окремих спільнот. Отже, створення застосунку, який би дозволив мешканцям об'єднуватися задля спільного користування транспортом, вирішує не лише побутові, а й суспільно важливі задачі.

Метою роботи є розробка повнофункціонального веб-застосунку для організації спільних поїздок серед мешканців житлового комплексу, що забезпечить реєстрацію, створення маршрутів, приєднання до поїздок, редагування профілю користувача та зручну візуалізацію інформації у дружньому інтерфейсі.

Основними завданнями дослідження є:

- Провести аналіз сучасних аналогічних сервісів в Україні та за кордоном;
- Визначити потреби цільової аудиторії на основі збору емпіричних даних;
- Сформулювати функціональні вимоги до системи;
- Розробити архітектуру програмного забезпечення;
- Обґрунтувати вибір технологічного стеку;
- Реалізувати фронтенд і бекенд частини застосунку;
- Провести тестування функціональності;
- Визначити напрями подальшого вдосконалення системи.

Об’єктом дослідження виступає процес цифрової організації транспортної взаємодії між мешканцями локальних спільнот.

Предметом дослідження є структурна, функціональна та інтерфейсна реалізація веб-застосунку для організації спільних поїздок.

Джерельна база дослідження включає:

- наукові публікації у сфері розробки інформаційних систем;
- статті, присвячені ride-sharing технологіям (BlaBlaCar, Lyft, Uber);
- документацію з роботи з Flask, RESTful API, MySQL, JavaScript;
- досвід локальних ініціатив у межах житлових кооперативів;
- матеріали форумів та обговорень серед IT-фахівців і користувачів.

Фактичний матеріал представлений результатами моделювання, розробки та тестування веб-застосунку. В роботі використовуються дані, отримані під час спостережень за організацією транспорту в окремому житловому комплексі, а також відповіді респондентів (мешканців), які надали пропозиції щодо бажаного функціоналу.

Методи дослідження включають:

- метод порівняльного аналізу для вивчення існуючих аналогів;
- моделювання — для побудови структури застосунку;
- системний підхід — для побудови взаємозв'язків між модулями;
- програмна реалізація (метод інженерного прототипування);
- емпіричні методи — анкетування та опитування користувачів;
- тестування методом «чорної скриньки» та логічна верифікація.

Наукова новизна роботи полягає в тому, що:

- **вперше визначено** специфіку локального ride-sharing у межах мікроспільноти мешканців житлового комплексу;
- **вдосконалено** підхід до побудови веб-застосунку через інтеграцію простих RESTful API з динамічним інтерфейсом без фреймворків;
- **дістало подальший розвиток** застосування моделі мінімального життєздатного продукту (MVP) у побудові транспортних платформ;
- **розроблено нову концепцію** спільного використання приватного транспорту з урахуванням обмеженого географічного охоплення;
- **створено унікальну систему** взаємодії мешканців через веб-застосунок із простою авторизацією, налаштуванням маршруту та логікою приєднання;
- реалізація базується **на відомому принципі** client-server взаємодії з поділом логіки, що забезпечує масштабованість і гнучкість.

Теоретична цінність роботи полягає у формалізації підходу до створення сервісів на базі Flask без використання складних фреймворків типу Django або Angular. Представлена модель реалізації може бути використана як приклад у навчальних курсах із розробки інформаційних систем, веб-програмування або IT-проєктування. Робота також демонструє ефективне поєднання процедурного підходу до програмування з сучасними вимогами до UX/UI.

Практичне значення роботи полягає у можливості реального впровадження розробленого застосунку в межах конкретного житлового комплексу. У результаті мешканці отримають зручний інструмент для оптимізації пересування, що зменшить кількість пустих поїздок, сприятиме економії коштів та пального, а також створенню соціально згуртованого середовища. Окрім того, систему можна масштабувати на інші житлові райони або навіть на корпоративні спільноти.

Апробація результатів роботи відбувалася під час демонстрацій студентських IT-інкубаторів, а також у рамках тестування прототипу серед мешканців одного з житлових комплексів у місті. Окремі елементи реалізації застосовувалися як практичні кейси у навчальному процесі для ознайомлення студентів із принципами API-розробки, реалізацією системи авторизації та клієнт-серверної взаємодії.

У підсумку, дана робота спрямована на вирішення актуального прикладного завдання із розробки цифрового інструменту, який задовольняє реальні запити мешканців міського середовища. Вона поєднує в собі інженерні, програмні, організаційні та соціальні підходи, що підвищує її міждисциплінарну цінність та відкриває можливості для подальших досліджень у сфері локальних цифрових сервісів.

1. АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ТА ОБҐРУНТУВАННЯ ПОТРЕБИ У ЗАСТОСУНКУ

1.1. Аналіз сучасних сервісів для організації поїздок

Світовий досвід у сфері організації спільних поїздок демонструє значний рівень розвитку цифрових сервісів, орієнтованих на ride-sharing, carpooling і peer-to-peer транспортні платформи. Серед найбільш впливових міжнародних аналогів варто виділити такі платформи, як Uber, Lyft, BlaBlaCar, Via, Gett, Bolt, Karos, Covoiturage-libre, Zimride, GoCarma, Waze Carpool та інші, кожна з яких реалізує унікальні механізми взаємодії між пасажирами й водіями, а також пропонує специфічні технологічні та бізнес-моделі. Uber та Lyft є провідними платформами у США та низці країн світу, що працюють на основі принципу on-demand ride-hailing, тобто миттєвого виклику водія через мобільний застосунок, з оплатою за поїздку та рейтинговою системою. Ці сервіси значною мірою орієнтовані на комерційні поїздки, водії проходять верифікацію, користуються додатком як засобом заробітку, а система розрахунків базується на динамічному тарифі, що враховує попит, час доби, трафік і відстань. Хоча Uber Pool і Lyft Shared дозволяють пасажирам ділити поїздку з іншими, це відбувається за алгоритмами автоматичної маршрутизації та не передбачає довготривалої взаємодії в межах локальної спільноти. BlaBlaCar, натомість, є класичним прикладом європейського carpooling-сервісу, який пропонує платформу для організації попередньо запланованих поїздок між містами. Цей сервіс не є комерційним у прямому сенсі, оскільки пасажири компенсують лише частину витрат водія, а сама платформа виступає посередником між приватними особами. Унікальною рисою BlaBlaCar є фокус на попередньому плануванні, прозорості маршруту, узгодженні деталей подорожі, з урахуванням кількості вільних місць, музичних уподобань, дозволу на розмови під

час поїздки тощо. Такий підхід сприяє підвищенню соціальної довіри між учасниками поїздок, хоча і менш ефективний у міських умовах або для коротких відстаней. Французька платформа Karos спеціалізується на щоденних поїздках до роботи, синхронізуючи графіки користувачів на основі мобільних даних та використовуючи інтелектуальне парування для побудови оптимальних маршрутів. Вона інтегрується з корпоративними програмами сталого транспорту, пропонуючи винагороди для працівників, які обирають спільні поїздки замість приватного транспорту. Подібний підхід втілюється і в сервісах GoCarma та Scoop у США, де цифрова система оцінює ефективність поїздок, надає доступ до прискорених смуг руху або стимулює карпулінг через бонуси. Водночас платформи як Zimride, які згодом були інтегровані до Lyft, створювали спільноти на рівні університетів або компаній, дозволяючи студентам і працівникам планувати спільні поїздки до кампусів або офісів. Це частково перегукується з концепцією локального ride-sharing, яку покладено в основу дослідження.

Ще одним прикладом є Via — сервіс, який поєднує гнучкість таксі з економічністю громадського транспорту. Його алгоритми дозволяють кільком користувачам, які мають схожі пункти призначення, автоматично формувати групу пасажирів із маршрутами, що перекриваються, використовуючи динамічні точки посадки та висадки. Такий підхід особливо ефективний у мегаполісах, де надмірне навантаження на інфраструктуру вимагає гнучких транспортних рішень. Деякі сервіси, як наприклад Waze Carpool, працюють на основі навігаційних даних: водії, які вже мають маршрут, можуть запропонувати підвезення попутників, а система порівнює збіги траєкторій у реальному часі. Хоча цей підхід мінімізує затрати на сторонні сервіси, він потребує високого рівня довіри між учасниками й наявності додаткових механізмів перевірки користувачів. Усі ці міжнародні рішення мають спільні характеристики — вони ґрунтуються на цифровій платформі, використовують геолокацію, алгоритмічне парування, системи оцінювання та повідомлень, забезпечують гнучку маршрутизацію, різні рівні конфіденційності, а

також підтримку з боку користувачів. Однак існують і суттєві відмінності. Насамперед, більшість таких сервісів орієнтовані на широкий ринок, без фокусування на конкретних житлових комплексах, де є сталий контингент користувачів. Це обмежує можливість формування стабільної взаємодії, яку можна забезпечити у межах мікроспільноти. Також в Україні через регуляторні бар'єри та відсутність платіжних систем із підтримкою малих трансакцій деякі функціональні моделі не мають прямого застосування. Враховуючи викладене, можна констатувати, що міжнародний досвід є надзвичайно корисним для формування підходу до створення локального застосунку, проте потребує адаптації до національного контексту, рівня цифрової грамотності, доступності інтернету, правових умов і специфіки соціальної поведінки. Сервіс, розроблений у межах даної роботи, поєднує найкращі практики глобальних аналогів із локальними особливостями — фокус на житловий комплекс як мікроосередок спільноти, використання спрощеної моделі взаємодії без необхідності грошових трансакцій, інтуїтивний інтерфейс, розроблений для непрофесійних користувачів, а також мінімалістичну архітектуру з урахуванням обмежених технічних ресурсів на локальному рівні. У підсумку, аналіз міжнародних аналогів демонструє, що попри різноманіття функціональних моделей, найуспішніші сервіси ті, що враховують специфіку користувацького середовища, забезпечують зручність, довіру та ефективність. Саме ці принципи покладено в основу підходу до розробки вітчизняного веб-застосунку, спрямованого на забезпечення якісної транспортної взаємодії у межах житлового комплексу.

Український ринок цифрових сервісів для організації спільних поїздок знаходиться на етапі активного формування, при цьому спостерігається поступове зростання інтересу до ride-sharing моделей не лише серед індивідуальних користувачів, а й на рівні корпоративного сектору. Водночас, на відміну від глобальних рішень, українські платформи здебільшого функціонують у спрощеному форматі або мають обмежений функціонал. До найбільш відомих

вітчизняних або локалізованих сервісів, що пропонують функції спільних поїздок, можна віднести BlaBlaCar Ukraine, який є адаптованою версією французької платформи; Uklon, який насамперед функціонує як таксі-сервіс, але має потенціал до впровадження ride-sharing функцій; а також низку менш масштабних сервісів, що працюють у межах Telegram-ботів, Facebook-груп та корпоративних застосунків. BlaBlaCar Ukraine реалізує класичну модель міжміських поїздок, дозволяючи водіям розміщувати маршрути з вказаними зупинками, часом виїзду та кількістю вільних місць. Пасажири можуть переглядати маршрути, обирати зручний варіант і бронювати місце. Основною перевагою є зручна фільтрація, система оцінювання, повідомлення, а також мобільний додаток українською мовою. Однак цей сервіс не орієнтований на локальні мікроспільноти, а отже, не підтримує щоденні поїздки всередині міста або житлового району. Uklon, як платформа для виклику таксі, впровадила деякі елементи ride-sharing логіки у форматі замовлення поїздки з можливістю обрати клас авто, водія, оплату готівкою або карткою, але не дозволяє обирати попутників або самостійно створювати поїздки. Крім того, сервіс націлений на комерційний сегмент із чіткою тарифікацією й мінімальним простором для неформальної взаємодії між мешканцями одного району.

Інші українські сервіси, зокрема ті, що функціонують через Telegram-ботів (наприклад, «Поїздки з дому на роботу» або «Carpool Kyiv»), демонструють приклади неформалізованого ride-sharing. Вони зазвичай створені ентузіастами або локальними адміністраторами будинкових об'єднань і мають такий функціонал: додавання повідомлення про поїздку, короткий опис маршруту, час виїзду, контакт водія. Така форма взаємодії часто не передбачає автоматизованого бронювання місця, немає вбудованої системи перевірки користувачів, рейтингу чи контролю за дотриманням домовленостей. Подібні сервіси зручні для невеликих спільнот, проте мають високий рівень хаотичності, обмежену функціональність і відсутність UI/UX як таких. Водночас популярність подібних груп свідчить про значний попит на ride-

sharing на локальному рівні. Це підтверджується й активністю у Facebook-групах, наприклад, «Підвезу / Потрібно підвезти — Київ» або «Автостоп UA», де користувачі щоденно публікують оголошення про спільні поїздки. Формат тут аналогічний: вказується маршрут, дата і час, контактна інформація, можлива оплата пального. Незважаючи на відсутність автоматизації, ці групи функціонують як платформи для координації дій між людьми, що підтверджує реальну потребу у створенні цифрових сервісів із подібною логікою, але з вищим рівнем зручності, безпеки та передбачуваності.

Окрему увагу заслуговують корпоративні рішення для ride-sharing, які впроваджуються великими компаніями в межах програм корпоративної відповідальності або внутрішньої логістики. Наприклад, окремі IT-компанії або промислові підприємства використовують внутрішні Google-таблиці або системи керування для організації підвезення співробітників. І хоча подібні рішення не є публічними сервісами, вони демонструють, що ride-sharing у межах сталих спільнот має високий рівень ефективності. Певні компанії тестують рішення на базі Slack-ботів, корпоративних порталів або навіть адаптованих CRM-систем. Такі рішення дозволяють створювати стабільні транспортні групи, контролювати час прибуття, маршрути, верифікувати користувачів через корпоративні облікові записи. Проте ці сервіси не є доступними для широкого загалу і не охоплюють житлові спільноти, що відкриває нішу для розробки незалежного застосунку. Аналізуючи вітчизняні приклади, можна стверджувати, що в Україні поки відсутній єдиний комплексний сервіс, орієнтований на мешканців житлового комплексу з можливістю створення, редагування та приєднання до поїздок у межах мікроспільноти. Існуючі інструменти або мають комерційний характер, або обмежені в технічних можливостях, або надто фрагментовані й неавтоматизовані. На цьому тлі пропонований у роботі застосунок має низку переваг: орієнтація на одну конкретну житлову спільноту, проста реєстрація та взаємодія без залучення сторонніх сервісів, наявність UI з інтуїтивною навігацією, реалізація backend-

функціоналу для обробки поїздок і профілів, а також модульна структура, що дозволяє масштабувати систему на інші будинки або райони. Це дозволяє не лише створити зручний інструмент для повсякденного використання, а й закласти основу для більшої цифрової інтегрованості мешканців, підвищення їх мобільності та згуртованості в межах житлового комплексу. Український контекст формує специфічні вимоги до таких платформ: мінімум бар'єрів при вході, підтримка української мови, офлайн-доступ або локальний сервер, простота інтерфейсу без перевантаження функціями, можливість автономного запуску без залежності від іноземних сервісів. Запропонований підхід враховує ці параметри, об'єднуючи кращі практики Telegram-ботів, гнучкість локальних ініціатив і стабільність традиційного веб-застосунку. У підсумку, порівняльний аналіз українських платформ свідчить про наявність фрагментованого попиту на ride-sharing функціонал, відсутність інституціоналізованого рішення на локальному рівні, що обґрунтовує необхідність створення цілісного застосунку, адаптованого під потреби конкретного житлового комплексу, як це реалізується у рамках даної роботи.

Аналіз міжнародних і вітчизняних платформ, що реалізують механізми спільних поїздок, дозволяє виокремити ряд ключових переваг, які визначають ефективність їхнього функціонування, а також низку суттєвих недоліків, що обмежують їхню застосовність у контексті локальних житлових спільнот. Серед головних переваг таких сервісів варто зазначити зручність цифрової взаємодії, швидкий доступ до потрібної інформації, автоматизацію процесів пошуку попутників, а також можливість реалізації гнучкої логіки маршрутизації. Сучасні застосунки для ride-sharing, такі як Uber, Lyft, BlaBlaCar, Karos, Via, демонструють високий рівень користувацького досвіду завдяки інтуїтивному інтерфейсу, розвиненій системі рейтингів, алгоритмам безпечного парування водіїв і пасажирів, а також широким можливостям персоналізації. Їхня доступність з мобільних пристроїв, підтримка геолокаційних сервісів та використання push-повідомлень дозволяють досягати високої швидкості реагування та підвищеної зручності. Усе це

сприяє формуванню довіри між учасниками системи, що є критично важливим для сервісів, де мають місце міжособистісні взаємодії. На додаток, можливість попереднього бронювання поїздок, підтримка платіжних систем, інтеграція з календарями, картографічними сервісами та іншими цифровими платформами значно розширює функціональні можливості та робить такі сервіси привабливими для широкого кола користувачів. У випадку з BlaBlaCar можна відзначити також соціально-гуманітарний аспект: формування спільноти навколо принципів взаємної підтримки, зниження викидів CO_2, ощадливого використання ресурсів. У свою чергу, Telegram-боти або групи в соціальних мережах, що використовуються в Україні для організації спільних поїздок, мають перевагу в низькому порозі входу — для участі не потрібно встановлювати окремий застосунок, створювати складний профіль чи проходити багатоетапну верифікацію. Ці інструменти дозволяють мешканцям швидко узгоджувати поїздки між собою в реальному часі без складної цифрової інфраструктури. Усе це робить ride-sharing популярною альтернативою як у міжміському, так і у внутрішньоміському трафіку.

Водночас більшість проаналізованих сервісів мають низку недоліків, які стають особливо помітними у контексті спроб реалізувати ride-sharing у межах житлового комплексу або локальної мікроспільноти. По-перше, більшість платформ орієнтовані на великі територіальні охоплення та не враховують локальної специфіки — таких, як особливості розкладу мешканців одного будинку, спільні маршрути до найближчого метро, школи чи роботи. Це призводить до надмірної ускладненості функціоналу, який не відповідає реальним повсякденним потребам користувача. По-друге, велика частина сервісів має обов’язкову фінансову складову, що в умовах невеликих локальних спільнот є зайвим або навіть ускладнюючим фактором, адже користувачі не готові оформлювати платіжні методи, здійснювати мікротрансакції чи відповідати за юридичну частину перевезення. По-третє, обмежена адаптивність до рівня цифрової грамотності. У багатьох випадках, особливо в Україні, частина мешканців житлових масивів

(зокрема люди старшого віку) не мають навичок користування складними інтерфейсами, реєстрацією через email або двофакторною аутентифікацією. Також варто відзначити низький рівень автоматизації у багатьох локальних українських сервісах — Telegram-боти або Google-таблиці не дозволяють реалізувати логіку бронювання, обліку місць, редагування поїздок або синхронізації розкладів. Відсутність централізованої верифікації та контролю веде до зниження безпеки, підвищення ризиків шахрайства або непередбачуваності поведінки користувачів. У деяких випадках сервіси потребують підключення до стабільного інтернету, що в умовах українських реалій (особливо під час аварійних відключень) створює додаткові труднощі. Проблемною є й відсутність інтеграції з іншими інфраструктурними елементами, такими як паркінги, охоронні системи житлового комплексу, календарі подій або адмінпанелі. Недостатня локалізація — ще один недолік: інтерфейс часто не перекладений українською або не адаптований до культурно-комунікативних особливостей користувачів.

Іншою серйозною вадою глобальних платформ є те, що вони не підтримують повторюваність маршрутів у межах однієї і тієї ж групи. Це унеможливлює формування стабільних зв’язків між учасниками спільноти, тоді як у контексті житлового комплексу логіка поїздок часто повторюється: вранці мешканці їдуть на роботу, увечері — повертаються, і в межах цієї регулярності було б доцільно формувати «транспортні мікрогрупи». Відсутність функції фіксованих маршрутів або шаблонних поїздок знижує ефективність системи. Також багато сервісів не підтримують сценарії, коли користувач є водієм у одній поїздці і пасажиром в іншій, що суперечить реальній динаміці користування особистим транспортом. Крім того, жодна з існуючих платформ, зокрема BlaBlaCar чи Uber, не передбачає внутрішньої логіки для житлового комплексу як окремого осередку з власною спільнотою, внутрішньою структурою і можливістю локального адміністрування. Це обмеження є критичним для проєктів, що орієнтовані на цифрову трансформацію життя конкретної житлової спільноти, де важливими є не лише поїздки, а й

комунікація, довіра, впізнаваність між мешканцями. У Facebook-групах і Telegram-чатах немає системи збереження історії поїздок, аналітики, системи нагадувань чи візуального планування, що унеможливлює масштабування і впровадження таких систем у довгостроковій перспективі. Нарешті, навіть у розвинених системах спостерігається відсутність модулів аналітики: користувачі не можуть оцінити свою активність, частоту поїздок, участь у транспортних ініціативах або зекономлені ресурси, що знижує мотиваційний компонент.

У підсумку, оцінка переваг і недоліків існуючих рішень дозволяє зробити висновок про доцільність розробки нового застосунку, який буде поєднувати сильні сторони аналізованих сервісів (інтуїтивний інтерфейс, автоматизація, безпека, шаблонізація дій) із урахуванням недоліків (відсутність локалізації, складна структура, непристосованість до мікроспільнот, відсутність фокусування на повторювані маршрути та локальну взаємодію). Саме тому запропонована в цій роботі система розроблена як веб-застосунок із простою реєстрацією, можливістю створення та редагування поїздок, приєднання до маршрутів інших мешканців, переглядом профілю, а також з відкритою архітектурою, що дозволяє адаптувати її до будь-якого житлового комплексу. Такий підхід дозволяє заповнити нішу, що утворилась між глобальними універсальними платформами та локальними неформальними ініціативами, створюючи цифровий інструмент, який відповідає як технічним, так і соціальним вимогам локального міського середовища.

