**Міністерство освіти і науки України**
**Тернопільський національний технічний університет імені Івана Пулюя**

Комп'ютерно інформаційних систем і програмної інженерії
(повна назва факультету )
Програмної інженерії
(повна назва кафедри)

# КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

магістр
(назва освітнього ступеня)

на тему: Проєктування та розробка вебзастосунку для оренди велосипедів з використанням .NET та Angular

Виконав(ла): студент(ка) VI курсу, групи СПм-61
спеціальності 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

| | | |
|---|---|---|
| | | Барабаш В.О. |
| | (підпис) | (прізвище та ініціали) |
| Керівник | | Петрик М. Р. |
| | (підпис) | (прізвище та ініціали) |
| Нормоконтроль | | Стоянов Ю. М. |
| | (підпис) | (прізвище та ініціали) |
| Завідувач кафедри | | Петрик М. Р. |
| | (підпис) | (прізвище та ініціали) |
| Рецензент | | |
| | (підпис) | (прізвище та ініціали) |

Тернопіль
2025

Міністерство освіти і науки України
**Тернопільський національний технічний університет імені Івана Пулюя**

Факультет Комп'ютерно інформаційних систем і програмної інженерії
(повна назва факультету)

Кафедра Програмної інженерії
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Петрик М. Р
(підпис) (прізвище та ініціали)
«__» _____ 2025 р.

# З А В Д А Н Н Я
## НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня магістра
(назва освітнього ступеня)

за спеціальністю 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

студенту Барабашу Віталію Олеговичу
(прізвище, ім'я, по батькові)

1. Тема роботи Проєктування та розробка вебзастосунку для оренди велосипедів з використанням .NET та Angular

Керівник роботи доктор фіз.-мат. наук професор кафедри ПІ Петрик Михайло Романович
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від «___» _____ 2024 року № _____

2. Термін подання студентом завершеної роботи _____

3. Вихідні дані до роботи Предметна область, технічне завдання, вимоги та специфікація, програмне рішення

4. Зміст роботи (перелік питань, які потрібно розробити):

Вступ, Розділ 1. Аналіз вимог до програмної системи, Розділ 2. Проєктування та розробка програмної системи, Розділ 3. Тестування програмної системи, Розділ 4. Охорона праці та безпека в надзвичайних ситуаціях, Висновки, Список використаних джерел, Додатки

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів):

Ілюстративні зображення, інформативні зображення для доповнення тексту, діаграми, знімки екрану з проробленою роботою.

6. Консультанти розділів роботи

| Розділ | Прізвище, ініціали та посада консультанта | Підпис, дата | |
|---|---|---|---|
| | | завдання видав | завдання прийняв |
| Охорона праці | Осухівська Г. М., зав. каф. КС | | |
| Безпека в надзвичайних ситуаціях | Стручок В. С., ст. викл. каф. ОФ | | |

7. Дата видачі завдання 14.10.2025

# КАЛЕНДАРНИЙ ПЛАН

| № з/п | Назва етапів роботи | Термін виконання етапів роботи | Примітка |
|---|---|---|---|
| 1. | Отримання завдання | 14.10.2025 | *Виконано* |
| 2 | Підбір джерел по теми кваліфікаційної роботи | 15.10.2025-25.10.2025 | ***Виконано*** |
| 3 | Аналіз та підготовка технічної специфікації | 30.10.2025 | ***Виконано*** |
| 4 | Розробка програмної системи | 01.11.2025-19.11.2025 | ***Виконано*** |
| 5 | Написання звіту по проробленій роботі | 20.11.2025-12.12.2025 | ***Виконано*** |
| 6 | Оформлення кваліфікаційної роботи | 13.12.2025-14.12.2025 | ***Виконано*** |
| 7 | Виконання завдання «Охорона праці та безпека в | 15.12.2025- | ***Виконано*** |
| | надзвичайних ситуаціях» | 19.12.2025 | |
| 8 | Нормоконтроль | 15.12.2025-19.20.2025 | ***Виконано*** |
| 9 | Перевірка на плагіат | 15.12.2025-19.12.2025 | ***Виконано*** |
| 10 | Попередній захист кваліфікаційної роботи | 15.12.2025 | ***Виконано*** |
| 11 | Захист кваліфікаційної роботи | 22.12.2025 | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

Студент _____ Барабаш В. О.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Петрик М. Р.
(підпис) (прізвище та ініціали)

# АНОТАЦІЯ

Проєктування та розробка вебзастосунку для оренди велосипедів з використанням .NET та Angular // Кваліфікаційна робота освітнього рівня «Магістр» // Барабаша Віталія Олеговича // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп'ютерно-інформаційних систем і програмної інженерії, кафедра програмної інженерії, група СПм-61 // Тернопіль, 2025 // С. 85, рис. – 40, табл. – 0, додат. – 2, кресл. – 0, бібліогр. – 32.

Ключові слова: веб-застосунок, оренда велосипедів, .NET, Angular, REST API, front-end, back-end, база даних, тестування.

Мета роботи полягає у створенні веб-застосунку для ефективного управління процесом оренди велосипедів із сучасним користувацьким інтерфейсом і надійною серверною частиною.

Об'єкт дослідження – веб-системи для організації оренди транспортних засобів з інтегрованою базою даних та функціями бронювання.

Предмет дослідження – методики та технології розробки веб-застосунків із використанням .NET для back-end та Angular для front-end, включаючи взаємодію через REST API, управління даними та інтерактивне відображення розташування велосипедів.

У роботі розглянуто процес проектування, створення, тестування та розгортання веб-застосунку. Серверна частина реалізована на .NET з підключенням до реляційної бази даних, а клієнтська – на Angular з інтерактивними компонентами та картами для відображення доступних велосипедів. Проведено аналіз існуючих рішень на ринку, виділено їхні сильні та слабкі сторони для формування оптимальної архітектури. Тестування системи виконано для перевірки продуктивності, коректності роботи функцій бронювання та стабільності роботи сервісу.

# ABSTRACT

Design and development of a web application for bicycle rental using .NET and Angular // Master's degree thesis // Barabash Vitalii Olegovych // Ivan Pul'uj Ternopil National Technical University, Faculty of Computer and Information Systems and Software Engineering, Department of Software Engineering, Group SPm-61 // Ternopil, 2025 // C. 85, fig. – 40, tab. – 0, app. – 2, drawing. – 0, bibliogr. – 32.

Keywords: web application, bicycle rental, .NET, Angular, REST API, front-end, back-end, database, testing.

The purpose of this work is to create a web application for effective management of the bicycle rental process with a modern user interface and a reliable server part. The object of research is web systems for organizing vehicle rental with an integrated database and booking functions.

The subject of the study is methods and technologies for developing web applications using .NET for the back-end and Angular for the front-end, including interaction via REST API, data management, and interactive display of bicycle locations.

The paper considers the process of designing, creating, testing, and deploying a web application. The server part is implemented on .NET with a connection to a relational database, and the client part is implemented on Angular with interactive components and maps to display available bicycles. An analysis of existing solutions on the market was conducted, and their strengths and weaknesses were identified to form an optimal architecture. The system was tested to verify its performance, the correctness of the booking functions, and the stability of the service.

# ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

Букінг (англ. Booking) – процес бронювання велосипеда користувачем

Шеринг (англ. Sharing) – отримання чогось у право власності на конкретний період

Front-end – клієнтська частина веб-застосунку, що відображає інтерфейс та взаємодіє з користувачем

Back-end – серверна частина веб-застосунку, яка обробляє бізнес-логіку та дані

SPA (англ. Single Page Application) – веб-застосунок, що працює як єдина інтерактивна сторінка без перезавантаження

UI (англ. User Interface) – елементи інтерфейсу для взаємодії користувача із системою

UX (англ. User Experience) – зручність та комфорт користування веб-застосунком

API (англ. Application Programming Interface) – набір методів для обміну даними між клієнтом і сервером

REST (англ. Representational State Transfer) – архітектурний стиль організації веб-сервісів

HTTP (англ. Hypertext Transfer Protocol) – протокол передачі даних у мережі

HTTPS (англ. Hypertext Transfer Protocol Secure) – захищений протокол передачі даних

JSON (англ. JavaScript Object Notation) – формат представлення структурованих даних

Authentication – процес перевірки особи користувача для надання доступу

Authorization – визначення прав користувача на виконання дій у системі

JWT (англ. JSON Web Token) – стандарт захищеної передачі даних про користувача

БД – організована колекція даних, що зберігається у системі

РБД – база даних з таблицями та зв’язками між ними

СУБД – програмне забезпечення для управління базами даних

CRUD (англ. Create, Read, Update, Delete) – основні операції над даними

Session – тимчасовий об’єкт для збереження стану користувача

Entity Framework – бібліотека .NET для роботи з реляційними базами даних

Dependency Injection – механізм передачі залежностей між класами та компонентами

Component – елемент Angular-застосунку, який відповідає за частину інтерфейсу

Module – структурна одиниця Angular-застосунку для організації коду та логіки

Service – клас Angular, що надає функціональність та обмін даними між компонентами

**ЗМІСТ**

# ВСТУП

Сучасний цифровий світ докорінно змінив наше уявлення про комфорт у місті та взаємодію із звичною для кожного інфраструктурою. Уже нікого не здивуєш наявністю мобільного, або веб-застосунку для перегляду наявності взаємодії із громадськими послугами. Так і способи пересування по місту проходять свої цифрові та фізичні метаморфози, де персональний та громадський транспорт відходять на задній план. Оренда, чи б то пак шеринг, засобів пересування – швидко, екологічно та комфортно.

Важливо також розуміти, сучасний користувач – це неймовірно вибагливий клієнт, який живе в швидкоплинному та рухливому світі, що згорне додаток після 3-4 секунд очікування та піде далі, закриє його якщо дизайн веб-сайту буде занадто старомодним. Тобто, точки прикладання найбільшої уваги та зусиль у цьому та й у інщих не пов’язаний з шерингом застосунках, – швидкість, зручність, інтуїтивність та стиль. Отож, при розробці потрібно націлитись на першочергове вирішення цих аспектів, адже відповідно саме вони є вирішальними для успіху описаного в пояснювальній записці проєкту веб-застосунку для оренди велосипедів.

Мета цієї магістерської роботи та її основна ціль – дослідження спроможностей проєктування та розробки невеличкого веб-застосунок для оренди велосипедів засобами технологій .NET та Angular. Також, важливо щоб веб-застосунок забезпечував надійну, безпечну та швидку взаємодію цього стеку, що безперечно також є основною ціллю цієї роботи. Щодо питання цінності такої роботи, то пояснювальна записка як і програмний продукт, що планується до розробки, мають на меті детально розглянути усі наявні засоби та інструментарій у обох технологіях стеку та оцінити спроможності та доцільність застосування їх у розробці.

Відповідно до встановленої мети і цілі, основна задача полягає в проєктуванні та створенні надійного, безпечного, швидкого та комфортного

сервісу, яким будуть користуватись і будуть поширювати його серед знайомих та друзів через його переваги та безпрецедентні підходи. Тому фронт основних робіт, щодо реалізації над якими варто задуматись першочергово виглядатиме таким чином:

· ефективний та компактний RESTful API;

· сучасний, інтуїтивний та комфортний інтерфейсу веб-застосунку;

· імплементацію принципу “нажав-отримав”:

· наявність надійного захисту;

· швидкість та загальна зручність використання сервісу.

Також, варто звернути увагу більш детально щодо користувачів. А саме, система відповідно повинна бути орієнтована на дві групи користувачів. Звичайний пересічний користувач матиме змогу переглядати потрібну йому інформацію відповідно по усіх та по конкретному велосипеду, відповідно, та надалі оформляти оренду і користуватись велосипедом. Менеджери, працівники сервісу та адміністратори отримують весь необхідний, для роботи, інструментарій зі управління запасами, моніторингу стану обладнання та оновлення даних. Такий поділ функцій робить сервіс зручним і ефективним для всіх учасників.

Технічно завдання включає в себе імплементацію RESTful API та забезпечення комфортної оку клієнтської частини засобами Angular та найсучаснішими підходами в проєктуванні користувацьких інтерфейсів. І створення надійної, ефективної та здатної обробити потоки даних від різних користувачів серверної частини. А також, поєднання їх у одному середовищі для виконання усіх поставлених задач в ефективній та надійній спарці.

Фреймворк .NET та її основу, у вигляді мови програмування C#, було обрано для серверної частини через високу продуктивність, сучасність і сумісність з хмарними та контейнеризованими середовищами, що зможе забезпечити майбутнє розгортання в продакшені. Інструменти фреймворку, як-от ін’єкція залежностей, асинхронне програмування та Entity Framework Core, спрощують реалізацію масштабованої серверної логіки. Мінімальні API,

контролери та OpenAPI забезпечують зрозумілу документацію та зручну роботу з API. Кросплатформеність дозволяє розгортати додаток на Windows, Linux або в контейнерах, забезпечуючи гнучкість розвитку.

Angular обрано для фронтенду через типобезпечність, модульну структуру та компонентно-орієнтовану архітектуру. Це спрощує організацію інтерфейсу, повторне використання компонентів і підтримку узгодженого дизайну. Angular забезпечує плавну і логічну взаємодію користувача з додатком, створюючи комфортний досвід користувача, а компонентно-орієнтований підхід полегшує масштабування та майбутню підтримку системи.

# РОЗДІЛ 1. АНАЛІЗ ВИМОГ ДО ПРОГРАМНОЇ СИСТЕМИ

Розділ зосереджується на аналізі вимог до веб-додатку для прокату велосипедів. У ньому розглядаються сучасні тенденції у веб-системах прокату, визначаються потреби цільових користувачів та функціональні й нефункціональні вимоги. Крім того, у розділі обґрунтовується вибір технологій і порівнюються подібні рішення, щоб забезпечити міцну основу для проектування системи.

## 1.1 Огляд сучасних систем та тенденцій в сфері оренди велосипедів

Міський транспорт переживає глибокі трансформації, а послуги шерингу, використання орендованих на певний період часу транспортних засобів, стають важливою складовою сучасних міст. Системи прокату автомобілів, електросамокатів, велосипедів та електробайків, які часто називають платформами спільного використання транспорту на прокат. В той ж час, все більше набуває популярності прокат двоколісних засобів пересування без двигуна внутрішнього згорання, галузь що швидко розвиваються завдяки своїй зручності, екологічності та здатності зменшувати затори на дорогах. Сучасні рішення з прокату велосипедів поєднують фізичну інфраструктуру, таку як док-станції або велосипеди без док-станцій, з цифровими платформами, що дозволяють користувачам знаходити, бронювати та оплачувати велосипеди через веб- та мобільні додатки. Інтеграція GPS-відстеження в режимі реального часу, автоматизованих систем оплати та аналітики використання дозволила містам і приватним компаніям ефективно управляти великими парками велосипедів.