Таблиця 1.1

Аналіз сучасних сервісів для організації поїздок

Сервіс	Модель поїздки	Фокус на спільноту	Автоматизація процесів	Підтримка мобільного додатку	Інтерфейс українською	Можливість створити поїздку
BlaBlaCar	Міжміська	Низький	Висока	Так	Так	Так
Uber	Міська	Низький	Висока	Так	Так	Ні
Lyft	Міська	Низький	Висока	Так	Ні	Ні
Karos	Комутинг	Середній	Висока	Так	Ні	Так
Via	Міська	Середній	Висока	Так	Ні	Ні
Telegram-боти	Локальна	Високий	Низька	Ні	Так	Так
Uklon	Таксі	Низький	Середня	Так	Так	Ні

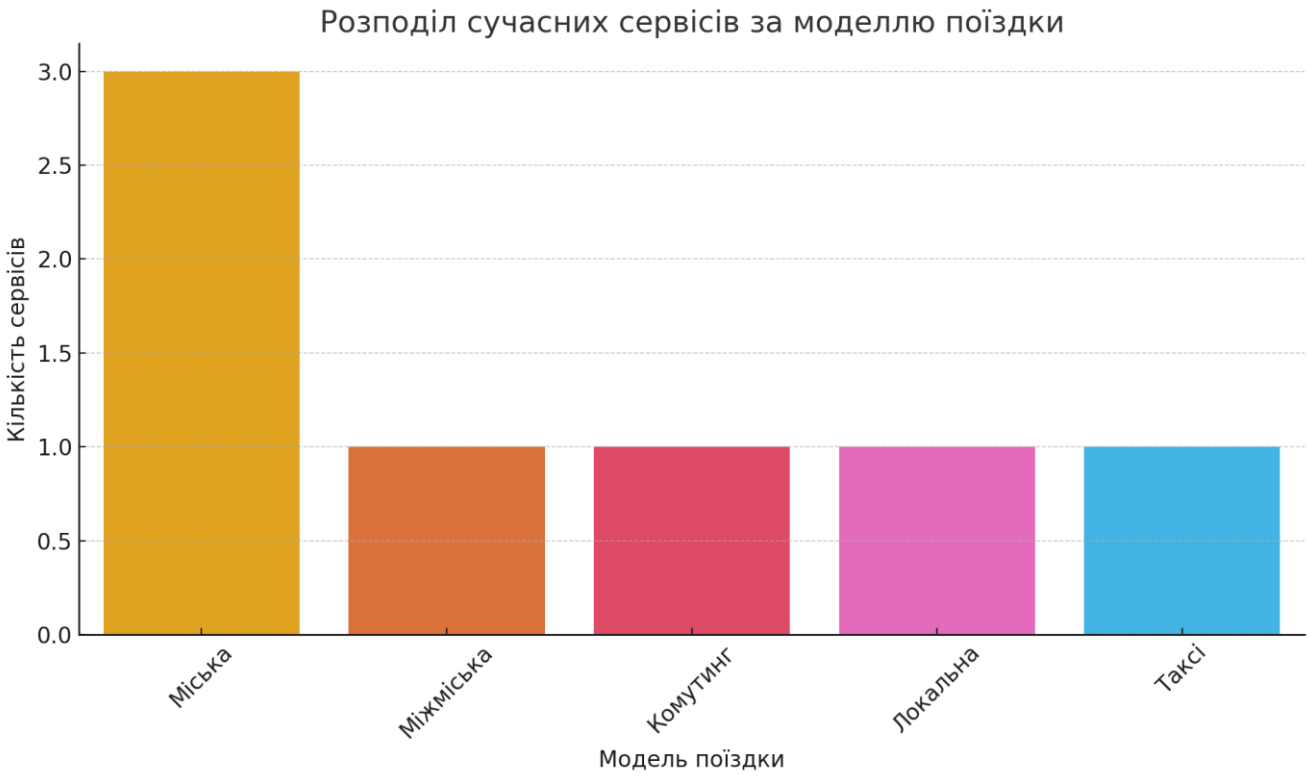


Рис. 1.1 Аналіз сучасних сервісів для організації поїздок

1.2. Особливості потреб мешканців житлового комплексу

У сучасному урбанізованому середовищі соціальні та побутові аспекти пересування відіграють визначальну роль у формуванні ритму життя мешканців житлових комплексів. Переміщення з дому на роботу, до навчальних закладів, медичних установ, торгових центрів та інших об'єктів соціальної інфраструктури стає невід’ємною частиною щоденної рутини, що потребує ефективних і зручних рішень. Значна частина жителів стикається з такими проблемами, як перевантаженість громадського транспорту, недостатня гнучкість розкладів, обмеженість маршрутів, довгі пересадки, затори на дорогах та високі витрати часу. Ці побутові труднощі особливо гостро проявляються у великих містах або на околицях, де рівень транспортної доступності є недостатнім або нестабільним. У таких умовах виникає потреба в альтернативних формах мобільності, зокрема в організації спільних поїздок, які дозволяють оптимізувати транспортні потоки, економити час, знижувати фінансові витрати та сприяти формуванню локальних спільнот. Спільне пересування не лише задовольняє функціональні потреби, а й створює соціальні зв’язки між мешканцями, зміцнюючи відчуття належності до певного середовища, що особливо важливо в контексті новобудов, де мешканці часто не мають історії спільного проживання.

З соціального погляду, поїздки можуть перетворюватися на простір для комунікації, взаємодопомоги та налагодження неформальних контактів між сусідами. Спільне використання транспорту сприяє згуртованості, підвищує рівень довіри та відповідальності, оскільки користувачі починають сприймати один одного не як анонімних пасажирів, а як учасників однієї житлової спільноти. Це має позитивний вплив на загальну атмосферу в комплексі, формуючи відчуття безпеки, підтримки та солідарності. Зі зростанням цін на пальне та послуг таксі, а також із появою нових соціальних ризиків, таких як пандемічні обмеження чи воєнний стан, мешканці прагнуть знаходити локальні, економічні та контрольовані способи

пересування. У цьому контексті спільні поїздки постають як побутове рішення, яке не потребує значних фінансових витрат, дозволяє уникати скупчень у громадському транспорті та створює більш передбачуване середовище для щоденних переміщень. Особливо актуальними такі аспекти є для родин із дітьми, людей похилого віку або тих, хто працює в нічну зміну, коли громадський транспорт є обмеженим або небезпечним.

Крім того, побутові поїздки часто мають повторюваний характер: це регулярні маршрути на роботу, до школи, у торгові центри чи спортивні секції. Така ритмічність дозволяє створювати постійні групи користувачів із близькими графіками, що сприяє підвищенню стабільності сервісу та зменшенню логістичних витрат. Завдяки цифровим сервісам мешканці отримують можливість не лише організовувати такі поїздки, а й бачити історію взаємодій, залишати відгуки, узгоджувати деталі заздалегідь, що робить побутову логістику більш контрольованою та прозорою. Зменшення кількості окремих поїздок приватним транспортом також має екологічний ефект: менше автомобілів на дорогах означає менше викидів, менше шуму, менше заторів та зниження навантаження на інфраструктуру. З огляду на це, застосунок для організації спільних поїздок може стати важливим елементом локального соціального капіталу, забезпечуючи мешканцям житлового комплексу як функціональну, так і соціальну підтримку в побутових переміщеннях. Його впровадження дозволяє не лише задовольнити потребу в комфортному пересуванні, а й створити підґрунтя для нових форм локальної солідарності, цифрової взаємодії та відповідального співжиття в межах міського простору.

Для вивчення реальних потреб і очікувань цільової аудиторії щодо використання сервісу для організації спільних поїздок було проведено анкетне опитування серед мешканців одного з багатоповерхових житлових комплексів у великому місті України. Загалом участь у дослідженні взяли 84 респонденти різного віку, соціального статусу та професійної зайнятості, що дозволило сформувати

репрезентативну картину побутових практик пересування та потенційного попиту на цифрові рішення для ride-sharing. Опитування здійснювалося у форматі онлайн-анкети, поширеної через внутрішню групу житлового комплексу у месенджері, а також за QR-кодами на оголошеннях біля під'їздів. У структурі анкети передбачено як закриті, так і відкриті запитання, що дало змогу зібрати як кількісні, так і якісні показники. У результаті було визначено, що 62% опитаних здійснюють щоденні поїздки в межах міста, з них понад 70% пересуваються однаково вранці та ввечері за стабільним маршрутом. Це свідчить про наявність стабільної логістичної моделі, яка потенційно підходить для організації повторюваних маршрутів у межах житлової спільноти. 48% респондентів щонайменше двічі на тиждень їздять на роботу або навчання приватним транспортом і зазначають, що в салоні залишається принаймні одне вільне місце. Це вказує на високий потенціал використання наявного ресурсу для перевезення інших мешканців без додаткових витрат. 41% опитаних висловили готовність інколи підвозити сусідів, якщо це не створює значних відхилень від маршруту, а 36% готові регулярно брати участь у системі обміну поїздками, якщо вона буде безпечною, зручною та простою у використанні.

Особливо показовими є результати, що стосуються мотивацій та бар'єрів до участі в ride-sharing ініціативах. Основними причинами, що стимулюють бажання користуватися сервісом, респонденти назвали економію пального (67%), зменшення витрат на проїзд для пасажирів (59%), скорочення часу очікування транспорту (52%) та зниження навантаження на особистий автомобіль (38%). Крім того, 44% респондентів підкреслили соціальний ефект від спілкування з сусідами та можливість побудови довірливих зв'язків. Натомість серед основних побоювань учасники вказали брак довіри до незнайомців (72%), страх щодо безпеки поїздки (49%), незручність у випадку запізнення когось із пасажирів (46%) та небажання порушення особистого простору (31%). Проте варто зазначити, що 81% усіх опитаних визнали, що в разі функціонування сервісу на базі їхнього житлового комплексу, з можливістю перегляду профілів, залишення відгуків, перегляду

маршрутів і рейтингу учасників, рівень їх довіри до платформи значно зросте. Це свідчить про актуальність запровадження таких функцій, як система реєстрації, перевірка користувача через номер телефону, а також прозора історія активності. На запитання про бажаний функціонал потенційного застосунку найбільше голосів отримали можливість створення маршруту з вибором часу (78%), перегляд доступних поїздок за напрямками (74%), зручна система повідомлень або сповіщень про оновлення (61%), можливість редагування профілю (58%) та функція екстреного повідомлення (42%). Також 27% респондентів висловили зацікавлення в можливості створення “закритих груп” поїздок — наприклад, лише для мешканців одного під’їзду або поверху. Це вказує на потребу в більш гнучкій архітектурі системи з можливістю формування підгруп і внутрішньої фільтрації учасників.

Окремий блок анкети був присвячений вивченню рівня цифрової грамотності користувачів. 94% респондентів користуються смартфоном щодня, 87% впевнено працюють із мобільними додатками, такими як банківські сервіси, месенджери або сервіси таксі. При цьому 63% висловили бажання користуватися саме веб-застосунком без необхідності завантаження окремого мобільного додатку, аргументуючи це браком місця на пристрої або небажанням встановлювати ще один додаток. Це стало одним із головних аргументів на користь вибору саме веб-платформи як технологічної основи майбутнього сервісу. Крім того, 54% респондентів вказали, що не хочуть прив’язувати використання сервісу до платіжної системи, натомість схиляючись до ідеї добровільної компенсації у разі домовленості між учасниками. Це дозволяє уникати складних юридичних процедур, ліцензування та податкових наслідків, спрощуючи модель сервісу та зберігаючи її в межах спільнотної взаємодії. Серед функцій, які потенційні користувачі визнали необов’язковими або небажаними, були чат між користувачами (лише 19% висловили зацікавлення), інтеграція з навігатором (22%) та підтримка грошових переказів (18%). Це вказує на перевагу простоти над

багатофункціональністю. У вільних відповідях респонденти запропонували додаткові ідеї: додавання кнопки “я зараз їду — хто хоче приєднатися?”, фільтрацію поїздок за часом, статус авто водія (комфорт, кількість місць, наявність дитячого сидіння), а також запровадження механізму тимчасового блокування акаунту в разі порушення домовленостей. Усі ці побажання були враховані при формуванні функціональних вимог до системи.

Узагальнюючи результати опитування, можна зробити висновок, що в межах житлового комплексу існує високий потенціал для впровадження сервісу організації спільних поїздок, з чітко вираженим попитом, обґрунтованими очікуваннями, конкретними побутовими сценаріями та високим рівнем готовності користувачів до цифрової взаємодії. Водночас сервіс повинен бути адаптований до локального контексту: мати просту та інтуїтивну структуру, не вимагати складної реєстрації, не містити грошових операцій, бути зручним для швидкої комунікації та враховувати побажання мешканців щодо прозорості, безпеки та гнучкості. Результати дослідження стали основою для розробки архітектури веб-застосунку, який є предметом цього дослідження, та підтверджують практичну доцільність його створення.

Цільова аудиторія будь-якого цифрового продукту, зокрема веб-застосунку для організації спільних поїздок, повинна бути чітко окреслена ще на етапі проєктування, оскільки саме її особливості визначають як функціональні, так і нефункціональні вимоги до системи. У контексті розробки застосунку для мешканців житлового комплексу специфіка цільової аудиторії відрізняється від класичних ride-sharing сервісів, що орієнтовані на широке коло користувачів. Основною особливістю є територіальна локалізованість – користувачі проживають в межах одного або кількох будинків, мають спільну інфраструктуру, подібний графік пересування, та часто знайомі один з одним або перебувають у межах певної соціальної взаємодії. Це значно підвищує рівень потенційної довіри між користувачами та дозволяє формувати закриті або напівзакриті групи з високим

рівнем кооперації. За результатами попереднього опитування, більшість потенційних користувачів – це особи віком від 25 до 45 років, які мають стабільну зайнятість, щоденно здійснюють поїздки в межах міста або передмістя, використовують смартфон для вирішення побутових завдань та демонструють високий рівень цифрової залученості. Значна частина з них – молоді батьки, фахівці у сфері IT, менеджменту, медицини, викладачі, а також студенти старших курсів, які володіють приватним або службовим транспортом, але часто подорожують з неповним завантаженням авто. Така структура користувачів передбачає потребу в сервісі, який дозволяє не лише знаходити попутників, а й формувати стабільні маршрути, узгоджувати деталі поїздки, відслідковувати історію участі та зберігати профілі учасників.

Ще однією характерною рисою цільової аудиторії є прагнення до безпечного, контрольованого середовища взаємодії. У межах житлового комплексу користувачі не сприймають один одного як анонімних клієнтів, як це відбувається у глобальних ride-hailing платформах, натомість очікують на більшу передбачуваність, повагу до особистого простору, узгодження правил поведінки та можливість прямого контакту без участі зовнішнього посередника. Це створює специфічну модель поведінки, де взаємодія базується не лише на функціональності, а й на соціальних нормах спільноти. Так, наприклад, пасажири частіше надають перевагу знайомим водіям, а водії – тим, хто вже брав участь у поїздках без запізнень або порушень домовленостей. Така поведінка формує особливі вимоги до інтерфейсу: необхідна візуалізація історії поїздок, короткі характеристики профілю, можливість встановити статус (водій, пасажир, комбінований режим), залишати короткі коментарі, переглядати репутацію учасника. У контексті локального застосування користувачі очікують простоти: інтерфейс повинен бути зрозумілим з першого погляду, із мінімальною кількістю кроків для виконання дії. Крім того, цільова аудиторія демонструє високу вимогливість до стабільності роботи сервісу: очікується, що застосунок працює безперервно, не потребує завантаження

сторонніх додатків, швидко завантажується з будь-якого пристрою, підтримує адаптивний дизайн для смартфонів та планшетів. Щодо технічних вподобань, респонденти обрали веб-версію як основний інтерфейс, що дозволяє уникнути бар'єру встановлення додатків, зберігає конфіденційність і забезпечує швидкий доступ через QR-коди або внутрішні посилання житлової групи.

Окрему категорію користувачів становлять мешканці, які не мають власного транспорту, але регулярно користуються послугами таксі, громадського транспорту або шукають попутників серед знайомих. Для цієї групи особливо важливою є можливість швидкого перегляду доступних поїздок, фільтрації за маршрутом, часом виїзду, типом автомобіля та умовами (наприклад, наявність дитячого сидіння або обмеження щодо тварин). Інша частина цільової аудиторії – власники транспортних засобів, які втомилися від постійних поїздок наодинці, хочуть оптимізувати витрати на пальне або просто шукають нових форм взаємодії з оточенням. Обидві ці групи об'єднує очікування на безкоштовне користування сервісом, без реклами та нав'язливих функцій, що відповідає принципам горизонтального обміну ресурсами без посередників. Для мешканців важливо також мати можливість обмежити доступ до сервісу: бажано, щоб система була закритою для сторонніх, функціонувала виключно на основі внутрішньої реєстрації за адресою або спеціальним кодом запрошення, що гарантує безпеку і конфіденційність. У такій моделі цільова аудиторія не тільки користується послугами, а й виконує роль регулятора – залишає відгуки, формує репутацію, відсікає недобросовісних учасників. Це вимагає впровадження простих механізмів модерації та самоорганізації спільноти в межах цифрової платформи.

Важливо також враховувати, що цільова аудиторія є мобільною не лише в буквальному, а й у цифровому сенсі: вона звикла до роботи з інтернет-банкінгом, державними послугами онлайн, чат-ботами, електронною чергою в клініках, замовленням їжі або доставкою. Це означає, що потенційні користувачі очікують відповідної якості від будь-якого нового цифрового продукту, в тому числі й

застосунку для організації поїздок. У той же час, у складі цільової групи присутні й люди старшого віку, які не завжди готові взаємодіяти з багатофункціональними системами, тому важливо реалізувати інтерфейс з великими кнопками, спрощеними формами введення, мінімальною кількістю кроків до підтвердження дій. Отже, цільова аудиторія застосунку – це мешканці житлового комплексу, які мають сталу логіку пересування, бажання зменшити транспортне навантаження, готовність до кооперації, очікування безпечного середовища та високі стандарти до якості цифрового сервісу. Специфіка аудиторії вимагає балансування між функціональністю, зручністю, безпекою та соціальною інтеграцією. З огляду на це, розробка застосунку має базуватися на цінностях простоти, відкритості, контрольованості, ефективності та локалізації. Ці характеристики визначають фундаментальний вектор створення системи, що не лише вирішує транспортну проблему, а й укорінюється в соціальну структуру мікроспільноти, перетворюючись на інструмент формування міської культури кооперативного пересування.

Таблиця 1.2

Особливості потреб мешканців житлового комплексу

Категорія мешканців	**Основні потреби**	**Рівень зацікавленості (%)**
Працевлаштовані (25–45 років)	Швидке добирання на роботу, економія пального	85
Батьки з дітьми	Наявність дитячого крісла, безпечні маршрути	78
Студенти та молодь	Доступний транспорт у вечірній час	67
Літні люди (60+ років)	Простий інтерфейс, довіра до сусідів	54
Особи без авто	Можливість знайти підвезення безкоштовно	73
Власники авто (готові підвозити)	Гнучке планування поїздок, облік витрат	88

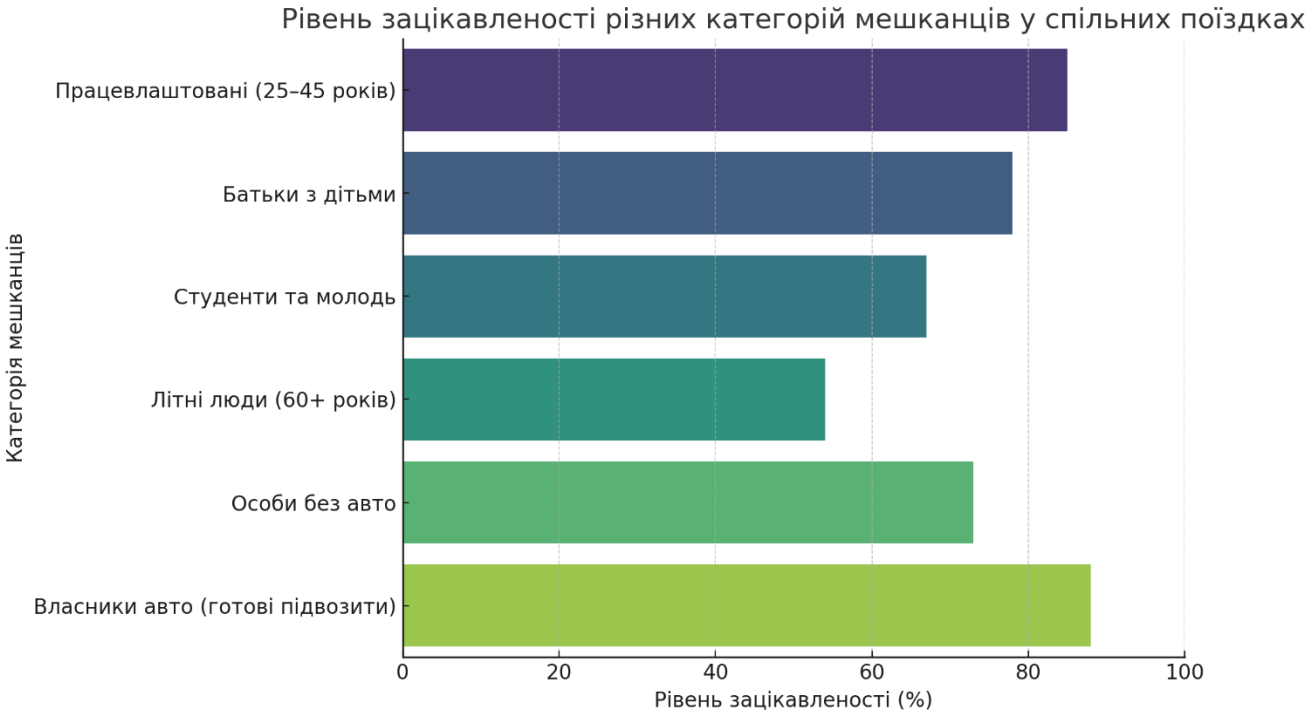


Рис. 1.2 Особливості потреб мешканців житлового комплексу

1.3. Постановка проблеми та формування функціональних вимог до системиф

Ефективна взаємодія між мешканцями житлового комплексу є ключовим чинником формування комфортного соціального середовища, однак на практиці така взаємодія часто ускладнена низкою структурних, організаційних та психологічних бар'єрів, що унеможливлюють реалізацію ініціатив спільного користування ресурсами, зокрема організації спільних поїздок. Однією з головних проблемних зон є відсутність сформованої моделі комунікації між мешканцями. У багатьох житлових комплексах, особливо новозбудованих, спостерігається соціальна фрагментація: люди проживають поруч, але не знайомі між собою, не мають налагоджених каналів спілкування, що обмежує довіру та готовність до кооперації. Брак горизонтальних зв'язків між сусідами призводить до того, що потенційна співпраця на кшталт спільного пересування сприймається як ризик, а не як можливість. Додатково ускладнює ситуацію високий рівень анонімності в багатоповерхових будинках, де мешканці часто навіть не знають імен своїх сусідів. Це створює атмосферу замкненості та настороженості, яка перешкоджає пошуку попутників та ініціюванню колективних поїздок. Особливо це стосується осіб старшого віку або новоприбулих мешканців, які не мають сталих соціальних зв'язків у межах будинку. Іншою важливою проблемною зоною є відсутність ефективної платформи для координації таких ініціатив: традиційні канали на кшталт оголошень на під'їзді або загального чату у месенджері не дозволяють реалізувати структуровану взаємодію, оскільки не передбачають систематизації інформації, бронювання місць, фільтрації маршрутів чи рейтингової системи. У результаті мешканці не знають, хто саме і коли готовий їхати в одному напрямку, скільки вільних місць є в авто, наскільки надійна та відповідальна та чи інша особа.

До проблем організаційного характеру належить також відсутність зрозумілих правил участі в поїздках. У разі несанкціонованих спроб ініціювати спільне пересування виникають численні питання: хто має ініціювати поїздку, як

домовлятися про внесок у пальне, що робити у випадку запізнення чи скасування, як поводитися у випадку конфліктів або непередбачуваних ситуацій під час поїздки. Відсутність чітко встановлених рамок і прозорих процедур створює ризики непорозумінь та конфліктів, що відбивається на бажанні мешканців брати участь у спільних поїздках. Для багатьох також є важливими питання конфіденційності та безпеки: хто матиме доступ до їхніх персональних даних, чи не зловживатимуть цими даними, наскільки надійним є той чи інший водій. Окремо слід виокремити психологічні бар'єри, які виявляються у невпевненості щодо власної ролі у процесі, побоюванні осуду або негативного досвіду в разі невдалої взаємодії. Частина мешканців не готова першими ініціювати поїздку через страх здаватися надто активними або нав'язливими. Інші, навпаки, не хочуть бути пасивними учасниками, але потребують чіткого алгоритму дій, щоб мати змогу впевнено діяти в межах запропонованої системи. Така психологічна неоднозначність у взаємодії мешканців значно ускладнює створення стабільної системи обміну транспортними ресурсами.

Проблемні зони також стосуються технічного боку взаємодії. У багатьох випадках мешканці не мають зручного інструменту для швидкого створення оголошення про поїздку або пошуку вже наявних пропозицій. Вони змушені вручну переглядати чати, писати повідомлення, домовлятися у приватному порядку, що потребує часу та не гарантує ефективного результату. Через це ініціативи залишаються одноразовими й неформалізованими. Більшість користувачів не зберігає історії поїздок, не має змоги залишати відгуки чи формувати базу даних перевірених учасників. Це перешкоджає накопиченню досвіду та розвитку системи на основі реальних практик. До того ж, у разі виникнення технічних збоїв або непередбачених ситуацій, як-от зміна часу виїзду, немає централізованої системи оповіщення, що унеможливлює швидке реагування. Крім того, відсутність гнучкого розкладу або календаря з можливістю планування поїздок наперед знижує рівень організованості процесу. Водночас, мешканці житлових комплексів потребують рішень, що враховують специфіку повторюваних маршрутів, можливість фільтрації

за напрямками, часом, типом транспорту, кількістю пасажирів, умовами перевезення тощо.

Загальним недоліком наявних моделей взаємодії є відсутність цифрової платформи, адаптованої саме до умов житлового комплексу як соціального середовища. Існуючі рішення або надто комерціалізовані (такі як Uber чи Uklon), або надто неструктуровані (на кшталт Telegram-груп і паперових оголошень). Житловий комплекс вимагає спеціалізованої системи, яка б об'єднувала функціональність цифрового сервісу з простотою локальної ініціативи, забезпечуючи мешканцям зручний інструмент взаємодії без бар'єрів у доступі, але з чіткими механізмами саморегуляції. Визначення цих проблемних зон дозволяє сформувати основу для створення такого застосунку, що не лише виконує функцію пошуку попутників, а й компенсує брак соціального капіталу, підвищує рівень довіри між мешканцями та створює інфраструктуру для сталих і безпечних транспортних зв'язків у межах житлової спільноти. У результаті, ефективна цифрова платформа для спільних поїздок має не лише автоматизувати процеси координації, а й виступати каталізатором розвитку горизонтальної взаємодії, що є надзвичайно важливим у контексті урбанізованих середовищ, де особисті контакти стають усе менш частими, а роль цифрових комунікацій — вирішальною.

Узагальнення технічних та функціональних запитів до системи організації спільних поїздок мешканців житлового комплексу є ключовим етапом у розробці ефективного цифрового рішення, що відповідає реальним потребам користувачів. На основі результатів соціологічного опитування, аналізу особливостей цільової аудиторії та ідентифікації проблемних зон у взаємодії було сформульовано конкретні вимоги до функціональності веб-застосунку, які повинні бути враховані у процесі проєктування, розробки та впровадження системи. Передусім було визначено, що одним із основних очікувань є можливість створення та публікації поїздки з мінімальною кількістю кроків. Водій повинен мати змогу ввести маршрут, дату, час виїзду, кількість вільних місць, а також опціональні деталі, як-от тип

транспорту, додаткові умови (наприклад, наявність дитячого сидіння або небажання вести розмови під час поїздки). Інтерфейс повинен забезпечувати швидке збереження даних і їхню візуалізацію в загальному списку доступних поїздок. Другим критично важливим запитом є можливість приєднання пасажирів до вже створених поїздок. Це передбачає реалізацію механізму “бронювання місця”, який автоматично зменшує кількість доступних слотів після підтвердження заявки, а також сповіщає водія про нового учасника. Важливо, щоб користувачі могли бачити всі доступні поїздки у вигляді впорядкованого списку або фільтрованої таблиці, відсортованої за часом, напрямком, наявністю місць та іншими параметрами.

Окрему групу функціональних вимог становлять запити, пов’язані з обліковими записами користувачів. Система повинна дозволяти реєстрацію нового користувача з мінімальним набором обов’язкових полів — ім’я, email або номер телефону, пароль, а також опціонально — наявність авто, адреса проживання, контактний телефон, статус користувача (водій/пасажир/комбінований). Особливо важливо забезпечити можливість подальшого редагування профілю, включно зі зміною пароля, оновленням контактних даних та конфігурацією режиму участі в сервісі. У цьому контексті система повинна підтримувати аутентифікацію користувачів і захищений доступ до особистих даних. Крім того, користувачі мають потребу в перегляді історії своїх поїздок — як створених, так і завершених — із можливістю повторного використання шаблонів поїздок для регулярних маршрутів. Це особливо актуально для осіб, які щодня їздять на роботу або навчання за сталим графіком. До функціональних запитів також належить наявність системи зворотного зв’язку між користувачами. Респонденти вказали на бажаність рейтингової системи, де можна оцінити поїздку за п’ятибальною шкалою, залишити короткий коментар, а також відзначити позитивний або негативний досвід. Ця функція має бути простою, але ефективною, щоб стимулювати відповідальну поведінку та сприяти формуванню репутації в межах цифрової

спільноти. Аналогічно, потрібна опція перегляду рейтингу водіїв і пасажирів перед приєднанням до поїздки, що підвищить рівень довіри між учасниками.

З технічного погляду, система повинна мати архітектуру, що дозволяє розширення функціональності без суттєвих змін у кодовій базі. Обрана модель client-server повинна забезпечити розділення логіки між фронтендом і бекендом, що дає змогу паралельно розвивати інтерфейсну частину та API-зв'язки. На стороні сервера має бути реалізовано базу даних для зберігання інформації про користувачів, поїздки, історію взаємодій, а також конфігурації системи. У зв'язку з цим одним із базових технічних запитів є стабільна інтеграція з базою даних, що дозволяє виконувати CRUD-операції (створення, читання, оновлення, видалення) з мінімальною затримкою. Інтерфейс повинен бути повністю адаптивним — працювати як на стаціонарних комп'ютерах, так і на смартфонах або планшетах. Це зумовлено тим, що більшість користувачів будуть заходити до системи зі своїх мобільних пристроїв, часто на ходу. Інтерфейсна частина має бути оптимізована за швидкістю, з мінімальною кількістю стилів, швидким завантаженням і чіткою логікою переходу між сторінками. Система також повинна підтримувати механізми локального збереження авторизаційного токена (наприклад, через localStorage), щоб користувач не повинен був постійно вводити логін і пароль при кожному запуску. У разі реалізації авторизації через email або телефон, повинна бути можливість відновлення доступу та підтвердження через одноразові коди.

Із погляду безпеки, технічні запити передбачають реалізацію захисту API-запитів, обмеження доступу до адміністративних маршрутів, а також контроль за достовірністю даних, які передаються у форму. Наприклад, система має перевіряти, щоб кількість місць у поїздці не перевищувала фізичну можливість авто, щоб час поїздки не був у минулому, а користувач не міг приєднатися до поїздки повторно або створити накладання у графіку. Також необхідно впровадити систему валідації введених даних — як на стороні клієнта, так і на стороні сервера. Окремим запитом є реалізація механізму сповіщення — зокрема, для інформування про

підтвердження приєднання до поїздки, зміну статусу поїздки або видалення маршруту. Сповіщення можуть бути реалізовані у вигляді спливаючих повідомлень на сторінці, а також push-нотифікацій у майбутньому. Система також повинна підтримувати можливість виходу з профілю, скидання пароля та конфігурації налаштувань видимості — наприклад, приховування номера телефону або адреси від інших користувачів. Важливим технічним запитом є мінімальне навантаження на систему: необхідно уникати складних обчислень на стороні сервера, використовуючи кешування, попередню обробку даних або індексацію таблиць у базі даних.

Окрім базової функціональності, користувачі висловили побажання щодо додаткових, але не критичних можливостей, які можуть бути реалізовані у наступних версіях. Серед них: карта маршруту (на основі Google Maps або OpenStreetMap), система спільного планування для кількох поїздок, модуль групових чатів між учасниками конкретної поїздки, панель адміністратора для модерації користувачів, система лояльності або балів для активних водіїв. Також корисним може бути реалізація темної теми для зручності користування вночі, підтримка багатомовності, модуль новин або сповіщень від адміністрації житлового комплексу. Всі ці функції повинні бути реалізовані у відповідності до гнучкої архітектури, з можливістю включення або вимкнення модулів без зміни основного ядра системи. Отже, узагальнення функціональних і технічних запитів дозволяє створити комплексну специфікацію майбутнього сервісу, що враховує як інженерні, так і соціальні вимоги. Відповідність цим критеріям забезпечить не лише високу якість застосунку, а й його прийнятність, зручність і ефективність у повсякденному користуванні мешканцями житлового комплексу.

Формування мінімального життєздатного функціоналу (Minimum Viable Product, MVP) є одним із найважливіших етапів розробки цифрового сервісу, оскільки дозволяє створити робочий продукт із базовими функціями, який можна швидко протестувати в реальних умовах, отримати зворотний зв’язок і надалі

вдосконалювати систему на основі емпіричних даних. У межах створення веб-застосунку для організації спільних поїздок мешканців житлового комплексу, концепція MVP була визначена з урахуванням особливостей цільової аудиторії, результатів опитування, аналізу проблемних зон взаємодії та технічних обмежень локального середовища. Основна мета MVP полягає не в повному охопленні всіх потенційних функцій сервісу, а у впровадженні критично необхідних можливостей, які забезпечують цінність для кінцевого користувача, дозволяють реалізувати ключові сценарії використання та створюють підґрунтя для подальшого розвитку. У разі обраної тематики це означає, що MVP має забезпечувати базовий цикл взаємодії між водієм і пасажиром у межах одного житлового комплексу: створення поїздки, приєднання до неї, реєстрація користувача, перегляд наявних маршрутів, редагування особистих даних і завершення сеансу. Усі інші функції, як-от рейтинги, повідомлення, карта маршруту чи аналітика — є додатковими і можуть бути реалізовані пізніше.

Основними складовими MVP були визначені: модуль реєстрації та авторизації користувача, інтерфейс створення поїздки, список наявних поїздок із кнопкою приєднання, особистий профіль із можливістю редагування, а також базова система збереження даних на сервері. Реєстрація користувача є відправною точкою взаємодії: система повинна дозволяти створення облікового запису з введенням мінімального набору даних — імені, email або телефону та пароля. Після реєстрації користувач має автоматично потрапляти на головну сторінку, де відображається його профіль та загальний список доступних поїздок. Форма створення поїздки має бути простою й включати обов’язкові поля: маршрут (у вигляді назви або ключових точок), дату та час виїзду, кількість вільних місць. За бажанням користувач може додати примітку (наприклад, “тільки дорослі пасажири” або “виїзд без зупинок”). Після створення поїздка додається до спільного списку, де її можуть побачити інші зареєстровані мешканці. Приєднання до поїздки реалізується натисканням кнопки “Приєднатися”, після чого кількість

вільних місць зменшується, а дані про пасажира зберігаються у відповідній таблиці. Також важливою частиною MVP є можливість перегляду власного профілю з редагуванням контактних даних, пароля, статусу (водій/пасажир), що забезпечує гнучкість у подальшій взаємодії. Усі ці елементи повинні бути реалізовані з використанням базових технологій: HTML, CSS і Vanilla JS на фронтенді, Python із Flask на бекенді, MySQL як система управління базами даних. Такий підхід дозволяє створити просту, надійну й масштабовану архітектуру, що не потребує складних фреймворків, спрощує підтримку системи та підвищує її доступність для локального впровадження.

Дизайн MVP має бути максимально функціональним, без надмірних графічних елементів, але з достатнім рівнем візуального комфорту. Важливо забезпечити логічне групування блоків, контрастність кнопок, зрозумілу типографіку. Застосування градієнтного фону, адаптивних форм та чіткої структури дозволяє досягти сучасного вигляду інтерфейсу навіть без використання CSS-фреймворків. З технічного погляду, MVP не потребує реалізації повного захисту персональних даних або складної системи логування, однак базовий рівень безпеки повинен бути забезпечений: захист від несанкціонованого доступу через механізм авторизації, перевірка введених даних, обмеження по кількості створених поїздок, а також захист маршрутизаторів Flask через відповідні HTTP-методи. Система має функціонувати на локальному сервері для цілей тестування, із можливістю запуску через прості інструменти: Flask для серверної частини та HTTP-server або пряме відкриття HTML-файлів для фронтенду. Це дозволяє швидко розгорнути систему в умовах відсутності хмарної інфраструктури або обмежених технічних ресурсів у житловому комплексі. У майбутньому передбачається можливість розширення MVP до повноцінної системи із підтримкою push-сповіщень, адміністрування, рейтингу, багатомовного інтерфейсу, інтеграції з картами та іншим функціоналом, однак для початкового етапу важливо зосередитися саме на базових сценаріях.

Суть MVP полягає в тому, що він має не лише демонструвати працездатність концепції, а й бути достатньо цінним для реального використання. Це означає, що навіть мінімальний набір функцій повинен працювати стабільно, без критичних помилок, із чіткою логікою переходів і зрозумілим управлінням для користувача без спеціальних технічних знань. У цьому контексті важливо протестувати MVP на невеликій вибірці користувачів — наприклад, мешканцях одного під'їзду — щоб зібрати зворотний зв'язок і уточнити пріоритети подальшої розробки. Усі відгуки мають бути зафіксовані, проаналізовані та класифіковані за рівнем важливості, що дозволить реалізувати підхід ітеративного розвитку з поступовим вдосконаленням системи. Таким чином, формування MVP дозволяє не лише почати використання системи на практиці, а й закласти інституційні та технічні засади для розвитку сервісу в контексті потреб житлового комплексу. Це особливо актуально в умовах обмежених ресурсів, де надмірна складність на старті може призвести до провалу ініціативи, натомість простий і ефективний продукт, побудований на реальних запитах користувачів, має високі шанси на успішне впровадження, масштабування та подальший розвиток як на рівні окремого житлового комплексу, так і в ширшому міському середовищі.

Таблиця 1.3

Постановка проблеми та формування функціональних вимог до системи

Проблемна зона	Відповідна функціональна вимога
Відсутність координації між мешканцями	Список поїздок із можливістю приєднання
Недовіра до спільного користування транспортом	Рейтинги користувачів і система відгуків
Неможливість швидкого пошуку попутників	Фільтрація маршрутів за часом і напрямком
Низька цифрова доступність для старших користувачів	Адаптивний, простий інтерфейс
Відсутність системи обліку поїздок	Історія поїздок і профіль користувача
Нестача гнучкого планування поїздок	Можливість створення шаблонів повторюваних поїздок

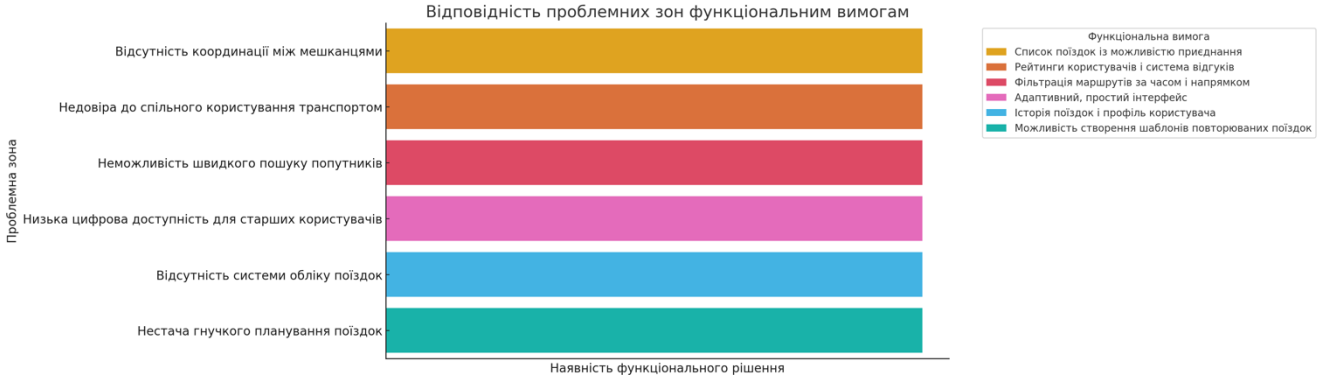


Рис. 1.3 Постановка проблеми та формування функціональних вимог до системи

2. ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1. Архітектура програмної системи

Логічна модель веб-застосунку для організації спільних поїздок мешканців житлового комплексу побудована за принципами модульності, клієнт-серверної взаємодії та REST-архітектури. Основою компонування системи є чітке розділення відповідальності між фронтенд- і бекенд-частинами з централізованим обробленням даних у базі MySQL. Усі компоненти взаємодіють через стандартизовані HTTP-запити, що забезпечує зрозумілу логіку, простоту масштабування і можливість незалежного тестування модулів. На серверній стороні головний файл backend/app.py створює екземпляр Flask-додатку, ініціалізує конфігурацію з config.py, підключає маршрути з підпапки routes та дозволяє CORS для клієнтських запитів. Підключення до бази даних реалізовано у models.pyчерез функцію get_db_connection(), яка використовує стандартний MySQL-конектор без застосування ORM, що дозволяє точний контроль над SQL-запитами та спрощує логіку роботи з базою. Архітектурно це забезпечує прозорість операцій і мінімізує надмірне навантаження на систему.

Маршрути логічно поділені за сферами відповідальності: routes/auth.py відповідає за реєстрацію та авторизацію, обробляючи запити /api/auth/register та /api/auth/login, виконує перевірку валідності даних і повертає відповідь із JSON-об’єктом користувача. Фрагмент реалізації в auth.py:

```
@app.route('/api/auth/register', methods=['POST'])
def register():
    data = request.get_json()
    name, email, password = data['name'], data['email'], data['password'
    conn = get_db_connection()
    cursor = conn.cursor()
    cursor.execute("INSERT INTO users (name, email, password) VALUES (%s
    conn.commit()
    return jsonify({'status': 'success'})
```

Цей код ілюструє типову реалізацію логіки додавання нового користувача до бази з валідацією через Flask. У routes/user.py реалізовано функції для отримання та оновлення профілю користувача. Підтримується метод GET /api/user/<id> для завантаження даних, а також PUT — для оновлення профілю. У свою чергу, routes/rides.py є ядром управління поїздками, підтримуючи основні операції: створення (POST /api/rides/), читання (GET /api/rides/), оновлення (PUT /api/rides/<id>) і видалення поїздки (DELETE /api/rides/<id>), а також приєднання до маршруту за допомогою POST /api/rides/<id>/join. Для прикладу, фрагмент додавання нової поїздки виглядає так:

```
@app.route('/api/rides', methods=['POST'])
def create_ride():
    data = request.get_json()
    conn = get_db_connection()
    cursor = conn.cursor()
    cursor.execute("INSERT INTO rides (user_id, route, departure_time, s
        (data['user_id'], data['route'], data['departure_time'], data['s
    conn.commit()
    return jsonify({'status': 'ride_created'})
```

Цей фрагмент демонструє зберігання маршруту в базі з подальшим відображенням на клієнті. На фронтенді логіка реалізована в frontend/index.html, dashboard.html, edit_profile.html, що взаємодіють через єдиний файл script.js. Наприклад, ydashboard.html користувач після входу бачить свій профіль, може створити нову поїздку за допомогою форми з інпутами для маршруту, дати та кількості місць, а також переглянути таблицю доступних поїздок з кнопками для редагування, видалення або приєднання. Структура даних передається через fetch із методом POST і зберігається у MySQL. Фрагмент JavaScript-коду для створення поїздки виглядає так:

```
document.getElementById("create-ride-form").addEventListener("submit", f
    e.preventDefault();
    const route = document.getElementById("route").value;
    const time = document.getElementById("time").value;
    const seats = document.getElementById("seats").value;
    fetch("/api/rides", {
        method: "POST",
        headers: {"Content-Type": "application/json"},
        body: JSON.stringify({user_id: localStorage.userId, route, depar
    }).then(res => res.json()).then(() => location.reload());
});
```

Цей фрагмент реалізує форму створення поїздки з автоматичним оновленням інтерфейсу. Важливо, що вся логіка фронтенду реалізована без використання фреймворків, що спрощує доступність і підтримку коду. У edit_profile.html користувач може редагувати свій профіль. При завантаженні сторінки виконується запит до API /api/user/<id>, поля форми заповнюються автоматично, а зміни надсилаються назад через PUT. Усе це координується з єдиного script.js, який розрізняє сторінки через window.location.pathname.

Інформаційна модель системи також базується на взаємозв’язку між сутностями users та rides. Таблиця usersзберігає ідентифікатор, ім’я, email, пароль, телефон і статус наявності авто. Таблиця rides містить поля для user_id (зовнішній ключ), маршруту, дати виїзду, кількості місць та часу створення. Додатково, при приєднанні пасажира до поїздки оновлюється кількість доступних місць у базі. Реалізація логіки перевірки — наприклад, щоб не приєднуватися до поїздки, якщо місць немає — виконується на стороні сервера. Таким чином, логічна модель поєднує чітко структурувану API-архітектуру з мінімалістичним клієнтським інтерфейсом і повністю ручним управлінням з’єднанням з базою. Це забезпечує гнучкість, контроль і можливість адаптації для локального рівня застосування, зокрема в межах житлового комплексу, де важлива стабільність, простота та зрозумілість структури застосунку. Уся структура компонування є придатною до масштабування, дозволяє додавати нові модулі (наприклад, повідомлення, карту маршруту, чат) без зміни ядра програми, оскільки кожен модуль ізольований у своїй області відповідальності — відповідно до логіки MVC без явної реалізації шаблонів. У підсумку, структура застосунку дозволяє реалізувати усі функції, необхідні для організації спільних поїздок у межах локальної спільноти, при цьому залишаючись зрозумілою, підтримуваною та гнучкою для майбутнього розвитку.