Еволюція веб-технологій стала рушійним фактором успіху усіх сучасних систем, які базуються на роботу з користувачем та його залученістю до використання інтерфейсу, що відіграло особливу роль в становленні бізнесу розбудованому на спільному використання засобів пересування. Односторінкові

додатки (SPA) та прогресивні веб-додатки (PWA) забезпечують швидкий та приємний користувацький досвід, що нагадує нативні мобільні додатки, залишаючись доступними через стандартні веб-браузери [7, 8]. А такі фреймворки, як Angular, полегшують створення динамічних інтерфейсів, забезпечуючи такі функції, як інтерактивні карти, оновлення доступності велосипедів у режимі реального часу та миттєве підтвердження бронювання. Відокремлюючи інтерфейс від бізнес-логіки, ці фреймворки забезпечують модульність і зручність обслуговування, що є надзвичайно важливим для платформ, які мають швидко розширюватися як за кількістю користувачів, так і за географічним покриттям.

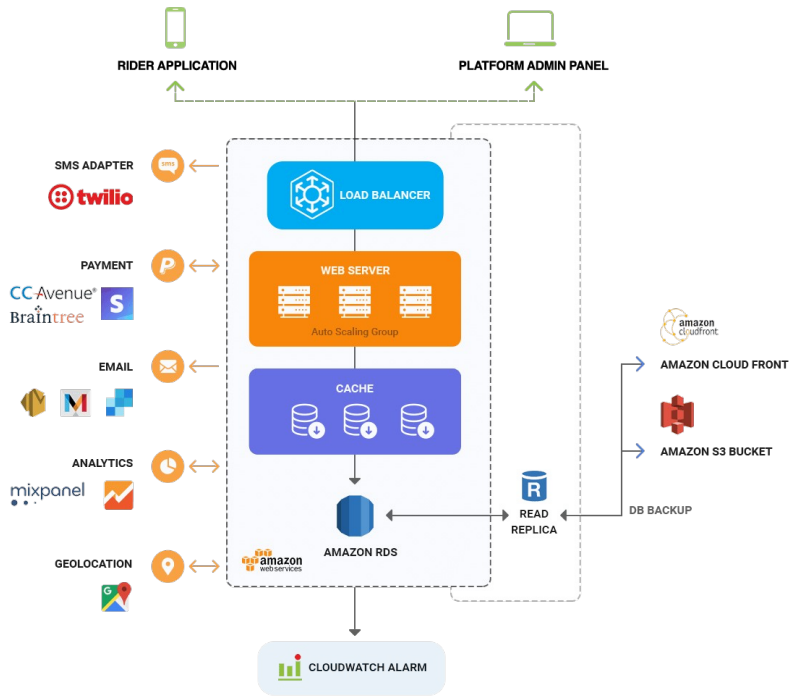



Рисунок 1.1 – Архітектура типового веб-додатку для спільного використання велосипедів від Mobisoft Infotech [18]

Окрім технологічних досягнень, змінилися й очікування користувачів. Сучасні користувачі вимагають інтуїтивно зрозумілих інтерфейсів, швидкого завантаження та безперебійної інтеграції з цифровими платіжними системами. Вони також очікують на зворотний зв'язок у режимі реального часу щодо

доступності велосипедів, їхнього місцезнаходження та статусу оренди. Щоб задовольнити ці очікування, системи оренди велосипедів покладаються на API, з'єднання WebSocket та хмарні сервіси, які обробляють великі обсяги одночасних запитів без шкоди для продуктивності. Таке поєднання інновацій у фронтенді та бекенді дозволяє операторам надавати надійні та зручні для користувачів послуги, що покращують загальний досвід міської мобільності.

Іншою важливою тенденцією є використання даних для оптимізації операцій. Системи прокату зараз збирають і аналізують величезні обсяги даних, включаючи моделі використання, години пік, популярні маршрути та потреби в технічному обслуговуванні [20, 21, 22]. Ця інформація дозволяє операторам динамічно змінювати розташування велосипедів, ефективно планувати графіки технічного обслуговування та покращувати загальний досвід користувачів. Аналітика даних також підтримує бізнес-рішення, пов'язані з ціноутворенням, розширенням парку та інтеграцією з іншими послугами міської мобільності, створюючи більш адаптивну та інтелектуальну транспортну мережу.

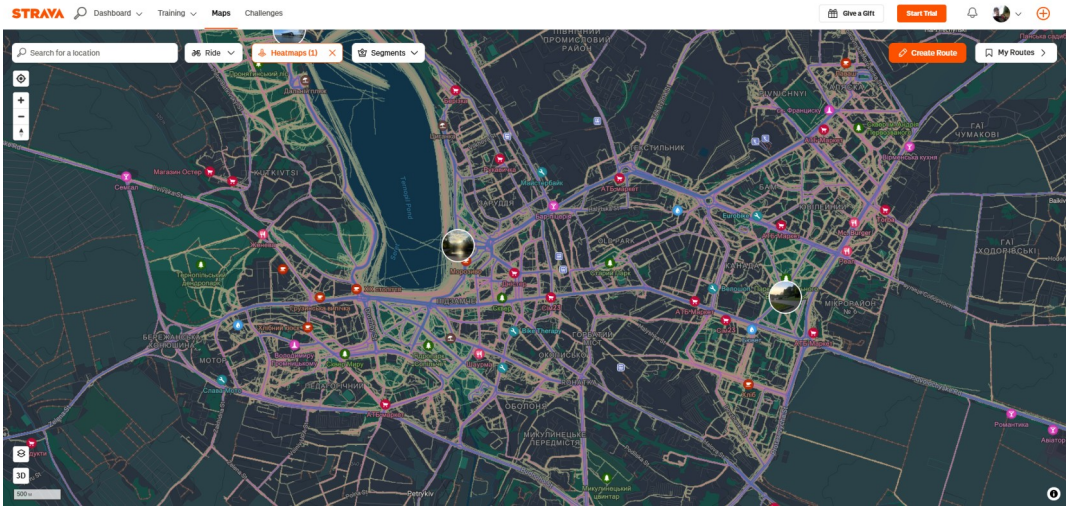



Рисунок 1.2 – Приклад теплової карти використання велосипедів у місті Тернопіль сервісу STRAVA [19]

Нарешті, сучасні системи прокату велосипедів повинні вирішувати проблеми, пов'язані з масштабованістю, безпекою та стійкістю. У міру зростання

міського населення та збільшення попиту на спільну мобільність платформи повинні обробляти мільйони користувачів та одночасних прокатів без простоїв. Необхідними є заходи безпеки, включаючи аутентифікацію користувачів, шифрування обробки платежів та захист від несанкціонованого доступу. Міркування щодо стійкості, такі як електровелосипеди на акумуляторних батареях та технічне обслуговування велосипедів без док-станцій, все частіше інтегруються в планування системи. Поєднання технологічних інновацій з операційною стратегією гарантує, що сучасні платформи прокату велосипедів залишаються ефективними, результативними та відповідають потребам сучасних міст.