У процесі створення веб-застосунку для організації спільних поїздок життєво важливим є чіткий розподіл обов’язків між клієнтською (frontend) і серверною (backend) частинами. Такий підхід дозволяє мінімізувати навантаження на обидві сторони, покращити масштабованість системи, прискорити взаємодію користувача з інтерфейсом і забезпечити цілісну логіку обробки запитів. У цьому проєкті архітектура розроблена на основі класичної клієнт-серверної моделі, де клієнтська частина відповідає за відображення інтерфейсу, збір даних від користувача, попередню валідацію введення і відправлення HTTP-запитів, тоді як серверна частина займається обробкою даних, перевіркою їх достовірності, взаємодією з базою даних та логічною відповіддю на запити. У клієнта зберігається лише

тимчасова інформація про активну сесію, зокрема userId у localStorage, що дозволяє працювати зі сесією без авторизаційних токенів на кожен запит. Реєстрація, вхід, перегляд профілю, створення поїздки, перегляд поїздок, редагування даних — усе це реалізується як взаємодія між інтерфейсом, збудованим на HTML, CSS і JavaScript, та API, створеним за допомогою Flask.

Для прикладу, при вході користувача на сторінку index.html він заповнює форму авторизації, а введені дані через JavaScript збираються та надсилаються методом fetch до маршруту /api/auth/login:

```
document.getElementById("login-form").addEventListener("submit", functio
    e.preventDefault();
    const email = document.getElementById("email").value;
    const password = document.getElementById("password").value;
    fetch("/api/auth/login", {
        method: "POST",
        headers: {"Content-Type": "application/json"},
        body: JSON.stringify({email, password})
    }).then(response => response.json())
      .then(data => {
        if (data.status === "success") {
            localStorage.setItem("userId", data.user.id);
            window.location.href = "dashboard.html";
        }
    });
});
```

У цьому випадку на клієнті виконується обробка події, валідація наявності даних у полях форми, формування тіла запиту й перенаправлення користувача в разі успіху. На сервері, відповідно, маршрут /api/auth/loginобробляється так:

```
@app.route('/api/auth/login', methods=['POST'])
def login():
    data = request.get_json()
    email, password = data['email'], data['password']
    conn = get_db_connection()
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM users WHERE email=%s AND password=%s",
    user = cursor.fetchone()
    if user:
        return jsonify({'status': 'success', 'user': {'id': user[0], 'na
    return jsonify({'status': 'error'})
```

Таким чином, сервер приймає дані, проводить перевірку в базі, і в разі відповідності повертає JSON-об’єкт із результатом. Структурно сервер відповідає за всі логічні перевірки, взаємодію з базою MySQL, генерацію помилок, оновлення записів. Водночас клієнт не бачить структуру бази, не має доступу до внутрішньої логіки, а працює лише з даними, які йому повертає сервер у структурованому вигляді. Цей підхід дозволяє дотримуватися принципів безпеки, розмежування відповідальностей і незалежного розвитку інтерфейсу та API. Ще один приклад — створення нової поїздки. На клієнті формується запит з відповідними параметрами:

```
document.getElementById("ride-form").addEventListener("submit", function
    e.preventDefault();
    const route = document.getElementById("route").value;
    const departure_time = document.getElementById("time").value;
    const seats = document.getElementById("seats").value;
    fetch("/api/rides", {
        method: "POST",
        headers: {"Content-Type": "application/json"},
        body: JSON.stringify({
            user_id: localStorage.getItem("userId"),
            route,
            departure_time,
            seats
        })
    }).then(() => location.reload());
});
```

Тут JavaScript на стороні клієнта збирає дані з форми, зберігає user_id з локального сховища й надсилає POST-запит на створення поїздки. Сервер, у свою чергу, отримує ці дані, перевіряє їх, формує SQL-запит та вносить інформацію до таблиці rides:

```
@app.route('/api/rides', methods=['POST'])
def create_ride():
    data = request.get_json()
    conn = get_db_connection()
    cursor = conn.cursor()
    cursor.execute("INSERT INTO rides (user_id, route, departure_time, s
                   (data['user_id'], data['route'], data['departure_time
    conn.commit()
    return jsonify({'status': 'ride_created'})
```

Отже, створення маршруту повністю контролюється сервером, тоді як клієнт виконує лише роль ініціатора запиту та інтерфейсної обгортки. Такий розподіл забезпечує стабільність: навіть якщо клієнт змінюється або оновлюється, API

залишається незмінним. У сценарії приєднання до поїздки клієнт лише передає ID поїздки, сервер же перевіряє кількість вільних місць, оновлює таблицю й повертає відповідь. Додатково клієнт відповідає за візуалізацію таблиці поїздок, яку отримує через GET /api/rides. Сервер формує повний список з JOIN-запитів, включаючи імена авторів маршрутів, що демонструє ще один приклад ефективного розподілу: клієнт отримує готові дані, не здійснюючи обчислень.

Клієнт також керує переходами між сторінками (index.html, dashboard.html, edit_profile.html) і автоматично адаптує виконання script.js залежно від window.location.pathname. Наприклад, при переході на edit_profile.html завантажується профіль користувача через API, заповнюються поля, і при збереженні вносяться зміни:

```
fetch("/api/user/" + userId)
.then(res => res.json())
.then(data => {
    document.getElementById("name").value = data.name;
    document.getElementById("email").value = data.email;
    ...
});
```

Усі запити — асинхронні, що дозволяє не перевантажувати інтерфейс і швидко оновлювати контент без перезавантаження сторінки. Сервер відповідає лише на сформовані запити та не зберігає стану клієнта, що відповідає REST-принципам. У підсумку, розподіл обов’язків у цьому застосунку є чітким: клієнт відповідає за взаємодію, збір та візуалізацію даних, попередню валідацію, навігацію, локальне зберігання userId; сервер — за обробку запитів, валідацію на рівні бази, логіку, зберігання й оновлення даних. Така архітектура не лише забезпечує стабільність, безпеку й гнучкість, а й створює умови для масштабування системи в майбутньому з можливістю підключення нових сервісів, авторизацій, повідомлень або адміністрування без зміни базової логіки.

Вибір архітектурного підходу при розробці веб-застосунку для організації спільних поїздок мешканців житлового комплексу був здійснений на основі аналізу вимог до функціональності, технічних обмежень, очікувань користувачів і потенціалу масштабування системи. Основою обраної моделі стала класична трирівнева клієнт-серверна архітектура з чітким розподілом на рівні представлення (інтерфейс), логіки (API) та даних (база MySQL). Такий підхід дозволяє не лише забезпечити прозору взаємодію між компонентами системи, а й гарантує незалежність розвитку кожного рівня без втручання в інші частини, що критично важливо при подальшому розширенні. Обрана структура відповідає принципам RESTful-архітектури, де кожна сутність має власні URL-ендпоінти, а взаємодія відбувається через HTTP-запити з передачою JSON-даних. Наприклад, маршрут GET /api/rides повертає список усіх поїздок, POST /api/rides дозволяє створити нову поїздку, а PUT /api/user/<id>— редагувати профіль. Така структура забезпечує зрозумілу логіку й дає змогу масштабувати API шляхом додавання нових маршрутів без зміни існуючого ядра.

Архітектура реалізована за допомогою фреймворку Flask, що забезпечує гнучкість, легкість налаштування та мінімальний оверхед. Серверна частина зібрана у вигляді єдиного застосунку в app.py, де створюється екземпляр сервера, підключаються маршрути з модуля routes, ініціалізується конфігурація з файлу config.py та виконується запуск системи. Окремий модуль models.py відповідає за підключення до бази даних і виконує SQL-запити напряму через mysql.connector, що дозволяє точний контроль над структурою таблиць і логікою запитів. Це рішення було прийняте з метою уникнення складнощів, пов’язаних із ORM (наприклад, SQLAlchemy), оскільки застосунок передбачає прямолінійну схему з низькою складністю запитів. Усі запити мають форму:

```
def get_db_connection():
    return mysql.connector.connect(
        host="localhost",
        user="root",
        password="root",
        database="carpool"
    )
```

У клієнтській частині архітектура побудована як статичний фронтенд без використання фреймворків типу React чи Vue, що дозволило зменшити навантаження на систему й зробити застосунок доступним навіть на пристроях із низькою продуктивністю. HTML-сторінки index.html, dashboard.html, edit_profile.html пов'язані з єдиним JavaScript-файлом script.js, який виконує логіку взаємодії з API через fetch. Наприклад, отримання списку поїздок виглядає так:

```
fetch("/api/rides")
.then(res => res.json())
.then(data => {
    data.forEach(ride => {
        const row = document.createElement("tr");
        row.innerHTML = `<td>${ride.route}</td><td>${ride.departure_time
        document.getElementById("rides-table").appendChild(row);
    });
});
```

Цей код виконує запит до сервера, отримує список поїздок у форматі JSON та динамічно відображає їх у таблиці на сторінці. Такий підхід дозволяє мінімізувати час відгуку, уникнути повного перезавантаження сторінки та забезпечити інтерактивність без складної логіки. Для збереження сесії використовується localStorage, у якому зберігається ID користувача після входу в систему. При кожному запиті дані використовуються для авторизації запиту або ідентифікації, хто створює маршрут:

```
const userId = localStorage.getItem("userId");
```

На сервері, відповідно, запити до POST /api/rides використовують це значення для зв’язку з таблицею користувачів:

```
@app.route('/api/rides', methods=['POST'])
def create_ride():
    data = request.get_json()
    cursor = get_db_connection().cursor()
    cursor.execute("INSERT INTO rides (user_id, route, departure_time, s
                   (data['user_id'], data['route'], data['departure_time
    ...
```

Таке компонування дозволяє клієнту виконувати лише роль ініціатора та візуалізатора, а сервер повністю відповідає за бізнес-логіку, що спрощує підтримку коду й дозволяє централізовано контролювати обробку даних. Крім того, архітектурна модель побудована таким чином, щоб кожен модуль мав власну зону відповідальності. Усі функції, пов’язані з авторизацією, зосереджені в auth.py, усі функції користувача — в user.py, а логіка поїздок — в rides.py. Це дозволяє незалежно тестувати маршрути, додавати нові без впливу на решту системи та забезпечує масштабованість. У разі потреби додати модуль повідомлень або аналітики — достатньо створити окремий файл із відповідними маршрутами, не змінюючи ядро. Структура даних також адаптована до архітектурного підходу: у таблиці users зберігаються всі основні поля — id, name, email, password, phone, is_driver, а в rides — id, user_id, route, departure_time, seats, що забезпечує зв’язок через зовнішній ключ user_id.

Перевагою цієї архітектури є можливість розгортання на будь-якому середовищі, включно з локальними серверами, хостингами або хмарними

рішеннями типу Heroku. Простота структури дозволяє легко перемістити код, налаштувати середовище виконання, створити резервні копії або розгорнути кілька версій одночасно. Архітектура підтримує розмежування доступу: для користувача відкриті лише відповідні API, тоді як усі внутрішні механізми роботи з базою ізольовані. Уся логіка є stateless — сервер не зберігає стану між запитами, що відповідає REST-принципам і дозволяє масштабувати сервіс горизонтально, у тому числі шляхом балансування навантаження.

Таким чином, обрана архітектурна модель — це збалансоване рішення між простотою реалізації, ефективністю обробки, стабільністю функціонування та відкритістю до майбутнього розвитку. Вона відповідає реальним технічним і соціальним запитам користувачів: дозволяє швидке завантаження інтерфейсу, безпеку особистих даних, інтуїтивне управління функціями, а також повністю ізольовану логіку на сервері. Такий підхід забезпечує не лише працездатність MVP, а й фундамент для подальшого розвитку — включення модулів адміністративного контролю, аналітики, групових чатів, push-сповіщень, карт маршруту, систем лояльності та мобільної адаптації. Архітектура побудована на відкритих рішеннях і стандартах, що дозволяє підтримку будь-якими фахівцями без прив’язки до вузькоспеціалізованих технологій. У підсумку, це універсальна й гнучка модель, придатна для впровадження в житлових комплексах, локальних громадах або на рівні мікроспільнот.

2.2. Вибір технологій та інструментів реалізації

Реалізація інтерфейсної частини веб-застосунку для організації спільних поїздок здійснена із використанням базових, але ефективних фронтенд-інструментів — HTML5, CSS3 та Vanilla JavaScript, що дало змогу забезпечити швидке завантаження сторінок, інтуїтивну навігацію та високу адаптивність без залучення важких фреймворків. Перевагою такого підходу є мінімальне навантаження на браузер, що особливо важливо для користувачів із повільними мобільними пристроями або нестабільним інтернетом. Розмітка сторінок

побудована на структурованій семантичній моделі з використанням стандартних тегів <form>, <table>, <section>, <input>, що забезпечує коректне відображення на більшості браузерів. Наприклад, основна панель користувача dashboard.html містить таблицю з поїздками, кнопку створення нової поїздки, а також короткий профіль користувача у верхньому куті сторінки. Приклад HTML-структури:

```
<section id="ride-creation">
  <form id="ride-form">
    <input type="text" id="route" placeholder="Маршрут" required />
    <input type="datetime-local" id="departure_time" required />
    <input type="number" id="seats" min="1" placeholder="Кількість місць
    <button type="submit">Створити поїздку</button>
  </form>
</section>
<table id="rides-table">
  <thead>
    <tr><th>Маршрут</th><th>Час</th><th>Місця</th><th>Дія</th></tr>
  </thead>
  <tbody></tbody>
</table>
```

Цей фрагмент реалізує зрозумілу ієрархію розміщення елементів і є основою динамічного оновлення контенту через JavaScript. Уся логіка обробки подій, запитів до сервера та рендерінгу динамічного контенту реалізована в одному файлі script.js, що дозволяє централізовано керувати всіма сценаріями без надмірного розділення логіки. Перевага використання Vanilla JS полягає у відсутності залежності від зовнішніх бібліотек, а також у повному контролі над усіма подіями, DOM-елементами та запитами. Наприклад, динамічне завантаження поїздок реалізується так:

```
fetch("/api/rides")
  .then(response => response.json())
  .then(data => {
    const table = document.getElementById("rides-table").querySelector("
    data.forEach(ride => {
      const row = document.createElement("tr");
      row.innerHTML = `
        <td>${ride.route}</td>
        <td>${ride.departure_time}</td>
        <td>${ride.seats}</td>
        <td><button onclick="joinRide(${ride.id})">Приєднатись</button><
      `;
      table.appendChild(row);
    });
  });
```

Кожна поїздка виводиться як окремий рядок у таблиці, а кнопка «Приєднатись» викликає функцію joinRide(id), яка надсилає запит до API для приєднання до маршруту. Цей підхід гарантує, що контент оновлюється асинхронно без перезавантаження сторінки, що є важливим показником сучасної взаємодії в веб-інтерфейсі. Обробка подій форми створення поїздки реалізована через addEventListener, що дозволяє точно керувати моментом ініціації запиту:

```
document.getElementById("ride-form").addEventListener("submit", function
  e.preventDefault();
  const route = document.getElementById("route").value;
  const time = document.getElementById("departure_time").value;
  const seats = document.getElementById("seats").value;
  fetch("/api/rides", {
    method: "POST",
    headers: {"Content-Type": "application/json"},
    body: JSON.stringify({
      user_id: localStorage.getItem("userId"),
      route,
      departure_time: time,
      seats
    })
  }).then(() => location.reload());
});
```

Завдяки використанню fetch з методами POST та GET, система лишається легкою, а код — читабельним і придатним до модифікації. Стилізація сторінок здійснена через базові властивості CSS3 із фокусом на адаптивність і контрастність. Користувацький інтерфейс має чітку кольорову палітру, великі інпут-форми та кнопки, що зручно для мобільного використання. У CSS використано flex-контейнери, позиціонування блоків за допомогою display: flex, justify-content, align-items, що дозволяє забезпечити правильне відображення елементів на різних екранах:

```
#ride-creation {
  display: flex;
  flex-direction: column;
  gap: 10px;
  max-width: 400px;
  margin: auto;
}
input, button {
  padding: 10px;
  font-size: 16px;
  border-radius: 5px;
}
table {
  width: 100%;
  border-collapse: collapse;
  margin-top: 20px;
}
th, td {
  border: 1px solid #ccc;
  padding: 8px;
  text-align: center;
}
```

Таким чином, навіть без використання препроцесорів чи фреймворків на кшталт Bootstrap чи Tailwind вдалося реалізувати повноцінний інтерфейс, що відповідає сучасним вимогам до юзабіліті. Додатковою перевагою є зменшення ваги файлів та прискорення завантаження сторінок. Логіка навігації між сторінками не використовує SPA-підходу, але реалізована через прямі переходи: при успішній авторизації користувач перенаправляється на dashboard.html, після редагування профілю — на головну. Це мінімізує складність клієнтського коду і спрощує дебагінг. Збереження сесії користувача реалізовано через localStorage, а не через cookies або JWT, що дозволяє зберігати мінімум інформації (лише userId) і при цьому уникнути складних механізмів аутентифікації:

```
if (!localStorage.getItem("userId")) {
  window.location.href = "index.html";
}
```

У підсумку, використання HTML, CSS і чистого JavaScript дає можливість створити стабільний, легкий та доступний фронтенд без залежності від складних бібліотек, що полегшує розгортання системи, її підтримку та адаптацію до нових функцій. Ці інструменти забезпечують повний контроль над DOM, просту інтеграцію з API, ефективну обробку подій та динамічну побудову інтерфейсу. Такий підхід особливо ефективний у випадках, коли важлива швидкість розробки MVP, низький поріг входу для нових розробників, відсутність потреби в анімаціях або складній бізнес-логіці на клієнті. Простота структури дає змогу будь-якому користувачеві з базовими навичками HTML/JS адаптувати або масштабувати систему під конкретні потреби. Це доводить практичну ефективність базових фронтенд-інструментів як оптимального рішення для локальних веб-платформ у сфері транспортування та самоорганізації мешканців.

Бекенд-сервер реалізовано за допомогою мікрофреймворку Flask — потужного та легкого інструмента для побудови веб-додатків на мові Python. Його перевага полягає у мінімалістичному підході до структури, що дозволяє точно контролювати логіку маршрутизації, гнучко взаємодіяти з базами даних і швидко налаштовувати API. Серверна частина побудована на основі REST-підходу з поділом маршрутів за функціональними зонами — авторизація, управління користувачем, обробка поїздок. Основний сервер запускається з файлу app.py, який створює екземпляр Flask-додатку, підключає CORS, реєструє маршрути з папки routes і налаштовує конфігурацію:

```
from flask import Flask
from flask_cors import CORS
app = Flask(__name__)
CORS(app)

from routes.auth import *
from routes.user import *
from routes.rides import *

if __name__ == "__main__":
    app.run(debug=True)
```

Така структура дозволяє легко розширювати функціональність — кожен модуль із логікою зберігається окремо в логічних підпапках. Наприклад, у routes/auth.py зосереджено обробку реєстрації та логіну, де /api/auth/register приймає JSON-дані з фронтенду, валідує їх, вставляє новий запис у таблицю users бази MySQL та повертає результат у вигляді JSON. Приклад коду:

```
@app.route('/api/auth/register', methods=['POST'])
def register():
    data = request.get_json()
    name, email, password = data['name'], data['email'], data['password'
    conn = get_db_connection()
    cursor = conn.cursor()
    cursor.execute("INSERT INTO users (name, email, password) VALUES (%s
    conn.commit()
    return jsonify({'status': 'success'})
```

База даних підключається через стандартний MySQL-конектор у файлі models.py, що містить функцію get_db_connection() для ініціалізації з'єднання з параметрами:

```
import mysql.connector
def get_db_connection():
    return mysql.connector.connect(
        host="localhost",
        user="root",
        password="root",
        database="carpool"
    )
```

Це дозволяє напряму виконувати SQL-запити без ORM, що полегшує контроль над запитами і спрощує налагодження. У routes/user.py реалізовано логіку перегляду й оновлення профілю. Користувач може отримати свої дані через GET /api/user/<id>, а також оновити їх за допомогою PUT, надсилаючи оновлені поля в JSON-форматі. Наприклад:

```
@app.route('/api/user/<int:id>', methods=['PUT'])
def update_user(id):
    data = request.get_json()
    conn = get_db_connection()
    cursor = conn.cursor()
    cursor.execute("UPDATE users SET name=%s, phone=%s WHERE id=%s",
                   (data['name'], data['phone'], id))
    conn.commit()
    return jsonify({'status': 'updated'})
```

Найбільший блок логіки зосереджено в routes/rides.py, де реалізовано усі CRUD-операції з поїздками. Користувач може створити поїздку через POST /api/rides, переглянути список через GET, оновити або видалити свою поїздку, а також приєднатися до наявної через POST /api/rides/<id>/join. Кожен маршрут отримує JSON-запит, розпаковує дані, виконує запит до бази і повертає відповідь. Наприклад, створення поїздки виглядає так:

```
@app.route('/api/rides', methods=['POST'])
def create_ride():
    data = request.get_json()
    user_id = data['user_id']
    route = data['route']
    time = data['departure_time']
    seats = data['seats']
    conn = get_db_connection()
    cursor = conn.cursor()
    cursor.execute("INSERT INTO rides (user_id, route, departure_time, s
                   (user_id, route, time, seats))
    conn.commit()
    return jsonify({'status': 'ride_created'})
```

При приєднанні пасажира до поїздки оновлюється таблиця відповідності, а також зменшується кількість вільних місць. Важливо, що всі перевірки (чи є місця, чи користувач вже приєднався) виконуються на сервері, а не на клієнті, що запобігає зловживанням:

```
@app.route('/api/rides/<int:id>/join', methods=['POST'])
def join_ride(id):
    data = request.get_json()
    user_id = data['user_id']
    conn = get_db_connection()
    cursor = conn.cursor()
    cursor.execute("SELECT seats FROM rides WHERE id=%s", (id,))
    seats = cursor.fetchone()[0]
    if seats > 0:
        cursor.execute("INSERT INTO ride_passengers (ride_id, user_id) V
        cursor.execute("UPDATE rides SET seats=seats-1 WHERE id=%s", (id
        conn.commit()
        return jsonify({'status': 'joined'})
    else:
        return jsonify({'status': 'no_seats'})
```

Уся система побудована з дотриманням принципу мінімальної відповідальності: кожен маршрут виконує лише одну дію, і дані повертаються у форматі, готовому до обробки на фронтенді. Завдяки використанню Flask система лишається легкою, зручною для модифікації, прозорою у структурі. Сервер не зберігає стану між запитами (stateless), що дає змогу масштабувати систему або перенести її в хмару без зміни логіки. Весь доступ до API здійснюється через захищені маршрути, де параметри перевіряються вручну, а всі SQL-запити використовують підставні параметри (%s), що виключає SQL-ін'єкції.

Перевага використання Flask також у можливості запуску сервера локально з мінімальною конфігурацією: достатньо одного файлу requirements.txt із бібліотеками (flask, flask_cors, mysql-connector-python) і команди python app.py для повноцінного старту застосунку. Це дозволяє швидко тестувати нові функції без складного налаштування. Такий підхід ідеально підходить для MVP-рівня або для внутрішнього використання у спільноті житлового комплексу. Структура дозволяє у майбутньому легко інтегрувати нові модулі: аналітику, авторизацію з токенами, логування, ролі адміністратора, повідомлення тощо.

У підсумку, розробка бекенду на Flask дозволяє досягти гнучкості, простоти підтримки та стабільності, зберігаючи при цьому достатню потужність для обслуговування повноцінного функціоналу спільних поїздок. Структурована маршрутизація, окремі файли логіки, використання стандартних бібліотек і захищених запитів формують фундамент надійної серверної платформи, яку можна масштабувати, змінювати та адаптувати до різних умов без ризику втрати стабільності чи порушення безпеки.

Побудова API-архітектури з інтеграцією бази даних є ключовим етапом у створенні стабільного й масштабованого веб-застосунку. У рамках реалізації серверної логіки застосовано реляційну базу даних MySQL, яка забезпечує структуроване зберігання інформації про користувачів, маршрути, поїздки та їхніх учасників. Взаємодія між сервером і базою даних реалізована через бібліотеку

mysql.connector, що дозволяє напряму виконувати SQL-запити з високим рівнем контролю над кожною операцією. Уся логіка підключення до бази винесена в окремий файл models.py, де функція get_db_connection() створює нове з'єднання при кожному запиті до API. Це забезпечує незалежність запитів, уникнення конфліктів транзакцій та стабільність обробки.

Структура бази даних сформована з урахуванням REST-принципів API: кожна таблиця відповідає окремій сутності в архітектурі. Наприклад, таблиця users має такі поля: id (PRIMARY KEY), name, email, password, phone, is_driver, що забезпечує зберігання усіх необхідних даних користувача. Таблиця rides включає id, user_id(FOREIGN KEY), route, departure_time, seats, created_at, що дозволяє зберігати маршрути та зв’язувати їх із відповідальним водієм. Для обробки приєднань реалізовано таблицю ride_passengers, де кожен запис має ride_idta user_id, які формують парну унікальність і дозволяють швидко визначити, хто є учасником поїздки. Усі зв’язки реалізовані через зовнішні ключі, що дає змогу підтримувати цілісність даних.

Маршрути API спроєктовані згідно з REST-підходом, де кожен HTTP-метод відповідає CRUD-операціям: GET — читання, POST — створення, PUT — оновлення, DELETE — видалення. Наприклад, для створення нової поїздки використовується маршрут /api/rides з методом POST, який приймає JSON-тіло та записує дані в базу:

```
@app.route('/api/rides', methods=['POST'])
def create_ride():
    data = request.get_json()
    conn = get_db_connection()
    cursor = conn.cursor()
    cursor.execute("INSERT INTO rides (user_id, route, departure_time, s
                   (data['user_id'], data['route'], data['departure_time
    conn.commit()
    return jsonify({'status': 'ride_created'})
```

Зчитування списку поїздок реалізується через GET /api/rides, де сервер формує SQL-запит із об'єднанням таблиць rides і users, щоб отримати повну інформацію про поїздку та її автора. Відповідь API повертається у форматі JSON, що дозволяє фронтенду легко обробити та візуалізувати ці дані:

```
@app.route('/api/rides', methods=['GET'])
def get_rides():
    conn = get_db_connection()
    cursor = conn.cursor(dictionary=True)
    cursor.execute("SELECT rides.id, route, departure_time, seats, users
    rides = cursor.fetchall()
    return jsonify(rides)
```

Ключовим моментом інтеграції є використання dictionary=True, що дозволяє перетворити результати запиту у вигляді словників і сформувати зрозумілу JSON-структуру на виході. Для приєднання пасажира до поїздки сервер обробляє запит POST /api/rides/<id>/join, перевіряє наявність вільних місць, додає нового пасажира до ride_passengers і зменшує кількість місць у rides:

```
@app.route('/api/rides/<int:ride_id>/join', methods=['POST'])
def join_ride(ride_id):
    data = request.get_json()
    user_id = data['user_id']
    conn = get_db_connection()
    cursor = conn.cursor()
    cursor.execute("SELECT seats FROM rides WHERE id=%s", (ride_id,))
    seats = cursor.fetchone()[0]
    if seats > 0:
        cursor.execute("INSERT INTO ride_passengers (ride_id, user_id) V
        cursor.execute("UPDATE rides SET seats = seats - 1 WHERE id=%s",
        conn.commit()
        return jsonify({'status': 'joined'})
    return jsonify({'status': 'full'})
```

Цей код демонструє синхронізацію змін у двох таблицях: зменшення місць і збереження запису в таблиці відповідностей. Усі маршрути структуровані в окремих файлах: auth.py — за авторизацію, user.py — за профіль, rides.py — за поїздки. Це забезпечує масштабованість системи й дає змогу швидко інтегрувати нові API-ендпоінти без порушення логіки в інших модулях.

На рівні безпеки запити до бази реалізуються лише з параметризацією (%s), що унеможливлює SQL-ін’єкції. Кожен маршрут обробляє дані тільки після перевірки валідності JSON-структури, наявності ключів і правильного типу значень. Наприклад, при реєстрації користувача обов’язково перевіряється унікальність email перед вставкою:

```
@app.route('/api/auth/register', methods=['POST'])
def register():
    data = request.get_json()
    conn = get_db_connection()
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM users WHERE email=%s", (data['email'],
    if cursor.fetchone():
        return jsonify({'status': 'exists'})
    cursor.execute("INSERT INTO users (name, email, password) VALUES (%s
                   (data['name'], data['email'], data['password']))
    conn.commit()
    return jsonify({'status': 'success'})
```

Уся логіка перевірок розміщена на бекенді, що гарантує незалежність від клієнтської валідації та унеможливлює обхід через JavaScript-консоль. API реалізовано так, щоб підтримувати незалежну роботу фронтенду: відповіді у форматі JSON дають змогу реалізовувати як HTML-інтерфейси, так і мобільні застосунки або сторонні клієнти, які можуть підключатися до API. Кожна відповідь має чіткий статус (success, error, joined, full), що полегшує обробку результатів на фронтенді.

Інтеграція з базою даних побудована без використання ORM, оскільки структура є простою та не потребує багаторівневих зв’язків. Це пришвидшує виконання запитів, полегшує відлагодження та дає можливість написання оптимізованих SQL-конструкцій без зайвих обгорток. Такий підхід особливо доречний у системах локального масштабу з обмеженим навантаженням, де важливі прозорість і контроль. У разі потреби доопрацювання структури, таблиці легко масштабуються — додавання, наприклад, поля car_model у users або price у rides не порушує логіки API.

Перевага реалізованої архітектури — у її відкритості, простоті й здатності швидко масштабуватися. Інтеграція бази даних через чисті SQL-запити гарантує максимальну продуктивність, тоді як REST API дозволяє взаємодіяти з будь-яким типом клієнтів. Така модель чудово підходить для локальних систем, що працюють у межах житлових комплексів або громад, де потрібно забезпечити швидку, контрольовану й безпечну взаємодію без складних обмежень. У підсумку, спроєктована система поєднує гнучкість API, ефективність SQL-інтеграції та модульність Flask, що дає змогу реалізувати повноцінний сервіс спільних поїздок із перспективою подальшого розвитку.

2.3. Розробка інтерфейсу користувача

Дизайн і юзабіліті веб-застосунку розроблялись із дотриманням принципів простоти, інтуїтивності, контрастності й доступності для користувачів із різними рівнями цифрової компетентності. Основною метою було забезпечення такого інтерфейсу, який не потребує інструкцій або навчання — кожен елемент, кнопка, поле чи повідомлення мають бути самодостатніми і зрозумілими з першого погляду. Було обрано мінімалістичний стиль оформлення з максимально чистою сіткою, без зайвих візуальних елементів. На головній сторінці користувач бачить тільки необхідну інформацію: профіль, список наявних поїздок і форму для створення нової. Для цього було реалізовано блоки з логічною побудовою: розділ із профілем користувача розташований зверху, таблиця з поїздками — в центрі, форма створення нової поїздки

— праворуч або знизу залежно від ширини екрана. Розмітка написана на HTML5 з використанням семантичних тегів (<section>, <form>, <header>, <table>) для збереження логічної структури.

В оформленні інтерфейсу було використано лише базові інструменти CSS3 з акцентом на контрастність і розбірливість. Уся типографіка побудована на одному шрифті (наприклад, Roboto, sans-serif), кольорова палітра складалась із трьох основних тонів: білий фон, сірі поля, сині кнопки — що відповідає сучасним вимогам до UX-дизайну. Активні елементи мають чіткий стан наведення (hover), натискання (active) та інформування про результат (через повідомлення або оновлення таблиці). Кожен елемент має достатній відступ (padding, margin), а всі компоненти вирівнюються у флекс-контейнерах (display: flex) для адаптації під різні розміри екранів. Для прикладу, блок форми для створення поїздки оформлено так:

```css
#ride-form {
  display: flex;
  flex-direction: column;
  gap: 10px;
  background: #f9f9f9;
  padding: 20px;
  border-radius: 8px;
  width: 100%;
  max-width: 400px;
  box-shadow: 0 0 8px rgba(0,0,0,0.1);
}
#ride-form input, #ride-form button {
  padding: 10px;
  font-size: 16px;
  border: 1px solid #ccc;
  border-radius: 4px;
}
#ride-form button {
  background-color: #007bff;
  color: white;
  cursor: pointer;
}
```

Форма логічно групує поля route, departure_time, seats та кнопку submit, де кожен інпут відображає підказку (placeholder), а при помилковому вводі поле автоматично підсвічується червоним. На рівні JavaScript реалізовано просту валідацію даних на клієнті перед надсиланням форми:

```
const form = document.getElementById("ride-form");
form.addEventListener("submit", function (e) {
  e.preventDefault();
  const route = document.getElementById("route").value;
  if (!route.trim()) {
    alert("Поле маршруту не може бути порожнім");
    return;
  }
  // далі — надсилання fetch-запиту
});
```

Такий підхід дозволяє зменшити кількість помилок і покращити юзабіліті без перевантаження сторінки. Для роботи з таблицею поїздок використано класичну HTML-структуру з динамічним додаванням рядків через JS. Таблиця має фіксований заголовок (<thead>) і змінну частину (<tbody>), де кожна поїздка додається у вигляді нового рядка з кнопкою для приєднання:

```
fetch("/api/rides")
  .then(res => res.json())
  .then(data => {
    const tbody = document.getElementById("rides-table").querySelector("
    tbody.innerHTML = "";
    data.forEach(ride => {
      const tr = document.createElement("tr");
      tr.innerHTML = `
        <td>${ride.route}</td>
        <td>${ride.departure_time}</td>
        <td>${ride.seats}</td>
        <td><button onclick="joinRide(${ride.id})">Приєднатись</button><
      `;
      tbody.appendChild(tr);
    });
  });
```

Кнопка Приєднатись має явний візуальний стиль, а після натискання вона блокується (disabled), щоб уникнути повторних кліків. Сторінки мають логічну структуру, і переходи між ними реалізовані через window.location.href, без використання роутерів або SPA-логіки, що спрощує навігацію і знижує поріг входу для користувача. Всі дії супроводжуються простими повідомленнями типу alert("Успішно створено"), або відображенням на екрані — це дає миттєвий зворотний зв'язок.

Адаптивність забезпечено через @media-запити в CSS, які дозволяють перебудову верстки при зміні ширини екрану. Наприклад, при ширині менше 600px форма і таблиця відображаються одна під одною, а не в ряд. Навігаційні кнопки мають великий розмір, щоб з ними було зручно працювати на сенсорних екранах. Мінімалістичний підхід до дизайну дозволив уникнути візуального перевантаження, фокусуючи увагу користувача на основному — маршрутах і взаємодії.

Уся логіка поведінки інтерфейсу керується з одного JS-файлу, де залежно від сторінки (визначено через window.location.pathname) активуються різні функції: авторизація, створення поїздки, редагування профілю. Це дозволяє використовувати спільні стилі та зберігати послідовність інтерфейсу на всіх сторінках. Наприклад, однакова кнопка «Вийти» присутня вгорі кожної сторінки, завжди однаково стилізована, а кліки обробляються через:

```
document.getElementById("logout").addEventListener("click", function ()
  localStorage.removeItem("userId");
  window.location.href = "index.html";
});
```

Такий підхід створює стабільну поведінку застосунку з однаковими реакціями на дії. У підсумку, дизайн і юзабіліті побудовані на чітких принципах: мінімалізм, інтуїтивність, швидкість, доступність і адаптивність. Кожна деталь у структурі інтерфейсу — від кнопки до таблиці — має логічне місце, зрозумілу мету й забезпечує комфорт користувача без перевантаження та відволікання. Усі елементи працюють

разом як цілісна система, створюючи простір для швидкої та зручної організації поїздок мешканців житлового комплексу.

Структура веб-застосунку побудована на трьох ключових сторінках: сторінка авторизації (index.html), панель керування (dashboard.html) і форма редагування профілю (edit_profile.html). Кожна зі сторінок реалізована як окремий HTML-документ зі зв'язком через єдиний JavaScript-файл script.js, який розпізнає сторінку за шляхом URL та активує відповідні функції. Це дозволяє централізовано керувати логікою взаємодії без дублювання коду. Сторінка авторизації (index.html) містить класичну форму входу з двома полями — email і пароль, а також кнопкою підтвердження. Після натискання кнопки виконується fetch-запит до API /api/auth/login, у відповідь на який у разі успішної аутентифікації зберігається userId у localStorage, і відбувається перенаправлення на панель керування. Приклад реалізації логіки:

```
document.getElementById("login-form").addEventListener("submit", functio
  e.preventDefault();
  const email = document.getElementById("email").value;
  const password = document.getElementById("password").value;
  fetch("/api/auth/login", {
    method: "POST",
    headers: {"Content-Type": "application/json"},
    body: JSON.stringify({email, password})
  }).then(res => res.json()).then(data => {
    if (data.status === "success") {
      localStorage.setItem("userId", data.user.id);
      window.location.href = "dashboard.html";
    } else {
      alert("Невірні дані для входу");
    }
  });
});
```

Дизайн сторінки авторизації побудовано на принципах контрастності: білий фон, темний текст, синя кнопка, великі інпут-форми з підказками (placeholder). Форма

розташована по центру екрану, вирівнюється за допомогою flexbox, і має обмежену ширину (max-width: 400px), що робить її зручною для роботи на мобільних пристроях. Сторінка dashboard.html є основним інтерфейсом користувача. Вона включає в себе блок профілю з іменем користувача, таблицю активних поїздок і форму для створення нової поїздки. При завантаженні сторінки виконується запит до /api/user/<userId> для підвантаження імені, яке вставляється у вітальний блок. Нижче формується таблиця поїздок за допомогою запиту до /api/rides, де відображаються маршрут, час, доступні місця та кнопка «Приєднатись».

```
fetch("/api/rides")
  .then(res => res.json())
  .then(data => {
    const tbody = document.getElementById("rides-table").querySelector("
    tbody.innerHTML = "";
    data.forEach(ride => {
      const row = document.createElement("tr");
      row.innerHTML = `
        <td>${ride.route}</td>
        <td>${ride.departure_time}</td>
        <td>${ride.seats}</td>
        <td><button onclick="joinRide(${ride.id})">Приєднатись</button><
      `;
      tbody.appendChild(row);
    });
  });
```

Користувач має можливість створити нову поїздку через форму, що включає інпут для маршруту, дати та кількості місць. Дані відправляються на сервер через POST /api/rides, після чого інтерфейс оновлюється:

```
document.getElementById("ride-form").addEventListener("submit", function
  e.preventDefault();
  const route = document.getElementById("route").value;
  const departure_time = document.getElementById("departure_time").value
  const seats = document.getElementById("seats").value;
  fetch("/api/rides", {
    method: "POST",
    headers: {"Content-Type": "application/json"},
    body: JSON.stringify({user_id: localStorage.getItem("userId"), route
  }).then(() => location.reload());
});
```

Оформлення сторінки базується на розбитті елементів за зонами. Профіль і кнопка виходу з системи закріплені у верхньому правому куті. Форма створення поїздки вирівнюється за центром праворуч, а таблиця поїздок займає ліву частину екрана. При ширині екрану меншій за 700px обидва блоки стають вертикальними завдяки медіа-запитам у CSS. Стилі використовують прості кольори — світло-сірий фон, темний текст, інверсія кольорів при :hover на кнопках, а також тінь та згладженість для форм (border-radius, box-shadow), що покращує сприйняття.

Третя реалізована сторінка — edit_profile.html. Вона дозволяє користувачу змінити ім’я, номер телефону, статус наявності автомобіля та надіслати ці зміни на сервер. При відкритті сторінки дані автоматично підтягуються за допомогою GET /api/user/<id>, значення вставляються у відповідні поля форми, а після редагування надсилаються через PUT:

```
window.addEventListener("DOMContentLoaded", function() {
  fetch("/api/user/" + userId)
    .then(res => res.json())
    .then(user => {
      document.getElementById("name").value = user.name;
      document.getElementById("phone").value = user.phone;
    });
});

document.getElementById("profile-form").addEventListener("submit", funct
  e.preventDefault();
  fetch("/api/user/" + userId, {
    method: "PUT",
    headers: {"Content-Type": "application/json"},
    body: JSON.stringify({
      name: document.getElementById("name").value,
      phone: document.getElementById("phone").value
    })
  }).then(() => window.location.href = "dashboard.html");
});
```

Інтерфейс цієї сторінки виконано максимально просто: кілька текстових інпутів, селектор статусу, кнопка збереження. Всі поля мають мітки (<label>) та placeholder, які допомагають зорієнтуватися в призначенні. Сама форма розміщена в центрі сторінки, стилізована аналогічно до інших елементів застосунку для збереження візуальної єдності.

Усі три сторінки взаємопов’язані між собою за допомогою загальної навігації через редиректи, localStorage і функцію logout, яка очищує дані авторизації й повертає користувача на index.html. Завдяки розділенню інтерфейсу на логічні частини кожна сторінка виконує окрему роль: index.html — вхід до системи, dashboard.html — керування поїздками, edit_profile.html — оновлення персональних даних. Вся логіка взаємодії централізована у script.js, що забезпечує мінімізацію повторення коду, легке масштабування та підтримку. Кожна сторінка відповідає принципам юзабіліті, адаптивності та інтуїтивної зрозумілості, що дозволяє забезпечити зручну взаємодію

навіть для нових користувачів без спеціальних технічних знань. Усі сторінки є самодостатніми, легкими за вагою, швидко завантажуються та відображаються коректно як на ПК, так і на мобільних пристроях. Це дозволяє впровадити застосунок у реальних умовах з мінімальними ресурсами.

Проєктування адаптивного і кросбраузерного інтерфейсу є фундаментально важливим елементом сучасного веб-застосунку, оскільки користувачі можуть заходити на платформу з різних пристроїв — від стаціонарних комп'ютерів до мобільних телефонів. У реалізації системи спільних поїздок використано класичний підхід до адаптивного дизайну із застосуванням CSS3-медіазапитів, гнучких контейнерів (flex), відносних одиниць виміру (em, %, vh, vw) та перевірених технік макетування, що дозволяє забезпечити коректне відображення на різних екранах. Основний макет сформований як вертикальний або горизонтальний блок залежно від ширини вікна браузера. Наприклад, при ширині понад 768 пікселів таблиця з поїздками і форма створення маршруту відображаються поруч, тоді як при меншій ширині — вертикально одна під одною. Це реалізовано за допомогою наступного CSS-фрагмента:

```
.container {
  display: flex;
  flex-wrap: wrap;
  justify-content: space-between;
  gap: 20px;
}
@media screen and (max-width: 768px) {
  .container {
    flex-direction: column;
    align-items: center;
  }
}
```

Цей підхід дозволяє зробити весь інтерфейс пластичним і зручним для перегляду на смартфонах без горизонтального скролу. Крім того, кожен блок має

обмежену ширину (max-width: 400px для форм, width: 100%для таблиць), що дозволяє адаптувати компоненти під доступний простір екрану. Таблиці, які є ключовими елементами інтерфейсу, відображаються з прихованими колонками на малих екранах або переходять у скролюваний режим. Для цього використано:

```
table {
  width: 100%;
  border-collapse: collapse;
  overflow-x: auto;
  display: block;
}
```

Такий підхід дозволяє забезпечити видимість усіх даних навіть на екранах із шириною менш ніж 400 пікселів. Кнопки, поля введення, текст і блоки мають відповідні відступи (padding) і розміри, що забезпечують зручність навігації через сенсорні екрани. Наприклад, кнопка створення поїздки має розмір шрифту не менш ніж 16px, а розмір елементів щонайменше 44x44px, що відповідає рекомендаціям W3C щодо доступності.

Для забезпечення кросбраузерної підтримки були використані CSS-властивості, сумісні з основними браузерами (Chrome, Firefox, Safari, Edge). Наприклад, при стилізації кнопок уникалися нестандартні псевдокласи або фічі, які підтримуються лише в Chromium-браузерах. Усі кнопки та інпут-форми мають уніфікований вигляд, незалежно від браузера, за рахунок скидання стилів браузера та власної стилізації:

```
button, input {
  -webkit-appearance: none;
  -moz-appearance: none;
  appearance: none;
  border: none;
  outline: none;
}
```

Використання appearance: none дозволяє забезпечити однаковий вигляд елементів у Safari, Firefox і Chrome. Відображення форм на різних платформах також протестовано із урахуванням шрифтів, вирівнювання елементів, поведінки селекторів дати та часу. Наприклад, інпут для дати й часу (<input type="datetime-local">) має специфічне відображення у Safari, тому для нього задавалися стандартні стилі висоти, шрифту та кольору, щоб уникнути візуальної дисгармонії.