### 1.2 Аналіз цільової аудиторії та потреб користувачів

Успіх системи прокату велосипедів значною мірою залежить від розуміння її цільової аудиторії. Міські пасажири є однією з основних груп користувачів, які покладаються на велосипеди для коротких поїздок між домом, роботою та транспортними вузлами. Ці користувачі надають пріоритет швидкості, зручності та надійності, очікуючи, що додаток надаватиме точну інформацію про наявність велосипедів, найближчі станції та приблизний час поїздки. Оновлення в режимі реального часу та можливість заздалегідь забронювати велосипед є особливо важливими для щоденних пасажирів, які ретельно планують свої маршрути.

Туристи та випадкові користувачі велосипедів становлять інший значний сегмент користувачів. На відміну від регулярних пасажирів, їхня основна увага зосереджена на зручності доступу, інтуїтивній навігації та мінімальному часі налаштування. Їм потрібні прості процеси реєстрації, чіткі вказівки щодо використання велосипедів та інтегровані варіанти оплати, які не вимагають довгострокових зобов'язань. Для цієї групи додаткові функції, такі як пропозиції маршрутів, найближчі визначні пам'ятки та багатомовна підтримка, покращують

загальний досвід користувачів, роблячи платформу більш привабливою та зручною.

З операційної точки зору, системні адміністратори становлять другорядну, але важливу групу користувачів. Їхні потреби відрізняються від потреб кінцевих користувачів і зосереджені на ефективному управлінні парком та платформою. Основні вимоги включають моніторинг місцезнаходження велосипедів, ведення точних журналів оренди, обробку графіків технічного обслуговування та аналіз моделей використання. Адміністративні інструменти повинні бути ефективними, безпечними та здатними генерувати звіти для підтримки процесів прийняття рішень, пов'язаних з оптимізацією парку та поліпшенням обслуговування.

Розуміння потреб користувачів також передбачає врахування нефункціональних очікувань. Користувачі очікують швидкої роботи додатків, мінімального часу простою та безпечної обробки особистих і фінансових даних. Мобільна адаптивність має вирішальне значення, оскільки більшість користувачів отримують доступ до платформи через смартфони під час пересування. Крім того, функції доступності, такі як чіткі візуальні підказки, інтуїтивна навігація та підтримка користувачів з інвалідністю, забезпечують інклюзивність та розширюють потенційну базу користувачів.

Нарешті, відображення вимог користувачів у функціональних можливостях системи є важливим етапом у процесі проектування. Доступність велосипедів, пошук на основі геолокації, управління бронюваннями, інтеграція платежів, та інші функції – все це випливає з визначених потреб користувачів. Завдяки аналізу цільової аудиторії платформа може надавати індивідуальні рішення, що забезпечують зручність та ефективність.

### 1.3 Функціональні та нефункціональні вимоги до системи

Важливим етапом у розробці веб-додатку для прокату велосипедів є визначення його функціональних та нефункціональних вимог. Функціональні

вимоги описують, що система повинна робити, щоб задовольнити потреби користувачів та адміністраторів, тоді як нефункціональні вимоги визначають, як система повинна поводитися, щоб забезпечити надійність, безпеку та зручність використання [9, 12]. Чітке визначення цих вимог гарантує, що система зможе забезпечити високу якість користувацького досвіду, залишаючись при цьому операційно ефективною та зручною в обслуговуванні. У наступних розділах наведено структурований огляд вимог із поясненнями та прикладами.

Функціональні вимоги:

Система повинна забезпечувати такі основні функції:

1. Реєстрація та автентифікація користувачів

· Безпечне створення облікового запису та вхід за допомогою електронної пошти, номера телефону або інтеграції з соціальними мережами [16, 17];

· Управління паролями, включаючи функцію скидання та багатофакторну автентифікацію;

· Персоналізоване управління профілем, включаючи налаштування користувача та історію прокату;

· Забезпечує кожному користувачеві безпечний та індивідуальний доступ до платформи.

2. Пошук та доступність велосипедів

· Відстеження місцезнаходження велосипедів у режимі реального часу за допомогою GPS та картографічних сервісів;

· Фільтрування за типом велосипеда (електричний, стандартний), рівнем заряду акумулятора або відстанню;

· Відображення статусу доступності, приблизного часу поїздки та найближчих станцій док-станцій;

· Дозволяє користувачам швидко знаходити та отримувати доступ до велосипедів, мінімізуючи час очікування та підвищуючи зручність.

3. Управління бронюванням та орендою

· Бронювання велосипедів на певний час та початок сеансів оренди;

· Завершення оренди та реєстрація тривалості, пройденої відстані та історії використання;

· Створення цифрових квитанцій та зведених звітів про завершені оренди;

· Підтримка прозорого відстеження використання та забезпечення підзвітності як для користувачів, так і для адміністраторів.

4. Адміністративні функції

· Управління парком велосипедів, включаючи додавання нових велосипедів, видалення старих або пошкоджених велосипедів та відстеження їхнього місцезнаходження;

· Моніторинг графіків технічного обслуговування та робочого стану всіх велосипедів;

· Створюйте детальні звіти про моделі використання, години пікового навантаження та тенденції доходів;

· Надання адміністраторам інструментів для оптимізації операцій, скорочення простоїв та поліпшення якості обслуговування.

Нефункціональні вимоги:

Нефункціональні вимоги визначають, наскільки добре працює система, та забезпечують її надійність, безпеку та зручність у використанні:

1. Продуктивність та масштабованість

· Система повинна підтримувати сотні або тисячі одночасних користувачів без помітних затримок.

· Хмарна архітектура дозволяє горизонтальне масштабування для адаптації до зростання розміру парку та бази користувачів.

· Час відгуку для пошуку, бронювання або оновлення статусу велосипеда повинен залишатися менше декількох секунд.

2. Надійність та доступність

· Забезпечити доступність сервісу 24/7, навіть під час технічного обслуговування або пікових періодів використання

· Автоматичні механізми відновлення після збоїв бекенду для запобігання втраті даних або перериванню роботи сервісу.

· Постійний моніторинг стану системи з попередженнями про потенційні перебої.

3. Безпека

· Шифрування конфіденційних даних користувачів, включаючи облікові дані та інформацію про місцезнаходження [16, 17].

· Впровадження безпечної аутентифікації, авторизації та управління сеансами.

· Захист від несанкціонованого доступу, витоку даних та кібератак за допомогою брандмауерів, протоколів SSL/TLS та регулярних аудитів безпеки.

4. Зручність використання та доступність

· Інтуїтивно зрозумілий інтерфейс для веб- та мобільних пристроїв, з послідовною навігацією та чіткими візуальними підказками.

· Адаптивний дизайн для різних розмірів екранів та орієнтацій.

· Функції доступності для користувачів з інвалідністю, включаючи сумісність із програмами для читання з екрану, режими високої контрастності та навігацію за допомогою клавіатури.

· Спрощує взаємодію користувачів, зменшує кількість помилок та забезпечує інклюзивність.

5. Можливість обслуговування

· Модульна архітектура для ефективних оновлень, вдосконалення функцій та виправлення помилок.

· Добре задокументована база коду та архітектура системи для полегшення адаптації нових розробників та адміністраторів.

· Дозволяє довгострокову еволюцію системи з мінімальним порушенням існуючих послуг.

Визначення функціональних та нефункціональних вимог забезпечує чіткий план дій для проектування, розробки та тестування системи. Функціональні вимоги гарантують, що платформа відповідає практичним потребам користувачів та адміністраторів, а нефункціональні вимоги гарантують продуктивність, безпеку та зручність використання. Разом ці вимоги формують основу для надійного, масштабованого та зручного веб-додатку для прокату велосипедів, створюючи основу для подальших етапів проектування та впровадження.