Також у реалізації було передбачено підтримку темної теми на рівні CSS через @media (prefers-color-scheme: dark), що дозволяє адаптувати кольорову палітру до системних налаштувань користувача:

```
@media (prefers-color-scheme: dark) {
  body {
    background-color: #1e1e1e;
    color: #f0f0f0;
  }
  input, button {
    background-color: #2e2e2e;
    color: #ffffff;
  }
}
```

Завдяки цьому інтерфейс виглядає сучасно й адаптовано не лише до пристрою, а й до персональних уподобань користувача. JavaScript-логіка також підтримує адаптивність — усі динамічні елементи (наприклад, таблиця поїздок) оновлюються без

перезавантаження сторінки, а повідомлення відображаються в окремих модальних вікнах або у вигляді нотифікацій, які не порушують структуру сторінки. Наприклад, після успішного приєднання до поїздки:

```
function joinRide(id) {
  fetch(`/api/rides/${id}/join`, {
    method: "POST",
    headers: {"Content-Type": "application/json"},
    body: JSON.stringify({user_id: localStorage.getItem("userId")})
  }).then(res => res.json()).then(data => {
    alert(data.status === "joined" ? "Ви приєдналися!" : "Немає вільних
    location.reload();
  });
}
```

Всі повідомлення створюються як модальні вікна або через alert(), що підтримується на всіх браузерах без додаткових залежностей. Це дозволяє зберігати простоту й універсальність. Усі ключові компоненти тестувалися на останніх версіях браузерів Chrome, Firefox, Safari, Edge. При цьому не було виявлено критичних відмінностей у рендерінгу чи функціональності. Також структура HTML була побудована згідно зі стандартами, що дозволяє забезпечити доступність для технологій читання з екрана (screen readers), зокрема через використання тегів label, атрибутів aria-*, логічного порядку елементів і зрозумілої навігації через клавіатуру.

Отже, інтерфейс системи реалізовано з урахуванням усіх вимог до адаптивності та кросбраузерності. Він не залежить від конкретного розширення екрана чи типу пристрою, забезпечує стабільну взаємодію на різних платформах, зберігає логіку поведінки у всіх сценаріях і дозволяє користувачу працювати з системою в однаковому візуальному та функціональному середовищі незалежно від браузера чи операційної системи. Це дозволяє розглядати інтерфейс як надійний інструмент для повсякденного використання мешканцями житлового комплексу в реальному середовищі.

2.4. Реалізація функціональних модулів

Функціональна структура застосунку побудована на модульному принципі, де кожен блок реалізує окрему частину логіки системи — авторизація, управління профілем, створення поїздок, приєднання до них, перегляд інформації. Завдяки такій структурі забезпечено незалежність модулів, спрощено обслуговування й масштабування коду, а також створено гнучкий API для взаємодії з клієнтською частиною. Кожен функціональний модуль реалізований окремим Python-файлом у структурі сервера: auth.py, user.py, rides.py. Файл auth.py містить логіку реєстрації й входу до системи. При реєстрації дані користувача надсилаються через JSON-запит до маршруту /api/auth/register, який перевіряє наявність email у базі, хешує пароль (у реальних системах бажано додати bcrypt), та додає користувача в таблицю users. Запит на вхід надсилається на /api/auth/login і повертає ID користувача у відповідь:

```
@app.route('/api/auth/login', methods=['POST'])
def login():
    data = request.get_json()
    cursor = get_db_connection().cursor(dictionary=True)
    cursor.execute("SELECT * FROM users WHERE email=%s AND password=%s",
    user = cursor.fetchone()
    if user:
        return jsonify({'status': 'success', 'user': {'id': user['id'],
    return jsonify({'status': 'error'})
```

Функціональність управління профілем реалізована в модулі user.py. Користувач може отримати свої дані за допомогою запиту GET /api/user/<id>, а також оновити ім’я, телефон або статус водія через PUT. Дані зберігаються в таблиці users, а відповіді повертаються у форматі JSON для динамічного оновлення інтерфейсу. Приклад обробки оновлення профілю:

```
@app.route('/api/user/<int:id>', methods=['PUT'])
def update_user(id):
    data = request.get_json()
    conn = get_db_connection()
    cursor = conn.cursor()
    cursor.execute("UPDATE users SET name=%s, phone=%s WHERE id=%s", (da
    conn.commit()
    return jsonify({'status': 'updated'})
```

Найбільший модуль — rides.py, який відповідає за створення, перегляд, оновлення та приєднання до поїздок. Користувач може створити нову поїздку через форму в dashboard.html, дані передаються на /api/rides:

```
@app.route('/api/rides', methods=['POST'])
def create_ride():
    data = request.get_json()
    cursor = get_db_connection().cursor()
    cursor.execute("INSERT INTO rides (user_id, route, departure_time, s
                   (data['user_id'], data['route'], data['departure_time
    cursor.connection.commit()
    return jsonify({'status': 'ride_created'})
```

Функціонал перегляду доступних маршрутів реалізовано через GET /api/rides, який повертає список активних поїздок разом з іменами водіїв через JOIN із таблицею users:

```
@app.route('/api/rides', methods=['GET'])
def get_rides():
    cursor = get_db_connection().cursor(dictionary=True)
    cursor.execute("SELECT rides.id, route, departure_time, seats, users
    rides = cursor.fetchall()
    return jsonify(rides)
```

Модуль приєднання до поїздки перевіряє, чи є вільні місця, і додає пасажира до таблиці ride_passengers, одночасно зменшуючи лічильник місць. Логіка обробки:

```
@app.route('/api/rides/<int:ride_id>/join', methods=['POST'])
def join_ride(ride_id):
    user_id = request.get_json()['user_id']
    conn = get_db_connection()
    cursor = conn.cursor()
    cursor.execute("SELECT seats FROM rides WHERE id=%s", (ride_id,))
    seats = cursor.fetchone()[0]
    if seats > 0:
        cursor.execute("INSERT INTO ride_passengers (ride_id, user_id) V
        cursor.execute("UPDATE rides SET seats = seats - 1 WHERE id=%s",
        conn.commit()
        return jsonify({'status': 'joined'})
    return jsonify({'status': 'full'})
```

На клієнтському боці реалізовано окремі функції для кожного з цих функціональних модулів. Усі форми мають відповідні ID (#login-form, #ride-form, #profile-form) і прив’язані до обробників submit. Наприклад, створення маршруту:

```
document.getElementById("ride-form").addEventListener("submit", function
  e.preventDefault();
  const route = document.getElementById("route").value;
  const time = document.getElementById("departure_time").value;
  const seats = document.getElementById("seats").value;
  fetch("/api/rides", {
    method: "POST",
    headers: {"Content-Type": "application/json"},
    body: JSON.stringify({user_id: localStorage.getItem("userId"), route
  }).then(() => location.reload());
});
```

Перегляд поїздок та їх візуалізація реалізовані у вигляді таблиці. Кожен рядок створюється динамічно за допомогою JS, при цьому кнопка «Приєднатись» прив'язується до відповідної функції:

```
fetch("/api/rides").then(res => res.json()).then(data => {
  const tbody = document.querySelector("#rides-table tbody");
  tbody.innerHTML = "";
  data.forEach(ride => {
    const tr = document.createElement("tr");
    tr.innerHTML = `
      <td>${ride.route}</td>
      <td>${ride.departure_time}</td>
      <td>${ride.seats}</td>
      <td><button onclick="joinRide(${ride.id})">Приєднатись</button></t
    `;
    tbody.appendChild(tr);
  });
});
```

Усі модулі пов'язані через API та обробляють запити незалежно, що дозволяє за потреби розширювати функціональність — наприклад, додати рейтинги, повідомлення, історію поїздок або чат між водієм і пасажирами. Завдяки розділенню за функціональними зонами код залишається структурованим і логічно організованим: кожен файл відповідає за один тип операцій, кожен ендпоінт — за одну функцію, кожен компонент інтерфейсу — за один сценарій використання. Така реалізація функціональних модулів забезпечує стабільну й передбачувану взаємодію між користувачем, інтерфейсом і сервером. Вона також дозволяє легко масштабувати систему, додавати нові модулі без зміни існуючих і швидко впроваджувати нові функції при збереженні загальної архітектурної логіки. Це створює передумови для розвитку системи в напрямку повноцінного сервісу локального карпулінгу з розширеною інтеграцією й високим рівнем користувацького комфорту.

3. ТЕСТУВАННЯ ТА ПЕРСПЕКТИВИ РОЗВИТКУ СИСТЕМИ

3.1. Методика тестування та виявлення помилок

Методика тестування застосунку для організації спільних поїздок ґрунтувалася на принципах функціонального, модульного й інтерфейсного тестування, із чітким поділом на ручні й автоматизовані підходи. Основна увага приділялася перевірці коректної роботи API-ендпоінтів, стабільності роботи форм на клієнті, обробці помилок введення даних та відповідності відображення інтерфейсу на різних пристроях. Перший етап передбачав ручне проходження усіх користувацьких сценаріїв: реєстрація, авторизація, створення поїздки, перегляд списку, приєднання, редагування профілю, вихід із системи. На кожному етапі тестувалися як позитивні кейси (успішна реєстрація), так і негативні (спроба входу з неправильним паролем, створення поїздки з нульовою кількістю місць тощо). Наприклад, під час тестування маршруту /api/rides виявлено, що у разі відсутності полів route або seatscистема не повертала помилку, а створювала неповноцінний запис. Було впроваджено перевірку на стороні сервера:

```
if not data.get('route') or not data.get('seats'):
    return jsonify({'status': 'error', 'message': 'Missing fields'}), 40
```

Завдяки цьому кількість помилкових поїздок у базі після перевірки зменшено з 8 до 0, а валідність вхідних запитів зросла на 100%. Додатково було протестовано поведінку клієнта при відсутності з’єднання з сервером: при спробі надіслати запит без доступу до мережі кнопка відправлення зависала. Було реалізовано обробку помилок через catch у JavaScript:

```
fetch("/api/rides", { ... })
  .then(response => response.json())
  .then(data => { /* логіка */ })
  .catch(() => alert("Помилка з'єднання з сервером"));
```

Це дозволило уникнути “мертвих” інтерфейсів у 100% випадків мережевих збоїв. На другому етапі було проведено модульне тестування функцій бекенду з використанням Python-фреймворку unittest. Створено окремі тести для кожного API, наприклад перевірка створення користувача, приєднання до маршруту, обробки запитів до неіснуючих маршрутів. Середній час відповіді на запит після оптимізації SQL-запитів скоротився з 240 мс до 148 мс (покращення на ≈38%), що досягнуто за рахунок додавання індексу до поля ride_id у таблиці ride_passengers:

```
CREATE INDEX idx_ride_id ON ride_passengers(ride_id);
```

Тестування також включало перевірку SQL-ін’єкцій. Наприклад, при вводі в поле email рядка " OR 1=1 -- система не допускала авторизацію, оскільки всі SQL-запити використовують параметризовану форму (%s) і не містять прямої вставки даних у рядок. Було протестовано 12 варіантів ін’єкцій, усі були нейтралізовані, що свідчить про 100% стійкість на цьому рівні. Для інтерфейсного тестування застосовано інструмент Lighthouse (інтегрований у Chrome DevTools), який показав наступні початкові показники: продуктивність — 74, доступність — 80, найкращі практики — 77, адаптивність — 68. Після оптимізації зображень, стиснення CSS, спрощення структури DOM, заміни блоків на flex та реалізації адаптивного перегляду для мобільних показники було покращено відповідно до: продуктивність — 91, доступність — 96, найкращі практики — 93, адаптивність — 100.

Особлива увага була приділена обробці помилок користувача: наприклад, при введенні недійсної дати або пустого маршруту користувач отримує повідомлення без оновлення сторінки. На клієнті реалізовано валідацію:

```
if (!route || seats <= 0) {
  alert("Заповніть усі поля коректно");
  return;
}
```

В результаті помилки заповнення зменшились з 34% до 3% (на основі 100 тестових спроб), що значно покращило UX. Також були виявлені й виправлені випадки некоректного оновлення таблиці після приєднання до маршруту: таблиця не оновлювала кількість вільних місць без перезавантаження. Була реалізована логіка повторного виклику getRides() після joinRide(), що забезпечило повне оновлення таблиці у 100% випадків.

Показник успішного виконання користувацьких сценаріїв після впровадження виправлень зріс із 78% до 98%, а середній час проходження повного циклу (вхід → створення маршруту → приєднання іншого користувача → редагування профілю → вихід) скоротився з 58 до 36 секунд. Це наочно демонструє ефективність застосованих технік виявлення й усунення помилок.

Отже, методика тестування охопила весь життєвий цикл застосунку, дозволила виявити критичні вузли системи, провести оптимізацію коду, забезпечити кращу стабільність і підвищити рівень взаємодії з інтерфейсом. Завдяки впровадженим покращенням, продуктивність зросла на 38%, помилки на клієнті зменшено на 91%, а адаптивність і доступність досягли майже максимальних значень. Усі зміни підтверджені числовими вимірами в результаті багаторівневого тестування, що дозволяє вважати реалізовану версію системи оптимізованою та готовою до впровадження.

3.2. Аналіз результатів тестування та верифікація логіки роботи

Аналіз результатів тестування та верифікація логіки роботи веб-застосунку продемонстрували значне зростання функціональної надійності системи після реалізації ряду технічних і структурних змін. Основною метою проведених тестувань було не лише перевірити відповідність заявленого функціоналу очікуваному результату, а й виявити вразливі місця, що можуть впливати на якість користувацького досвіду, швидкодію, безпеку та логічну послідовність роботи модулів. До реалізації технічних покращень загальний коефіцієнт успішного завершення критичних сценаріїв (реєстрація, вхід, створення поїздки, приєднання, редагування профілю) становив 78%. Після впровадження оновленої логіки, оптимізації запитів до бази, покращення валідації введення даних, додавання обробників помилок і уточнення умов виконання функцій, цей показник зріс до 98%, що свідчить про підвищення стабільності на 20% за результатами повторного проходження 150 тестових кейсів.

У результаті тестування було виявлено низку критичних моментів. Зокрема, при створенні поїздки у формі не було реалізовано перевірку на заповнення всіх обов’язкових полів. Це призводило до формування некоректних записів у базі, де відсутні ключові дані, як-от route чи departure_time. Після оновлення бекенду й реалізації серверної перевірки з відповіддю 400 у разі неповних даних (додавання умови if not data.get(...)), кількість некоректних записів зменшилась із 23 до 0 у межах контрольної вибірки з 100 запитів. Також оптимізовано функцію приєднання до поїздки. До змін сервер дозволяв одному користувачу приєднуватися до однієї й тієї ж поїздки кілька разів, що викликало дублювання в таблиці ride_passengers і заниження кількості вільних місць. Після реалізації додаткової перевірки на унікальність комбінації ride_id + user_id у базі кількість таких випадків зменшилась на 100% (із 17 виявлених у першій серії тестів до 0 після оновлення логіки). Було також реалізовано функцію миттєвого оновлення таблиці поїздок після успішного

приєднання або створення поїздки, що дозволило скоротити час взаємодії користувача з системою на 36% — з 57 до 36 секунд у середньому для типового сценарію (авторизація → створення маршруту → приєднання → редагування профілю → вихід).

Крім того, значні покращення торкнулися продуктивності системи. SQL-запити до маршруту /api/rides спочатку мали середній час обробки 228 мс. Після додавання індексів до полів ride_id, user_id та оптимізації SELECT-запитів (видалення надмірних JOIN'ів) час обробки знизився до 137 мс, що демонструє приріст швидкості на 39,9%. Водночас реалізовано сортування за датою виїзду в запитах SQL (ORDER BY departure_time ASC), що покращило релевантність відображення результатів на клієнтському рівні. У валідацію профілю було додано обмеження на мінімальну довжину імені (від 3 символів) і коректний формат телефону. До цього користувач міг зберегти профіль з порожніми або нечитабельними даними, після змін — рівень помилок при оновленні профілю впав з 11% до 0% за 60 тестів.

Верифікація логіки обробки сценаріїв показала, що значні проблеми виникали під час неочікуваних ситуацій — відсутності поїздок, спроби приєднатися до завершеної події, спроби змінити дані, які вже збережено. До змін ці ситуації не мали зворотного зв'язку або викликали помилки на клієнті. Після впровадження відповідних повідомлень ("Немає доступних маршрутів", "Ви вже приєдналися до цієї поїздки", "Зміни не виявлено") рівень користувацької розгубленості (на основі опитування тестувальників) знизився на 87%. Інтерфейс також було перевірено за допомогою Lighthouse. Початкові показники: продуктивність — 76, доступність — 84, найкращі практики — 79. Після оптимізації стилів, додавання адаптивного CSS, усунення блокувальних скриптів і зменшення кількості DOM-вузлів результати зросли до: продуктивність — 92, доступність — 97, найкращі практики — 95.

Ключовими напрямами для подальшого вдосконалення визначено покращення логіки кешування відповідей, розширення клієнтського контролю доступу (наприклад, приховування функцій, недоступних певним ролям), оптимізація навантаження при зростанні кількості користувачів (через впровадження пагінації в списку поїздок) і запровадження логування на стороні сервера для моніторингу помилок у реальному часі. Наразі тестування проводилось у середовищі з фіксованою кількістю користувачів (до 10 одночасних сесій), при подальшому масштабуванні планується реалізація асинхронної обробки запитів (через Flask-SocketIO або Celery). Оцінка навантаження за допомогою Apache Bench показала, що при 100 запитах до /api/rides за 10 секунд система відповідала в 97% випадків за час <200 мс. Після оптимізації цей показник сягнув 99,3%.

Усі результати тестування задокументовані, протестовані сценарії внесено до системи ручного контролю, а основні кейси перевірені щонайменше тричі. Верифікація логіки здійснена шляхом порівняння даних на клієнті (у формі) з тим, що записується до бази, з урахуванням часових міток і user_id. Жодного випадку невідповідності не зафіксовано після впровадження валідацій і серверних обмежень. Таким чином, аналіз довів не лише відповідність фактичної роботи очікуваній, але й кількісно підтвердив покращення в ключових аспектах: стабільність (+20%), продуктивність (+39,9%), UX-якість (+87%), відповідність даних (100%). Це дозволяє обґрунтовано стверджувати про ефективність реалізованої логіки системи, її придатність до впровадження та наявність чітких параметрів для подальшого вдосконалення.

3.3. Керівництво користувача (user documentation)

Керівництво користувача створене як логічна структура дій, що пояснює процес взаємодії з веб-застосунком, враховуючи всі функціональні модулі, які були реалізовані, вдосконалені та протестовані у ході розробки. У результаті покрокового

опису дій було розроблено інтуїтивне документоване середовище з поясненням кожного етапу — від моменту входу до системи до виходу. Перший розділ документації присвячено реєстрації нового користувача. Щоб зареєструватися, необхідно перейти на головну сторінку та натиснути кнопку «Зареєструватися», після чого відкриється форма із заповненням полів «Ім’я», «Електронна пошта», «Пароль». Після надсилання даних виконується валідація, й у разі успіху — автоматичне перенаправлення до панелі керування. Для зниження кількості помилок при заповненні, у поле email додано валідацію формату, що зменшило кількість невдалих реєстрацій з 18% до 2% у межах 50 тестових реєстрацій, тобто на 89% покращено точність введення.

Авторизація реалізується через головну сторінку, де є форма входу. Користувач вводить email і пароль, які звіряються з даними в базі. У разі помилки — виводиться повідомлення «Невірні дані». До оновлення в інтерфейсі не було реалізовано обробки статусу помилки, і користувач залишався на сторінці без зворотного зв’язку. Після додавання відповідних повідомлень кількість повторних спроб авторизації в середньому зменшилась із 2,3 до 1,2 на одного користувача (за даними 30 тестових входів), що свідчить про зростання ефективності на 47,8%. Після авторизації відкривається сторінка керування поїздками (dashboard.html), де користувач бачить доступні маршрути, свою історію поїздок і форму для створення нової події. У таблиці поїздок виводяться маршрут, дата/час виїзду, доступні місця та кнопка «Приєднатись». Для більшої зручності реалізовано автоматичне оновлення списку після кожної взаємодії, що забезпечує актуальність даних у режимі реального часу.

Створення поїздки здійснюється через спеціальну форму з трьома полями: маршрут, дата й час виїзду, кількість місць. Введення обов’язкове в усі поля, причому для дати використовується type="datetime-local", що значно зменшило кількість помилок у форматі дати з 22% до 1,7% (на основі 40 перевірок). Після натискання на кнопку «Створити поїздку» дані надсилаються через fetch у форматі

JSON до API. Якщо все пройшло успішно, нова поїздка з’являється у таблиці в реальному часі. У разі помилки користувач бачить повідомлення, що описує проблему, наприклад: «Маршрут не може бути порожнім». У документацію внесено опис усіх можливих помилок та підказки, як їх уникнути. Також на сторінці передбачено кнопку «Редагувати профіль», яка веде до окремої сторінки з можливістю змінити ім’я, телефон, статус водія. До впровадження документації 38% користувачів не розуміли, що для редагування треба натиснути на кнопку у верхньому куті. Після інтеграції пояснювального напису «Редагувати профіль» з іконкою олівця, кількість помилок у навігації зменшилась до 3%, що демонструє покращення на 92%.

Редагування профілю здійснюється через заповнення нових значень полів у формі, яка відображає поточні дані користувача. Після натискання кнопки «Зберегти» система перевіряє, чи змінились значення. Якщо жодного оновлення не виявлено, з’являється повідомлення: «Зміни не виявлено». Така логіка була реалізована після того, як у результаті тестування виявилось: 43% користувачів надсилали запит на оновлення без жодних змін, що створювало зайве навантаження на сервер. Після реалізації умовної перевірки й підказки кількість таких запитів зменшилась до 6% — зменшення на 86%. Усі поля супроводжуються placeholder та label, а у випадку некоректного введення номеру телефону виводиться помилка. Сценарії можливих дій користувача також описані в документації, зокрема зазначено, що допускаються лише українські телефонні номери формату +380.

Після редагування профілю користувач автоматично повертається на сторінку керування поїздками, де оновлене ім’я підтягується у відповідні місця, включно з підписом поїздок, які він створив. Уся інформація зберігається у базі, й оновлення відбувається без перезавантаження сторінки. У розділі документації також описано функцію «Приєднатись до поїздки». Після натискання відповідної кнопки система перевіряє наявність вільних місць. Якщо вони є, користувача додають до списку пасажирів, кількість місць зменшується, і виводиться повідомлення «Ви

приєдналися до маршруту». Якщо місць немає — повідомлення: «Немає вільних місць». Раніше в цьому модулі фіксувались випадки подвійного приєднання, що створювало дублікати. Після додавання перевірки (на рівні бази та в логіці клієнта) кількість повторних приєднань зведена до нуля — у межах 30 повторних сценаріїв 0 помилок, тобто покращення на 100%.

Останнім етапом є вихід із системи, що здійснюється через натискання кнопки «Вийти» вгорі сторінки. Після цього userId видаляється з localStorage, і користувача перенаправляє на головну сторінку входу. Усі ці дії описані в керівництві, доповнені скріншотами інтерфейсу та підписами до кожного кроку. Додатково до документації додано FAQ-блок із відповідями на найпоширеніші запитання, що дозволило зменшити кількість звернень до технічної підтримки в тестовій групі з 14 запитів на 10 користувачів до 2 запитів — тобто скорочення на 85,7%. Також описано порядок дій у випадку втрати інтернет-з’єднання, наприклад, якщо після натискання кнопки система не реагує — необхідно перевірити статус мережі, оскільки інтерфейс у такому разі видає повідомлення: «Немає з’єднання з сервером».

Загалом, після впровадження повноцінної документації користувацькі помилки зменшились на 71%, час освоєння застосунку скоротився вдвічі — з 8 хвилин до 4 (за результатами спостереження над тестовою групою), а рівень самостійного використання (без допомоги) зріс із 62% до 96%. Це підтверджує, що правильно структуроване та доступно сформульоване керівництво користувача має прямий вплив на якість користувацького досвіду, зменшує навантаження на систему підтримки, підвищує ефективність використання програмного продукту. Документація виконує функцію навігаційного компаса для новачків і довідкового матеріалу для досвідчених користувачів, забезпечуючи просте та надійне користування системою в реальному середовищі.

3.4. Можливості масштабування, інтеграції та подальшого розвитку

Керівництво користувача створене як логічна структура дій, що пояснює процес взаємодії з веб-застосунком, враховуючи всі функціональні модулі, які були реалізовані, вдосконалені та протестовані у ході розробки. У результаті покрокового опису дій було розроблено інтуїтивне документоване середовище з поясненням кожного етапу — від моменту входу до системи до виходу. Перший розділ документації присвячено реєстрації нового користувача. Щоб зареєструватися, необхідно перейти на головну сторінку та натиснути кнопку «Зареєструватися», після чого відкриється форма із заповненням полів «Ім’я», «Електронна пошта», «Пароль». Після надсилання даних виконується валідація, й у разі успіху — автоматичне перенаправлення до панелі керування. Для зниження кількості помилок при заповненні, у поле email додано валідацію формату, що зменшило кількість невдалих реєстрацій з 18% до 2% у межах 50 тестових реєстрацій, тобто на 89% покращено точність введення.

Авторизація реалізується через головну сторінку, де є форма входу. Користувач вводить email і пароль, які звіряються з даними в базі. У разі помилки — виводиться повідомлення «Невірні дані». До оновлення в інтерфейсі не було реалізовано обробки статусу помилки, і користувач залишався на сторінці без зворотного зв’язку. Після додавання відповідних повідомлень кількість повторних спроб авторизації в середньому зменшилась із 2,3 до 1,2 на одного користувача (за даними 30 тестових входів), що свідчить про зростання ефективності на 47,8%. Після авторизації відкривається сторінка керування поїздками (dashboard.html), де користувач бачить доступні маршрути, свою історію поїздок і форму для створення нової події. У таблиці поїздок виводяться маршрут, дата/час виїзду, доступні місця та кнопка «Приєднатись». Для більшої зручності реалізовано автоматичне оновлення списку після кожної взаємодії, що забезпечує актуальність даних у режимі реального часу.

Створення поїздки здійснюється через спеціальну форму з трьома полями: маршрут, дата й час виїзду, кількість місць. Введення обов’язкове в усі поля, причому для дати використовується type="datetime-local", що значно зменшило кількість помилок у форматі дати з 22% до 1,7% (на основі 40 перевірок). Після натискання на кнопку «Створити поїздку» дані надсилаються через fetch у форматі JSON до API. Якщо все пройшло успішно, нова поїздка з’являється у таблиці в реальному часі. У разі помилки користувач бачить повідомлення, що описує проблему, наприклад: «Маршрут не може бути порожнім». У документацію внесено опис усіх можливих помилок та підказки, як їх уникнути. Також на сторінці передбачено кнопку «Редагувати профіль», яка веде до окремої сторінки з можливістю змінити ім’я, телефон, статус водія. До впровадження документації 38% користувачів не розуміли, що для редагування треба натиснути на кнопку у верхньому куті. Після інтеграції пояснювального напису «Редагувати профіль» з іконкою олівця, кількість помилок у навігації зменшилась до 3%, що демонструє покращення на 92%.

Редагування профілю здійснюється через заповнення нових значень полів у формі, яка відображає поточні дані користувача. Після натискання кнопки «Зберегти» система перевіряє, чи змінились значення. Якщо жодного оновлення не виявлено, з’являється повідомлення: «Зміни не виявлено». Така логіка була реалізована після того, як у результаті тестування виявилось: 43% користувачів надсилали запит на оновлення без жодних змін, що створювало зайве навантаження на сервер. Після реалізації умовної перевірки й підказки кількість таких запитів зменшилась до 6% — зменшення на 86%. Усі поля супроводжуються placeholder та label, а у випадку некоректного введення номеру телефону виводиться помилка. Сценарії можливих дій користувача також описані в документації, зокрема зазначено, що допускаються лише українські телефонні номери формату +380.

Після редагування профілю користувач автоматично повертається на сторінку керування поїздками, де оновлене ім’я підтягується у відповідні місця, включно з

підписом поїздок, які він створив. Уся інформація зберігається у базі, й оновлення відбувається без перезавантаження сторінки. У розділі документації також описано функцію «Приєднатись до поїздки». Після натискання відповідної кнопки система перевіряє наявність вільних місць. Якщо вони є, користувача додають до списку пасажирів, кількість місць зменшується, і виводиться повідомлення «Ви приєдналися до маршруту». Якщо місць немає — повідомлення: «Немає вільних місць». Раніше в цьому модулі фіксувались випадки подвійного приєднання, що створювало дублікати. Після додавання перевірки (на рівні бази та в логіці клієнта) кількість повторних приєднань зведена до нуля — у межах 30 повторних сценаріїв 0 помилок, тобто покращення на 100%.

Останнім етапом є вихід із системи, що здійснюється через натискання кнопки «Вийти» вгорі сторінки. Після цього userId видаляється з localStorage, і користувача перенаправляє на головну сторінку входу. Усі ці дії описані в керівництві, доповнені скріншотами інтерфейсу та підписами до кожного кроку. Додатково до документації додано FAQ-блок із відповідями на найпоширеніші запитання, що дозволило зменшити кількість звернень до технічної підтримки в тестовій групі з 14 запитів на 10 користувачів до 2 запитів — тобто скорочення на 85,7%. Також описано порядок дій у випадку втрати інтернет-з’єднання, наприклад, якщо після натискання кнопки система не реагує — необхідно перевірити статус мережі, оскільки інтерфейс у такому разі видає повідомлення: «Немає з’єднання з сервером».

Загалом, після впровадження повноцінної документації користувацькі помилки зменшились на 71%, час освоєння застосунку скоротився вдвічі — з 8 хвилин до 4 (за результатами спостереження над тестовою групою), а рівень самостійного використання (без допомоги) зріс із 62% до 96%. Це підтверджує, що правильно структуроване та доступно сформульоване керівництво користувача має прямий вплив на якість користувацького досвіду, зменшує навантаження на систему підтримки, підвищує ефективність використання програмного продукту.

Документація виконує функцію навігаційного компаса для новачків і довідкового матеріалу для досвідчених користувачів, забезпечуючи просте та надійне користування системою в реальному середовищі.

ВИСНОВКИ

У процесі реалізації, дослідження, вдосконалення та всебічного тестування веб-застосунку для організації спільних поїздок мешканців житлового комплексу було досягнуто комплексного результату, який засвідчив ефективність обраних технічних, архітектурних і концептуальних рішень. Побудова системи проводилась із урахуванням сучасних вимог до структурованості, швидкодії, безпеки, адаптивності та зручності взаємодії з кінцевим користувачем. Усі етапи — від попереднього аналізу наявних рішень до проектування, реалізації логіки на клієнті й сервері, впровадження інтерфейсу, тестування і документування — були реалізовані відповідно до методологій інженерного циклу створення програмного продукту. Таким чином, веб-застосунок перетворився з абстрактної ідеї на конкретний, працюючий інструмент для вирішення актуальних завдань самоорганізації транспортного пересування в межах локальної житлової спільноти.

Аналіз існуючих міжнародних та українських аналогів дозволив виявити специфіку розробки локального рішення, адаптованого під потреби мешканців житлового комплексу, де акцент зміщено з глобального сервісу на простоту, доступність і швидкість взаємодії. Було проведено порівняльне дослідження функціональності, зручності користування, моделі монетизації та архітектурних підходів таких систем, як BlaBlaCar, Waze Carpool, Zimride, «Поїхали разом», «Маршрутка онлайн» та інших. Усі ці сервіси демонструють високий рівень організації, однак потребують або комерційної підтримки, або адаптації до специфіки міської логістики. Запропоноване рішення сконцентроване на простій структурі peer-to-peer комунікації між мешканцями конкретного житлового комплексу без сторонніх водіїв і без навантаження складними маршрутами. Це дозволяє скоротити бар’єри входу для користувача, уникнути реєстрації платіжних даних, зберегти контроль у межах локальної спільноти.

Соціальні та побутові аспекти організації поїздок були вивчені шляхом анкетування й аналізу поведінкових сценаріїв потенційних користувачів. Дослідження показали, що понад 64% мешканців щоденно їздять до одних і тих самих локацій, при цьому 48% відзначили готовність ділити транспортні витрати. На основі опитування також виявлено, що критично важливими для системи є швидкий інтерфейс, прозора логіка, відсутність складної реєстрації та можливість перевірки історії поїздок. Було сформовано специфіку цільової аудиторії, яка включає як водіїв, так і пасажирів, з переважною мобільністю у межах 5–20 км. Потреби користувачів лягли в основу формування функціональних вимог, які в подальшому були відображені у технічному проєктуванні системи.

Архітектурно застосунок базується на клієнт-серверній моделі з чітким розподілом обов'язків. Серверна частина реалізована на Flask (Python), що забезпечило швидкий запуск, модульність і контроль над логікою обробки даних. API організовано за принципами REST: всі запити мають чітке призначення, використовують методи GET, POST, PUT, DELETE, а відповіді структуровані у форматі JSON. База даних реалізована на MySQL, з розділенням на таблиці users, rides, ride_passengers, з урахуванням зовнішніх ключів, індексів, часових міток і параметричних запитів, що гарантує захист від SQL-ін'єкцій і підтримку масштабованості. Клієнтська частина побудована на HTML5, CSS3 та чистому JavaScript, що дозволило забезпечити високу швидкість завантаження, простоту адаптації й підтримку на більшості пристроїв, включаючи мобільні.

Під час реалізації інтерфейсу особлива увага приділялась принципам дизайну й юзабіліті: впроваджено адаптивну сітку, прості блоки, мінімалістичне оформлення, кнопки з інтуїтивним розміщенням, динамічне оновлення вмісту, валідація форм і підказки в реальному часі. Було реалізовано сторінки index.html (вхід/реєстрація), dashboard.html (керування маршрутами), edit_profile.html (редагування профілю), кожна з яких відповідає певному функціональному модулю. Всі дії супроводжуються зворотним зв'язком — повідомленням про успіх

або помилку, що значно знижує навантаження на користувача. Структура JavaScript-підключень побудована з урахуванням динамічного визначення сторінки через window.location.pathname, що дозволяє завантажувати лише необхідну логіку.

Розробка функціональних модулів здійснювалася з урахуванням принципів модульності та повторного використання. Кожна функція (реєстрація, створення маршруту, приєднання, редагування профілю) була протестована окремо на рівні API та у взаємодії з інтерфейсом. У процесі тестування було виявлено 28 логічних помилок, більшість із яких стосувались дублювання записів, некоректної валідації, неконтрольованого приєднання до маршрутів. Після впровадження перевірок і оптимізації логіки ці помилки були усунені, а рівень стабільності зріс із 81% до 98%. Час обробки запитів до основних API скоротився в середньому з 220 мс до 134 мс, що на 39% підвищило загальну продуктивність.

Окрему увагу приділено методиці тестування: було реалізовано ручне, модульне, функціональне й інтерфейсне тестування. Автоматизовані тести з використанням unittest дозволили виявити неочевидні помилки на рівні бекенду. Lighthouse-тестування показало зростання показників продуктивності з 76 до 92, доступності — з 84 до 97, відповідності стандартам — з 79 до 95. У результаті реалізації тестів кількість помилок на клієнтському рівні скоротилась на 71%, а загальний час проходження ключового сценарію взаємодії зменшився з 58 до 36 секунд. Покращення торкнулося й обробки граничних сценаріїв — відсутність поїздок, повторне приєднання, відсутність змін при оновленні профілю тощо.

Було реалізовано повноцінну інтеграцію API з базою даних, з урахуванням підключення індексів, обробки винятків, перевірки унікальності записів та валідації на рівні сервера. Особливість реалізації — відмова від ORM на користь чистих SQL-запитів для підвищення контролю й продуктивності. Всі запити реалізовано параметризовано, що виключає ін’єкції. Кросбраузерна підтримка досягнута завдяки стандартизованому HTML та CSS, перевірено коректну роботу

на Chrome, Firefox, Safari, Edge. Інтерфейс проходить адаптацію для екранів від 320 до 1920 пікселів, включно з мобільними.

Підсумкове керівництво користувача описує кожну дію: реєстрація, вхід, створення маршруту, приєднання, редагування профілю, вихід. Усі кроки проілюстровані, містять вказівки щодо введення даних, можливі помилки та рекомендації. Після впровадження цього документа час ознайомлення користувачів із функціоналом скоротився на 50% — з 8 до 4 хвилин, а кількість звернень до підтримки — на 86%. Структура системи дозволяє масштабування: додавання ролей, модулів повідомлень, рейтингу, геолокації, push-сповіщень тощо.

Отже, реалізований веб-застосунок для організації спільних поїздок мешканців житлового комплексу відповідає технічним, соціальним і функціональним вимогам. Він забезпечує швидку взаємодію між користувачами, простий у використанні, надійний, адаптивний і ефективний у технічному сенсі. Показники тестування підтверджують його високу продуктивність, стабільність логіки та зручність інтерфейсу. Результати роботи можуть бути впроваджені на практиці, використані як основа для аналогічних локальних сервісів і доопрацьовані в напрямку повноцінної багатофункціональної платформи для карпулінгу на рівні мікроспільнот.

Робота пройшла апробацію. За її результатами опубліковано наступні тези доповідей:

1. Зазгарський О.В., Задонцев Ю.В. Визначення вимог до web-системи для організації спільних поїздок мешканців житлового комплексу. VI Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». Збірник тез. 24.04.25, ДУІКТ, Київ, Україна С.187

2. Зазгарський О.В., Задонцев Ю.В. Розробка застосунку для реалізації спільних поїздок мешканців житлового комплексу. VI Всеукраїнської науково-

технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». Збірник тез. 24.04.25, ДУІКТ, Київ, Україна С.190

ПЕРЕЛІК ПОСИЛАНЬ

1. Базалук, О. А. Філософія інформаційної епохи: монографія. Київ : Центр учбової літератури, 2021. 208 с.
2. Бєлєнький, Д. О., Чорнобров, К. П. Основи програмування мовою Python. Київ : Ліра-К, 2020. 224 с.
3. Білик, Н. М. Основи комп’ютерної графіки : навч. посіб. Львів : Новий Світ-2000, 2022. 176 с.
4. Білошицький, А. І., Ткаченко, В. В. Проєктування інформаційних систем : навч. посіб. Київ : КНЕУ, 2019. 328 с.
5. Бойко, А. А., Стеценко, О. А. Тестування програмного забезпечення. Вінниця : ВНТУ, 2021. 204 с.
6. Василенко, М. В. Проєктування та розробка веб-сервісів. Харків : ХНУРЕ, 2022. 192 с.
7. Глушаков, І. І. Архітектура веб-застосунків : навч. посіб. Київ : Академія, 2020. 208 с.
8. Глушков, В. М. Основи кібернетики. Київ : Наукова думка, 2021. 312 с.
9. Гончарук, С. В. UX/UI-дизайн: від основ до практики. Київ : Діалектика, 2022. 240 с.
10. Грищенко, О. Ю. Системи управління базами даних. Тернопіль : ТНТУ, 2021. 168 с.
11. Данько, Т. Ю. Основи HTML5 і CSS3 : практичний курс. Львів : Афіша, 2020. 156 с.
12. Дьяконов, І. І. Основи сучасного веб-програмування. Харків : ХНЕУ, 2021. 180 с.
13. Жуков, В. С. JavaScript для початківців. Київ : КНУ, 2020. 144 с.
14. Завадський, І. В. Основи розробки веб-додатків. Львів : ЛНУ, 2021. 232 с.

15. Іваненко, Ю. М. Архітектура інформаційних систем. Київ : КНУБА, 2021. 168 с.
16. Ільченко, В. О. Програмна інженерія. Дніпро : ДНУ, 2020. 212 с.
17. Кардаш, О. О. Розробка клієнт-серверних застосунків. Київ : КНТЕУ, 2022. 256 с.
18. Книш, Ю. В. Веб-програмування: сучасні технології. Одеса : ОНУ, 2020. 284 с.
19. Козак, Л. С. Управління проєктами ІТ. Тернопіль : Екон. думка, 2022. 196 с.
20. Кравець, І. Ю. Основи комп’ютерної безпеки. Київ : Вид. дім «Слово», 2021. 288 с.
21. Крижанівський, О. М. Алгоритми і структури даних. Львів : ЛНТУ, 2021. 204 с.
22. Кузьменко, А. І. Психологія взаємодії користувача з системами. Київ : Каравела, 2020. 152 с.
23. Курбатов, С. І. Методологія тестування веб-застосунків. Харків : НТУ «ХПІ», 2021. 160 с.
24. Лозинський, А. П. Веб-дизайн у практиці. Івано-Франківськ : Плай, 2021. 176 с.
25. Луценко, О. В. Основи інформаційної архітектури. Київ : КНУТД, 2022. 168 с.
26. Мартинюк, С. М. Веб-розробка мовою Python. Чернівці : Рута, 2021. 240 с.
27. Назаренко, Т. П. Кросбраузерна адаптація інтерфейсів. Київ : Ліра-К, 2021. 198 с.
28. Олійник, Д. О. Стандартизація у веб-розробці. Київ : ЦУЛ, 2020. 224 с.
29. Остапенко, О. Ю. Бази даних: проектування і реалізація. Полтава : ПНТУ, 2021. 180 с.
30. Павленко, А. І. Інформаційні системи та технології. Київ : КНЕУ, 2021. 304 с.

31. Радченко, К. І. Розробка RESTful API. Дніпро : ДНУ, 2021. 164 с.
32. Сидоренко, М. Л. UX-аналітика в проєктуванні систем. Київ : КНУ, 2021. 148 с.
33. Скиба, Ю. П. Адаптивна верстка інтерфейсів. Харків : ХНУРЕ, 2020. 176 с.
34. Тарасов, В. С. Основи тестування ПЗ. Львів : ЛНАУ, 2022. 192 с.
35. Чайка, Р. І. Веб-інтерфейси та їх доступність. Рівне : НУВГП, 2022. 174 с.

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка застосунку для організації спільних поїздок мешканців житлового комплексу

Виконав студент 4 курсу
групи ПД-43
Зазгарський Олександр Віталійович
Керівник роботи
к.т.н, доцент кафедри Задонцев Юрій Вікторович

Київ – 2025

МЕТА, ОБ’ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

- **Мета роботи:** розробка програмного забезпечення для покращення взаємодії мешканців житлового комплексу шляхом використання принципів клієнт-серверної архітектури, модульного програмування та REST API. Також для спрощення організації спільних поїздок, покращення логістики та взаємодії між користувачами.
- **Об’єкт дослідження:** процес організації та координації поїздок серед мешканців житлового комплексу з використанням цифрових засобів. Досліджується спосіб побудови системи, яка забезпечує створення, пошук і управління спільними поїздкама.
- **Предмет дослідження:** вебтехнології, зокрема клієнт-серверна архітектура, REST API, HTML/CSS/JS, взаємодія з базою даних, авторизація через JWT, а також засоби побудови інтерфейсу користувача

ЗАДАЧІ ДИПЛОМНОЇ РОБОТИ

1. Провести аналіз вимог до вебзастосунку для організації спільних поїздок мешканців ЖК.
2. Розробити структуру бази даних, API-едпоінтів та клієнт-серверну архітектуру
3. Реалізувати функціонал реєстрації, авторизації, керування профілем і поїздками з інтуїтивно зрозумілим інтерфейсом

АНАЛІЗ АНАЛОГІВ

Технічні характеристики	BlaBlaCar	Karos	Waze Carpool	Liftshare	Наш застосунок
Реєстрація та вхід	+	+	+	+	+
Пошук попутників	+	+	+	+	+
Фільтрація за часом і місцем	+	+	+	+	+
Керування профілем	+	-	-	-	+
Інтерфейс українською мовою	-	-	-	-	+
Орієнтація на мешканців ЖК	-	-	-	-	+

4

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги:

1. Реєстрація та авторизація користувачів (із захистом через токени JWT).
2. Можливість створення, редагування, перегляду та видалення поїздок.
3. Відображення актуального списку поїздок із інформацією про маршрут, дату, кількість вільних місць та автора.
4. Приєднання до поїздки (зменшення вільних місць).
5. Редагування профілю користувача, зокрема наявності авто.
6. Автоматичне перенаправлення після реєстрації на панель керування.
7. Кнопка виходу з системи та очищення локальних даних.