### 1.4 Вибір технологій та обґрунтування використання .NET та Angular

Розробка високопродуктивного та масштабованого веб-додатку для прокату велосипедів вимагає вибору надійних, зручних в обслуговуванні та технологічно зрілих інструментів. У цьому проекті основними обраними технологіями є .NET для бекенду та Angular для фронтенду. Обидві платформи широко використовуються в корпоративних середовищах і забезпечують структурні, архітектурні та продуктивні характеристики, необхідні для побудови розподіленої системи з взаємодією в режимі реального часу, безпечним потоком даних і тривалим циклом експлуатації. Поєднання цих технологій дозволяє створити додаток, який є продуктивним під навантаженням, простим в обслуговуванні та пристосованим до подальшого розширення.


Рисунок 1.3 – Логотип .NET

Рисунок 1.4 – Логотип Angular

Використання .NET на стороні сервера надає кілька переваг, необхідних для обробки автентифікації користувачів, логіки оренди, платіжних операцій, управління парком та інтеграції з послугами картографування та відстеження [3]. Також, висока продуктивність ASP.NET Core, що забезпечується оптимізованим конвеєром запитів, асинхронною моделлю програмування та сервером Kestrel [6]. Ці функції дозволяють бекенду ефективно обробляти велику кількість одночасних запитів, що є критично важливим, при взаємодії користувачів з додатком. Платформа також побудована з урахуванням модульності, пропонуючи введення залежностей, розширюваність проміжного програмного забезпечення та чітке розділення шарів бізнес-логіки, що підтримує добре структуровану архітектуру.

Екосистема .NET надає широкий спектр готових до виробництва інструментів, які безпосередньо підвищують надійність системи [4, 5]. До них належать Entity Framework Core для взаємодії з реляційними базами даних, вбудовані механізми ідентифікації для безпечної автентифікації, стандартизоване управління конфігурацією та потужна підтримка хмарних середовищ. Бекенд також користується перевагами функцій безпеки корпоративного рівня, таких як шифрування, автентифікація на основі токенів (JWT), політики авторизації та захист від поширених векторів атак [17]. Оскільки дані користувачів містять особисту інформацію та дані про місцезнаходження, обов'язковою є наявність надійних механізмів безпеки.

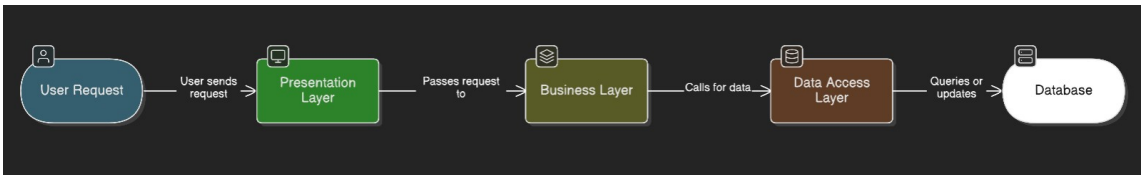



Рисунок 1.5 – Архітектура бекенду з використанням .NET

Фронтенд системи реалізовано за допомогою Angular, фреймворку, придатного для створення інтерактивних, компонентно-орієнтованих користувацьких інтерфейсів. Angular базується на TypeScript, що підвищує безпеку типів, в свою чергу підвищуючи безпеку і надійність додатку [9]. Це дуже корисно для системи, де дані повинні точно відображатися в інтерфейсі користувача, особливо для оновлення статусу велосипедів у реальному часі, розрахунку маршрутів та відображення поточної доступності. Компонентна модель Angular дозволяє розділити інтерфейс на повторно використовувані одиниці, такі як компонент карти, панель профілю користувача і таке інше [9, 12]. Ця модульна структура спрощує масштабування та полегшує підтримку.

Angular також пропонує потужну підтримку реактивного програмування через RxJS, що є необхідним для ефективної обробки асинхронних потоків даних [9]. У платформі прокату велосипедів багато елементів інтерфейсу покладаються на постійно змінювані дані – координати велосипедів, статус прокату та зайнятість станцій повинні оновлюватися без перезавантаження сторінки. Реактивні потоки допомагають досягти плавного оновлення інтерфейсу користувача в режимі реального часу. Фреймворк також надає безліч корисних out-of-box можливостей, як от: маршрутизація, перевірку форм та багато іншого, що забезпечує чисті та структуровані робочі процеси розробки [9]. Ці можливості мінімізують складність фронтенду, забезпечуючи стабільну інтеграцію з серверними службами.

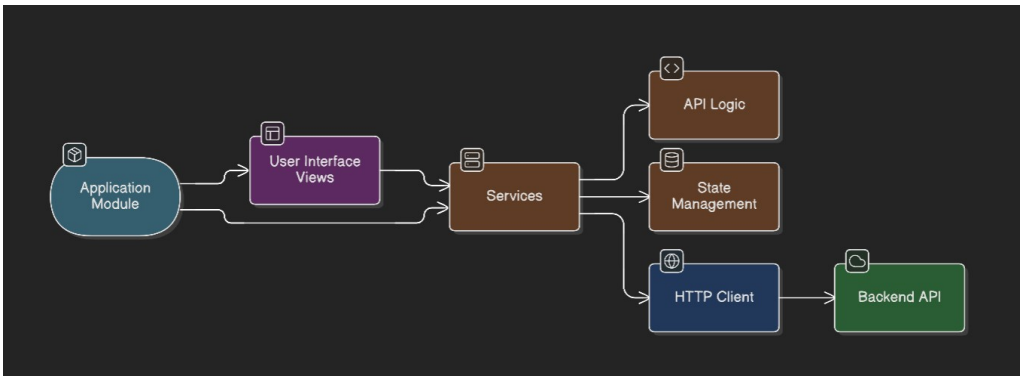



Рисунок 1.6 – Архітектура фронтенду з використанням Angular

Поєднання .NET та Angular створює середовище, яке є технічно узгодженим, простим в інтеграції та відповідає сучасним практикам розробки. Комунікація здійснюється через REST API на основі JSON, при цьому бекенд надає кінцеві точки, відповідальні за аутентифікацію, операції з велосипедами, події життєвого циклу оренди та адміністративні функції [14, 15]. Angular використовує ці кінцеві точки для заповнення інтерфейсу користувача та управління взаємодіями користувачів. Аутентифікація здійснюється за допомогою токенів JWT, що видаються бекендом і безпечно обробляються клієнтом. Вся система може бути розгорнута як окремі служби або як єдине розгортання, де Angular обслуговується додатком .NET.

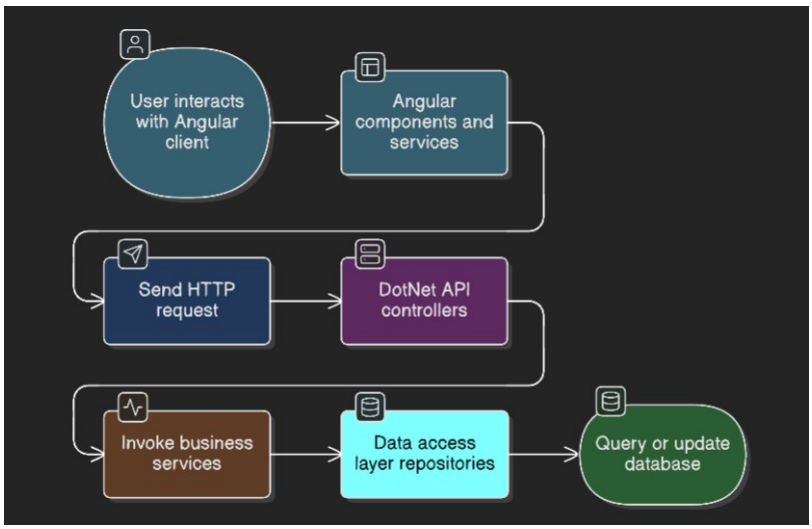



Рисунок 1.7 – Комбінована архітектура системи .NET та Angular

Вибір .NET та Angular забезпечує міцну технологічну основу для платформи прокату велосипедів. Бекенд забезпечує швидку обробку запитів, безпечну обробку даних та масштабоване розподілення послуг, а фронтенд забезпечує чуйний, структурований та зручний у обслуговуванні користувальницький інтерфейс, здатний підтримувати взаємодію в режимі реального часу. Обидві технології є добре відомими, мають широку підтримку спільноти та корпорацій і забезпечують довгострокову стабільність, що робить їх відповідним вибором для системи, яка має розвиватися, розширюватися та надійно працювати у виробничому середовищі.

### 1.5 Огляд подібних рішень та порівняльний аналіз

Детальний огляд існуючих платформ прокату велосипедів дає уявлення про загальні функціональні можливості системи, дизайн інтерфейсу та операційні підходи. Популярні платформи, такі як Donkey Republic, Citi Bike та Spoked [20, 21, 22], пропонують відстеження велосипедів у режимі реального часу, управління бронюваннями та зручні для мобільних пристроїв інтерфейси. Аналіз цих рішень висвітлює їхні сильні сторони та обмеження, надаючи інформацію про найкращі практики для створення надійної та зручної для користувачів системи прокату.

1. Citi Bike [22]

· Переваги: надійний бек-енд, що підтримує великі парки, інтегрований з міськими мобільними сервісами, комплексна звітність для операторів.

· Недоліки: зосереджена переважно на велосипедах із док-станціями, що зменшує гнучкість оренди без док-станцій; інтерфейс користувача може бути менш інтуїтивним для випадкових користувачів.

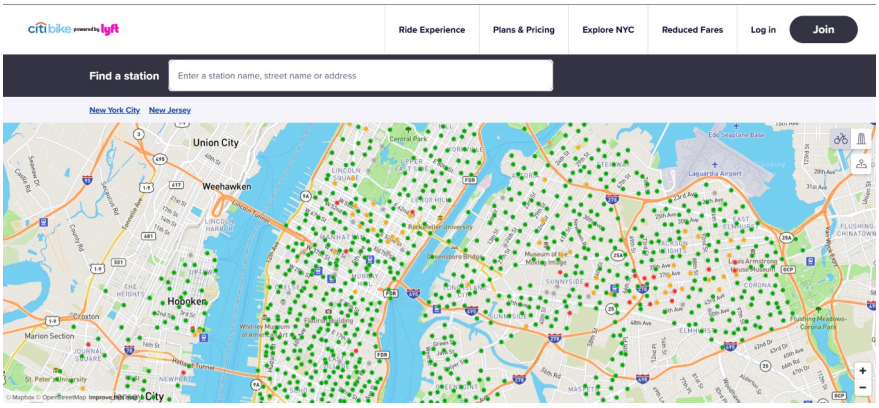



Рисунок 1.8 – Карта станцій Citi Bike та розподіл парку в місті [22]

2. Donkey Republic [20]

· Переваги: чіткий та модульний інтерфейс користувача, підтримка веб- та мобільного доступу, гнучкі варіанти прокату.

· Недоліки: відсутність на теренах України; деякі розширені функції вимагають додаткових передплат.

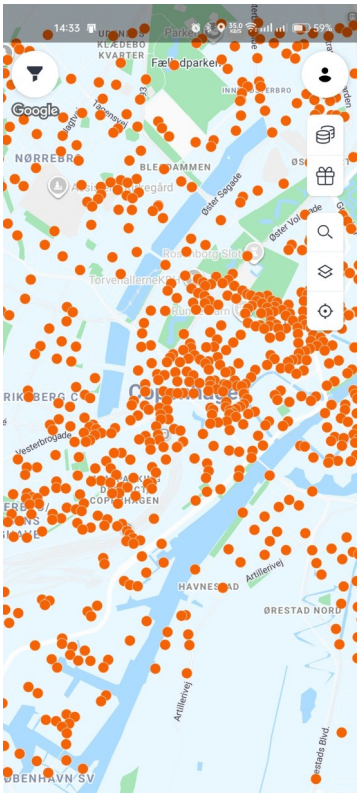



Рисунок 1.9 – Інтерфейс Donkey Republic [20]

3. Spoked [21]

· Переваги: легкий, простий інтерфейс, що дозволяють швидко зареєструватися та швидко орендувати велосипед.

· Недоліки: оренда лише у користувачів, що може призвести до відсутності пропозиції в певній перспективі

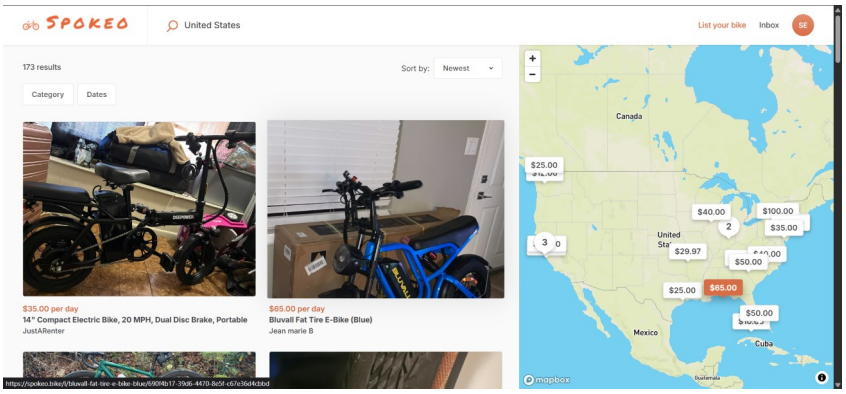



Рисунок 1.10 – Інтерфейс Spoked із картою та списком велосипедів [21]

Порівняльний аналіз демонструє кілька тенденцій у сучасних платформах прокату велосипедів: оновлення геопозиції та доступності в режимі реального часу є необхідними, інтерфейси розроблені так, щоб бути привабливими та зручними для користувачів, а хмарні серверні системи забезпечують масштабованість та надійність. Відмінності в основному проявляються в адміністративних функціях, аналітичних можливостях та підтримці велосипедів з док-станціями та без них. Ці висновки дають чітке розуміння сильних сторін та недоліків існуючих рішень, встановлюючи орієнтир для майбутнього проектування систем та планування функцій.

## 1.6 Висновки розділу 1

У підсумку проведений аналіз дав змогу сформувати цілісне розуміння специфіки ринку прокату велосипедів, ключових тенденцій галузі та вимог до сучасних веб-рішень. Визначено очікування користувачів різних категорій та

окреслено пріоритети, які мають безпосередній вплив на якість майбутнього сервісу:

· простота взаємодії;

· доступність;

· надійність;

· актуальність даних у режимі реального часу.