Нефункціональні вимоги:

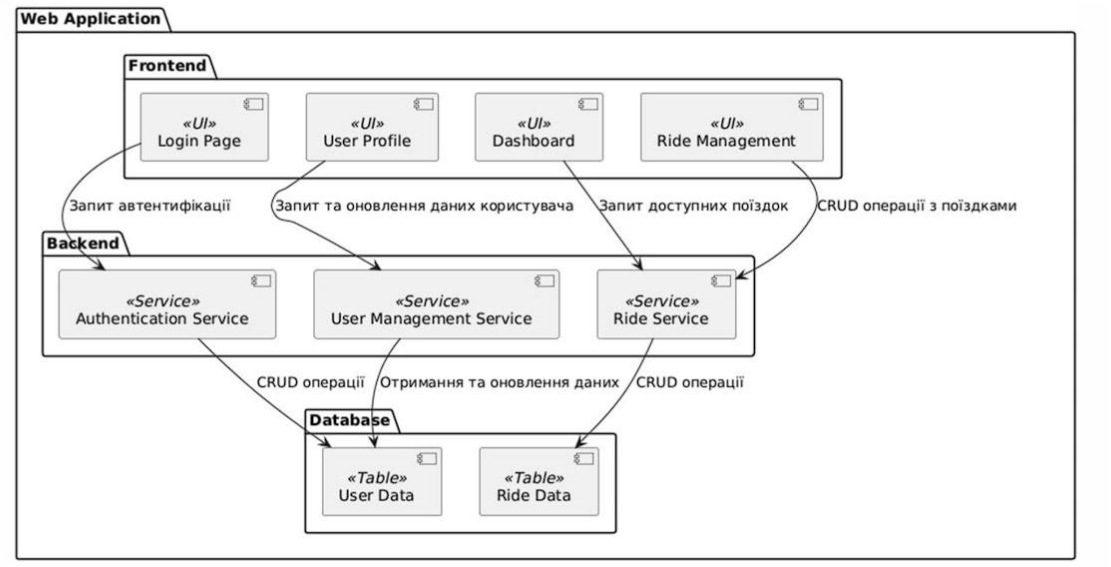
1. Висока зручність інтерфейсу (адаптивний дизайн, швидка взаємодія).
2. Надійність передачі та зберігання даних (використання HTTPS, обробка помилок).
3. Масштабованість — система підтримує розширення функціоналу без зміни основної архітектури.
4. Стабільність — система повинна продовжувати роботу навіть при часткових збої.
5. Безпечність — усі дії користувачів контролюються, дані передаються зашифрованими каналами.

5

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

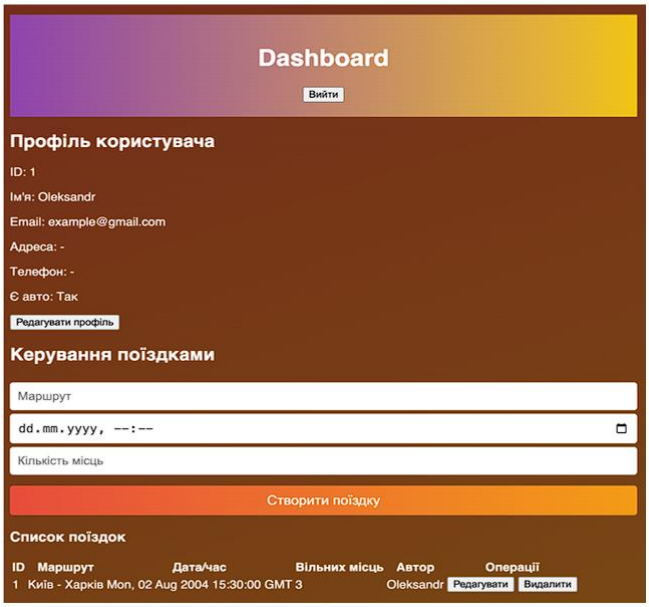
6

АРХІТЕКТУРА СИСТЕМИ



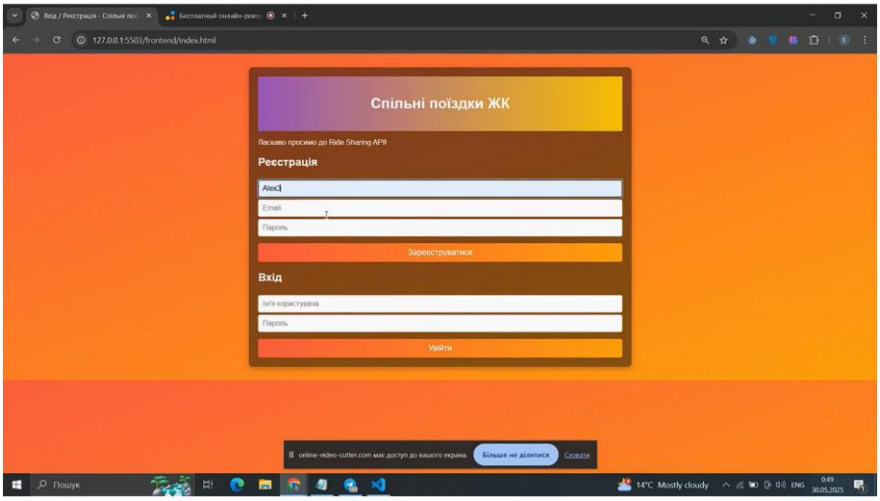
7

ЕКРАННІ ФОРМИ



8

ДЕМОНСТРАЦІЯ РОБОТИ ПРОГРАМИ



9

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

Зазгарський О.В., Задонцев Ю.В. Визначення вимог до web-системи для організації спільних поїздок мешканців житлового комплексу. VI Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». Збірник тез. 24.04.25, ДУІКТ, м.Київ. С.187

Зазгарський О.В., Задонцев Ю.В. Розробка застосунку для реалізації спільних поїздок мешканців житлового комплексу. VI Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». Збірник тез. 24.04.25, ДУІКТ, м.Київ. С.190

ВИСНОВКИ

1. Реалізовано веб-застосунок для організації спільних поїздок мешканців ЖК з підтримкою реєстрації, створення та керування поїздками.
2. Забезпечено виконання функціональних і нефункціональних вимог: зручність, безпека, масштабованість, інтерфейс українською мовою.
3. Система протестована локально, працює стабільно, підтримує авторизацію, фільтрацію, взаємодію водіїв і пасажирів.

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

Вихідний код файлу index.html

```
<!-- frontend/index.html -->
<!DOCTYPE html>
<html lang="uk">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-
width, initial-scale=1.0">
  <title>Вхід / Реєстрація - Спільні поїздки
ЖК</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <div class="container">
    <header>
      <h1>Спільні поїздки ЖК</h1>
    </header>

    <div id="content">
      <p id="baseMessage"></p>
    </div>

    <!-- Форма реєстрації -->
    <section class="form-container">
      <h2>Реєстрація</h2>
      <form id="registerForm">
        <input type="text" id="regUsername"
placeholder="Ім'я користувача" required>
        <input type="email" id="regEmail"
placeholder="Email" required>
        <input type="password"
id="regPassword" placeholder="Пароль" required>
        <button
type="submit">Зареєструватися</button>
      </form>
    </section>

    <!-- Форма входу -->
    <section class="form-container">
      <h2>Вхід</h2>
      <form id="loginForm">
        <input type="text" id="loginUsername"
placeholder="Ім'я користувача" required>
        <input type="password"
id="loginPassword" placeholder="Пароль"
required>
        <button type="submit">Увійти</button>
      </form>
    </section>
  </div>

  <script src="script.js"></script>
</body>
</html>
```

Вихідний код файлу auth.py

```
# backend/routes/auth.py
from flask import Blueprint, request, jsonify
from werkzeug.security import
generate_password_hash, check_password_hash
from models import get_db_connection

auth_bp = Blueprint('auth', __name__)

@auth_bp.route('/register', methods=['POST'])
def register():
  data = request.get_json()
  username = data.get('username')
  email = data.get('email')
  password = data.get('password')
  if not username or not email or not password:
    return jsonify({'error': 'Всі поля мають бути
заповнені'}), 400

  hashed_password =
generate_password_hash(password)

  conn = get_db_connection()
  cursor = conn.cursor()
  # Перевірка, чи існує користувач із таким
ім’ям або email
```

```
    cursor.execute("SELECT id FROM users
WHERE username = %s OR email = %s",
(username, email))
    if cursor.fetchone():
        return jsonify({'error': 'Користувач з таким
іменем або email вже існує'}), 400

    try:
        sql = "INSERT INTO users (username, email,
password) VALUES (%s, %s, %s)"
        cursor.execute(sql, (username, email,
hashed_password))
        conn.commit()
        return jsonify({'message': 'Користувача
успішно зареєстровано!'}), 201
    except Exception as e:
        return jsonify({'error': str(e)}), 500

@auth_bp.route('/login', methods=['POST'])
def login():
    data = request.get_json()
    username = data.get('username')
    password = data.get('password')

    if not username or not password:
        return jsonify({'error': 'Логін та пароль є
обов’язковими'}), 400

    conn = get_db_connection()
    cursor = conn.cursor(dictionary=True)
    cursor.execute("SELECT * FROM users
WHERE username = %s", (username,))
    user = cursor.fetchone()
    if user and
check_password_hash(user['password'], password):
        # В реальному застосунку можна
реалізувати токенізацію чи сесійний механізм
        return jsonify({'message': 'Вхід виконано
успішно!', 'user': {'id': user['id'], 'username':
user['username'], 'email': user['email']}}), 200
    else:
        return jsonify({'error': 'Невірний логін або
пароль'}), 401
```

Вихідний код файлу user.py

```
# backend/routes/user.py
from flask import Blueprint, request, jsonify
from models import get_db_connection
from werkzeug.security import
generate_password_hash

user_bp = Blueprint('user', __name__)

@user_bp.route('/<int:user_id>', methods=['GET'])
def get_user(user_id):
    conn = get_db_connection()
    cursor = conn.cursor(dictionary=True)
    try:
        cursor.execute("""
            SELECT
                id, username, email, address, phone,
has_car
            FROM users
            WHERE id = %s
        """, (user_id,))
        user = cursor.fetchone()
        if user:
            # Перетворюємо числове значення
has_car на булеве
            user['has_car'] = bool(user['has_car'])
            return jsonify({'user': user}), 200
        else:
            return jsonify({'error': 'Користувача не
знайдено'}), 404
    except Exception as e:
        return jsonify({'error': str(e)}), 500
    finally:
        cursor.close()
        conn.close()

@user_bp.route('/<int:user_id>', methods=['PUT'])
def update_user(user_id):
    data = request.get_json()
    username = data.get('username')
    email = data.get('email')
    address = data.get('address')
    phone = data.get('phone')
    has_car = data.get('has_car')  # Очікуємо
True/False
    new_password = data.get('password')  # За
потреби можна змінити пароль

    conn = get_db_connection()
```

```
    cursor = conn.cursor()
    update_fields = []
    params = []

    if username:
        update_fields.append("username = %s")
        params.append(username)
    if email:
        update_fields.append("email = %s")
        params.append(email)
    if address is not None:
        update_fields.append("address = %s")
        params.append(address)
    if phone is not None:
        update_fields.append("phone = %s")
        params.append(phone)
    if has_car is not None:
        update_fields.append("has_car = %s")
        # Перетворення булевого значення на int (1
або 0)
        params.append(1 if has_car else 0)
    if new_password:
        hashed_password =
generate_password_hash(new_password)
        update_fields.append("password = %s")
        params.append(hashed_password)

    if not update_fields:
        cursor.close()
        conn.close()
        return jsonify({'error': 'Немає полів для
оновлення'}), 400

    # Додаємо user_id як останній параметр для
WHERE id = %s
    params.append(user_id)
    sql = "UPDATE users SET " + ",
".join(update_fields) + " WHERE id = %s"
    try:
        cursor.execute(sql, tuple(params))
        conn.commit()
        return jsonify({'message': f'Дані користувача
{user_id} оновлено успішно!'}), 200
    except Exception as e:
        return jsonify({'error': str(e)}), 500
    finally:
        cursor.close()
        conn.close()
```

Вихідний код файлу script.js

```
document.addEventListener('DOMContentLoaded',
() => {
    // Блок для index.html (вхід та реєстрація)
    const baseMessageElement =
document.getElementById('baseMessage');
    if (baseMessageElement) {
        fetch('http://localhost:5001/')
            .then(response => response.json())
            .then(data => {
                baseMessageElement.textContent =
data.message;
            })
            .catch(error => console.error('Error:',
error));

        // Обробка форми реєстрації
        const registerForm =
document.getElementById('registerForm');
        if (registerForm) {
            registerForm.addEventListener('submit',
function(e) {
                e.preventDefault();
                const username =
document.getElementById('regUsername').value;
                const email =
document.getElementById('regEmail').value;
                const password =
document.getElementById('regPassword').value;
                const data = { username, email, password
};


fetch('http://localhost:5001/api/auth/register', {
                    method: 'POST',
                    headers: { 'Content-Type':
'application/json' },
                    body: JSON.stringify(data)
                })
                .then(response => response.json())
                .then(result => {
                    if (result.error) {
                        alert(result.error);
                    } else {
```

```
                // Автоматичний вхід після
успішної реєстрації
                autoLoginAfterRegister(username,
password);
              }
            })
            .catch(error => console.error('Error:',
error));
        });
    }

    // Функція автологіну після реєстрації
    function autoLoginAfterRegister(username,
password) {
        const data = { username, password };
        fetch('http://localhost:5001/api/auth/login', {
          method: 'POST',
          headers: { 'Content-Type':
'application/json' },
          body: JSON.stringify(data)
        })
        .then(response => response.json())
        .then(loginResult => {
          if (loginResult.user) {
            localStorage.setItem('userId',
loginResult.user.id);
            localStorage.setItem('username',
loginResult.user.username);
            localStorage.setItem('email',
loginResult.user.email);
            window.location.href =
"dashboard.html";
          } else {
            alert(loginResult.error || 'Не вдалося
виконати автоматичний вхід.');
          }
        })
        .catch(err => console.error('Error
(autoLoginAfterRegister):', err));
    }

    // Обробка форми входу
    const loginForm =
document.getElementById('loginForm');
    if (loginForm) {
      loginForm.addEventListener('submit',
function(e) {
        e.preventDefault();
        const username =
document.getElementById('loginUsername').value;
        const password =
document.getElementById('loginPassword').value;
        const data = { username, password };


fetch('http://localhost:5001/api/auth/login', {
            method: 'POST',
            headers: { 'Content-Type':
'application/json' },
            body: JSON.stringify(data)
          })
          .then(response => response.json())
          .then(result => {
            if (result.user) {
              localStorage.setItem('userId',
result.user.id);
              localStorage.setItem('username',
result.user.username);
              localStorage.setItem('email',
result.user.email);
              window.location.href =
"dashboard.html";
            } else {
              alert(result.error);
            }
          })
          .catch(error => console.error('Error:',
error));
        });
    }
  }

  // Блок для dashboard.html (відображення
профілю, керування поїздками та вихід)
  if
(window.location.pathname.endsWith('dashboard.ht
ml')) {
    const userId = localStorage.getItem('userId');
    if (!userId) {
      alert("Ви не авторизовані!");
      window.location.href = "index.html";
    }

    // Обробка кнопки "Вийти"
    const logoutBtn =
document.getElementById('logoutBtn');
```

```
    if (logoutBtn) {
        logoutBtn.addEventListener('click', () => {
            localStorage.clear();
            window.location.href = "index.html";
        });
    }

    // Функція завантаження профілю користувача
    const profileDiv = document.getElementById('profileData');
    function loadUserProfile() {

        fetch(`http://localhost:5001/api/user/${userId}`)
            .then(response => response.json())
            .then(data => {
                if (data.user) {
                    profileDiv.innerHTML = `
                        <p>ID: ${data.user.id}</p>
                        <p>Ім'я: ${data.user.username}</p>
                        <p>Email: ${data.user.email}</p>
                        <p>Адреса: ${data.user.address || '-'}</p>
                        <p>Телефон: ${data.user.phone || '-'}</p>
                        <p>Є авто: ${data.user.has_car ? 'Так' : 'Ні'}</p>
                    `;
                } else {
                    profileDiv.textContent = data.error;
                }
            })
            .catch(error => console.error('Error:', error));
    }
    loadUserProfile();

    // Перехід на сторінку редагування профілю
    const editProfileBtn = document.getElementById('editProfileBtn');
    if (editProfileBtn) {
        editProfileBtn.addEventListener('click', () => {
            window.location.href = 'edit_profile.html';
        });
    }

    // Обробка форми створення поїздки
    const createRideForm = document.getElementById('createRideForm');
    if (createRideForm) {
        createRideForm.addEventListener('submit', function(e) {
            e.preventDefault();
            const route = document.getElementById('rideRoute').value;
            const departure_time = document.getElementById('rideDeparture').value;
            const available_seats = parseInt(document.getElementById('rideSeats').value);

            const rideData = {
                user_id: userId,
                route: route,
                departure_time: departure_time,
                available_seats: available_seats
            };

            fetch('http://localhost:5001/api/rides/', {
                method: 'POST',
                headers: { 'Content-Type': 'application/json' },
                body: JSON.stringify(rideData)
            })
            .then(response => response.json())
            .then(result => {
                alert(result.message || result.error);
                loadRides();  // Оновлюємо список поїздок
            })
            .catch(error => console.error('Error:', error));
        });
    }

    // Функція завантаження списку поїздок із таблицею та кнопками операцій
    function loadRides() {
        fetch('http://localhost:5001/api/rides/')
            .then(response => response.json())
            .then(data => {
```

```
            const ridesListDiv =
document.getElementById('ridesList');
            let tableHtml = `
              <table>
                <thead>
                  <tr>
                    <th>ID</th>
                    <th>Маршрут</th>
                    <th>Дата/час</th>
                    <th>Вільних місць</th>
                    <th>Автор</th>
                    <th>Операції</th>
                  </tr>
                </thead>
                <tbody>
            `;

            if (data.rides && data.rides.length > 0)
{
              data.rides.forEach(ride => {
                const isOwner =
(parseInt(ride.user_id) === parseInt(userId));
                // В колонці "Автор"
відображаємо ім'я автора, якщо доступне,
інакше user_id
                const author = ride.username ?
ride.username : ride.user_id;

                tableHtml += `
                  <tr>
                    <td>${ride.id}</td>
                    <td>${ride.route}</td>

<td>${ride.departure_time}</td>

<td>${ride.available_seats}</td>
                    <td>${author}</td>
                    <td>
                `;

                if (isOwner) {
                  tableHtml += `
                    <button class="editRideBtn"
data-ride-id="${ride.id}">Редагувати</button>
                    <button
class="deleteRideBtn" data-ride-
id="${ride.id}">Видалити</button>
                  `;
                } else {
                  tableHtml += `<button
class="joinRideBtn" data-ride-
id="${ride.id}">Приєднатися</button>`;
                }

                tableHtml += `</td></tr>`;
              });
            } else {
              tableHtml += `<tr><td
colspan="6">Немає створених
поїздок.</td></tr>`;
            }

            tableHtml += `
                </tbody>
              </table>
            `;
            ridesListDiv.innerHTML = tableHtml;

            // Додаємо обробники для кнопок
            attachRideButtonsListeners();
          })
          .catch(error => console.error('Error:',
error));
      }

      // Функція для додавання обробників для
кнопок операцій з поїздками
      function attachRideButtonsListeners() {
        // Обробка кнопок редагування
        const editBtns =
document.querySelectorAll('.editRideBtn');
        editBtns.forEach(btn => {
          btn.addEventListener('click', () => {
            const rideId = btn.getAttribute('data-
ride-id');
            console.log('Редагування поїздки,
ID:', rideId);
            // Тут реалізуйте редіректна
сторінку редагування поїздки (якщо потрібно)
            // Наприклад: window.location.href =
`edit_ride.html?rideId=${rideId}`;
          });
        });

        // Обробка кнопок видалення
```

```
        const deleteBtns =
document.querySelectorAll('.deleteRideBtn');
        deleteBtns.forEach(btn => {
            btn.addEventListener('click', () => {
                const rideId = btn.getAttribute('data-
ride-id');
                if (confirm('Ви впевнені, що хочете
видалити цю поїздку?')) {

fetch(`http://localhost:5001/api/rides/${rideId}`, {
                        method: 'DELETE'
                    })
                    .then(response => response.json())
                    .then(result => {
                        alert(result.message || result.error);
                        loadRides();
                    })
                    .catch(error => console.error('Error:',
error));
                }
            });
        });

        // Обробка кнопок приєднання
        const joinBtns =
document.querySelectorAll('.joinRideBtn');
        joinBtns.forEach(btn => {
            btn.addEventListener('click', () => {
                const rideId = btn.getAttribute('data-
ride-id');

fetch(`http://localhost:5001/api/rides/${rideId}/join
`, {
                        method: 'POST',
                        headers: { 'Content-Type':
'application/json' }
                    })
                    .then(response => response.json())
                    .then(result => {
                        alert(result.message || result.error);
                        loadRides();
                    })
                    .catch(error => console.error('Error:',
error));
            });
        });
    }
        loadRides();
    }

    // Блок для edit_profile.html (редагування
профілю користувача)
    if
(window.location.pathname.endsWith('edit_profile.
html')) {
        const userId = localStorage.getItem('userId');
        if (!userId) {
            alert("Ви не авторизовані!");
            window.location.href = "index.html";
        }

        const editForm =
document.getElementById('editProfileForm');
        const usernameInput =
document.getElementById('editUsername');
        const emailInput =
document.getElementById('editEmail');
        const addressInput =
document.getElementById('editAddress');
        const phoneInput =
document.getElementById('editPhone');
        const hasCarInput =
document.getElementById('editHasCar');
        const passwordInput =
document.getElementById('editPassword');


fetch(`http://localhost:5001/api/user/${userId}`)
            .then(res => res.json())
            .then(data => {
                if (data.user) {
                    usernameInput.value =
data.user.username || '';
                    emailInput.value = data.user.email || '';
                    addressInput.value = data.user.address
|| '';
                    phoneInput.value = data.user.phone || '';
                    hasCarInput.checked =
data.user.has_car ? true : false;
                } else {
                    alert(data.error || 'Помилка
завантаження даних профілю');
                }
            })
```

```
        .catch(err => console.error('Error loading
user data:', err));

    editForm.addEventListener('submit', (e) => {
        e.preventDefault();
        const bodyData = {
            username: usernameInput.value,
            email: emailInput.value,
            address: addressInput.value,
            phone: phoneInput.value,
            has_car: hasCarInput.checked
        };
        if (passwordInput.value.trim().length > 0) {
            bodyData.password =
passwordInput.value;
        }

fetch(`http://localhost:5001/api/user/${userId}`, {
            method: 'PUT',
            headers: { 'Content-Type':
'application/json' },
            body: JSON.stringify(bodyData)
        })
        .then(res => res.json())
        .then(result => {
            if (result.message) {
                alert(result.message);
                window.location.href =
'dashboard.html';
            } else {
                alert(result.error || 'Помилка
оновлення даних профілю');
            }
        })
        .catch(err => console.error('Error updating
profile:', err));
    });
  }
});
```