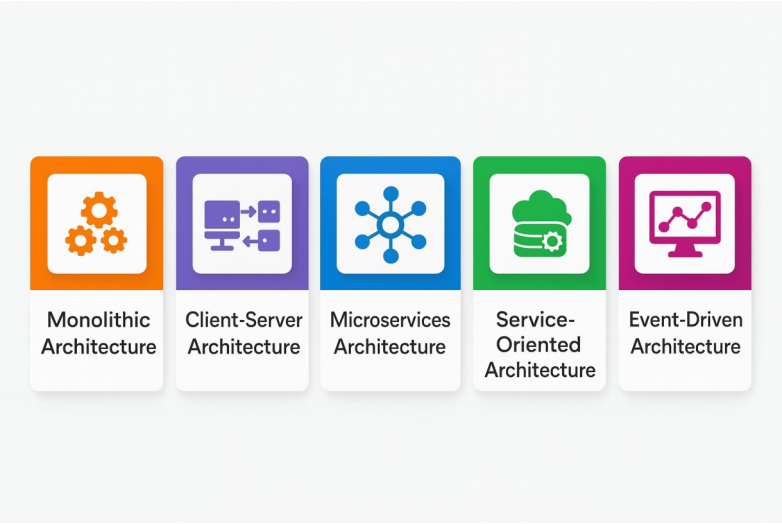
Узагальнення функціональних і нефункціональних вимог дозволило чітко окреслити межі та критерії майбутньої системи, а аналіз наявних платформ виявив практичні орієнтири та обмеження, яких варто дотримуватися при проектуванні. Отримані результати формують обґрунтовану основу для переходу до проєктної частини та забезпечують логічну послідовність подальшого розроблення.

# РОЗДІЛ 2. ПРОЄКТУВАННЯ ТА РОЗРОБКА ПРОГРАМНОГО РІШЕННЯ

Розділ представляє дизайн і розробку веб-додатку для прокату велосипедів. У ньому описано архітектурну модель, структуру та реалізацію компонентів бекенду і фронтенду, а також їх інтеграцію між собою. У розділі також пояснюються стратегії, що використовуються для забезпечення надійної роботи, зручності обслуговування коду та безпечного зв'язку між рівнями системи.

## 2.1 Архітектурна модель веб-застосунку

Сучасні веб-додатки базуються на структурованих архітектурних моделях, які визначають, як взаємодіють компоненти системи, як дані проходять через систему та як розподіляються обов'язки. Вибір відповідної архітектури має прямий вплив на масштабованість, зручність обслуговування, гнучкість розгортання та загальну надійність додатка [7, 8]. Перш ніж визначати структуру системи прокату велосипедів, необхідно розглянути кілька широко використовуваних архітектурних парадигм та оцінити їх придатність для розробки продукту на ранній стадії.




Рисунку 2.1 – Сучасні архітектури веб-додатків

Одним із традиційних підходів є монолітна архітектура, в якій усі функції додатка розміщені в одному розгортаємій одиниці. Її основні переваги включають мінімальні вимоги до інфраструктури, спрощене початкове налаштування та зменшену координацію між модулями. Однак монолітні системи мають тенденцію ставати жорсткими в міру зростання, що ускладнює незалежне масштабування або заміну модулів. Для додатка, який, як очікується, буде розширюватися за рахунок додаткових функцій, таких як інтеграція платежів, відстеження в режимі реального часу або адміністративні панелі інструментів, такі обмеження значно уповільнять подальший розвиток.

Архітектура мікросервісів пропонує протилежний підхід, розкладаючи систему на незалежні сервіси, що спілкуються за допомогою легких протоколів. Ця модель є високомасштабованою, дозволяє командам працювати незалежно та підтримує технологічну гетерогенність. Однак вона вносить складність розподіленої системи, вимагає високого рівня зрілості DevOps та ретельного поводження з міжсервісною комунікацією та кінцевою узгодженістю. Для сервісу прокату велосипедів на початковій стадії з помірним набором функцій впровадження мікросервісів спричинить непотрібні операційні витрати без пропорційних вигод.

Іншою широко використовуваною моделлю є архітектура клієнт-сервер, типова для систем SPA (Single Page Application) [9, 12]. У цьому підході фронтенд-додаток і бекенд-сервіс розробляються і розгортаються незалежно, але взаємодіють через стабільний API-шар. Таке розділення дозволяє гнучко розробляти інтерфейс користувача, спрощує масштабування бекенду і дає можливість у майбутньому впроваджувати декілька клієнтів (наприклад, мобільні додатки). Однак ця архітектура все одно вимагає визначення внутрішньої структури бекенду, особливо щодо перевірки даних, інкапсуляції бізнес-логіки та взаємодії з базою даних.

З огляду на потреби проекту – модульне зростання, передбачуване обслуговування, чітке розділення функцій та структурована бізнес-логіка –

трирівнева архітектура обрана як найбільш підходяще рішення. У цій моделі система поділяється на:

· Рівень презентації – односторінковий додаток Angular, відповідальний за візуальне представлення, маршрутизацію та взаємодію з користувачем [12];

· Рівень сервісу – .NET REST API, що управляє бізнес-правилами, автентифікацією, робочим процесом оренди та логікою додатка [12];

· Рівень даних – реляційна база даних, доступ до якої здійснюється через Entity Framework Core як ORM для узгодженого, строго типізованого збереження даних [12].

Концептуальна структура обраної архітектури представлена на рисунку 2.2

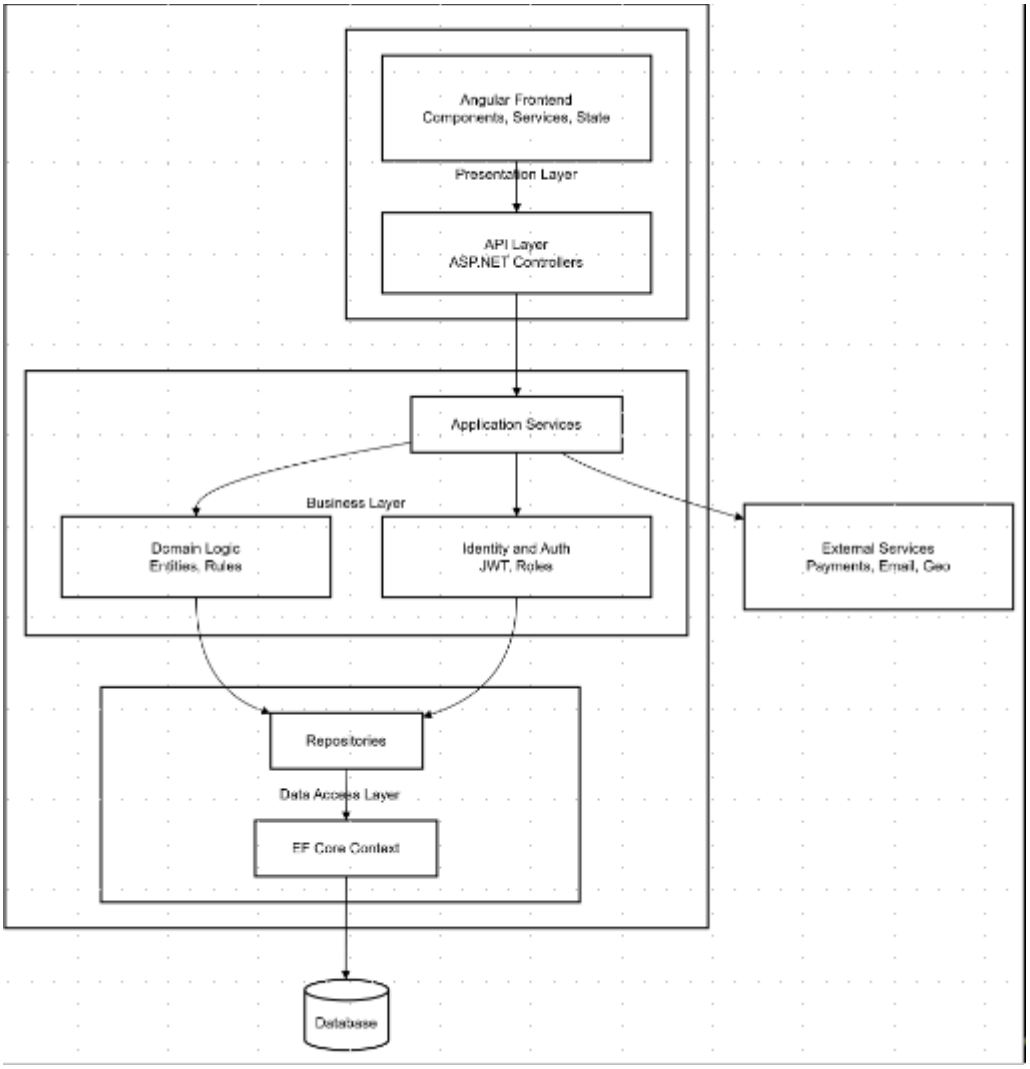



Рисунок 2.2 – Трирівнева архітектура для додатка прокату велосипедів

Трирівнева модель обрана тому, що вона забезпечує сувору ізоляцію відповідальності. Представницький рівень Angular працює незалежно від внутрішньої логіки бекенду, що дозволяє поступово переробляти інтерфейс користувача без впливу на API. Сервісний рівень централізує бізнес-правила та забезпечує єдиний інтерфейс для всіх зовнішніх клієнтів. EF Core додатково забезпечує контрольований та послідовний підхід до доступу до даних, дозволяючи еволюцію схеми та покращену тестованість. Ця архітектурна структура також спрощує адаптацію нових розробників, оскільки кожен рівень має чітко визначену роль та набір обов'язків.

Іншою важливою причиною вибору цього підходу є його придатність для ітеративної розробки. Ранні версії системи можуть зосередитися на основній функціональності оренди без необхідності складної інфраструктури або розподіленого управління системою. У міру зростання проекту архітектуру можна поступово розширювати, наприклад, шляхом впровадження рівнів кешування, фонової обробки або інтеграції зовнішніх сервісів, не порушуючи існуючу структуру. Обрана модель забезпечує достатню гнучкість для цих майбутніх розширень, зберігаючи початкову складність на відповідному рівні.

Підсумовуючи, архітектурний аналіз продемонстрував, що монолітні системи пропонують недостатню модульність, тоді як мікросервіси вимагають надмірних операційних витрат на ранній стадії розробки. Трирівнева архітектура з Angular як рівнем презентації, .NET REST API як рівнем сервісу та обробкою бази даних на базі EF Core забезпечує збалансовану та структурно надійну основу для створення веб-додатку для прокату велосипедів [12]. Вона забезпечує чіткість, зручність обслуговування та еволюційну масштабованість, зберігаючи при цьому керованість всієї системи на початкових етапах розробки.

## 2.2 Проєктування та реалізації бекедну з використанням .NET

Бекенд додатка реалізований як структурована та розширювана служба, побудована на .NET, де загальна логіка слідує гібридному підходу, що поєднує принципи вертикальної архітектури з функціональним дизайном Minimal API [3, 12, 13, 15]. Такий підхід робить кожну кінцеву точку самостійною операцією, побудованою навколо спеціального запиту, але при цьому дозволяє використовувати репозиторії, служби та DTO-мапування для забезпечення чіткості коду та підтримки довгострокової масштабованості. Така структура забезпечує достатнє розділення обов'язків, щоб уникнути накопичення монолітної бізнес-логіки, одночасно зберігаючи простоту та прямоту розробки кінцевих точок, орієнтованих на зрізи [13]. У майбутніх ітераціях цю модель можна легко розширити до більш доменно-орієнтованої або модульної архітектури, не порушуючи її основну організацію.

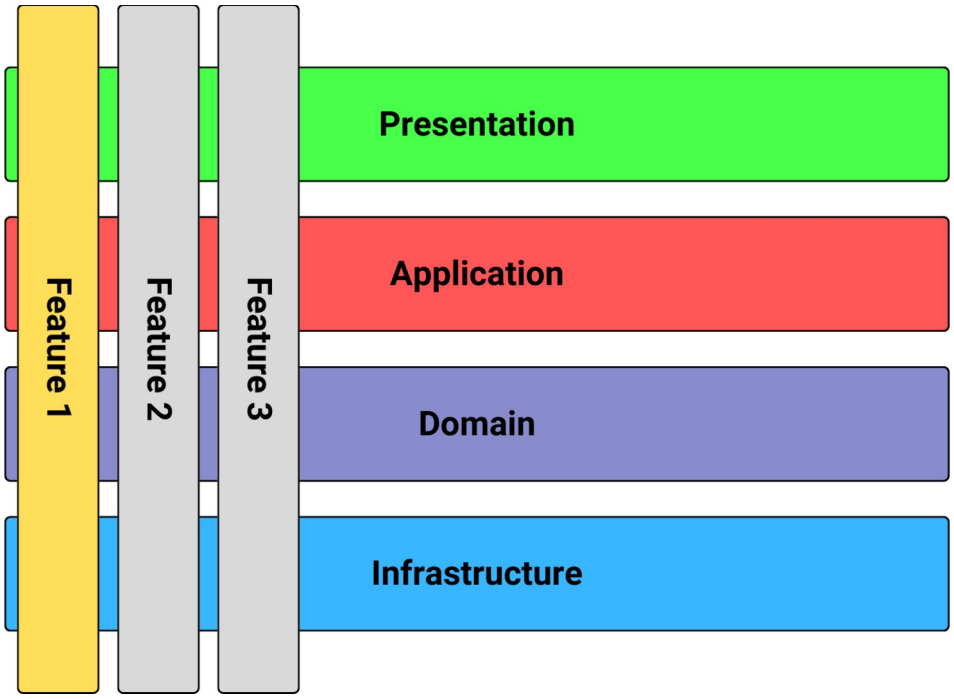



Рисунок 2.3 – Схема архітектури вертикального зрізу [13]

Доменний рівень зосереджений навколо двох ключових сутностей: User (Користувач) та Bike (Велосипед), що представляють ідентифікацію автентифікації та одиниці інвентаризації для операцій з оренди. Обидві сутності

моделюються чітко і послідовно, щоб вони відображали тільки основні атрибути, необхідні для бізнес-функціональності. Для забезпечення правильності та розширюваності проект використовує DTO для передачі даних, ізолюючи об'єкти рівня персистентності від API-контрактів [12, 15]. Це запобігає випадковому розкриттю внутрішніх структур і робить систему більш стійкою до версій та структурної еволюції. Очікується, що на цьому етапі буде включена діаграма класів UML для ілюстрації концептуальної моделі двох основних сутностей та їх полів, відносин і обмежень.

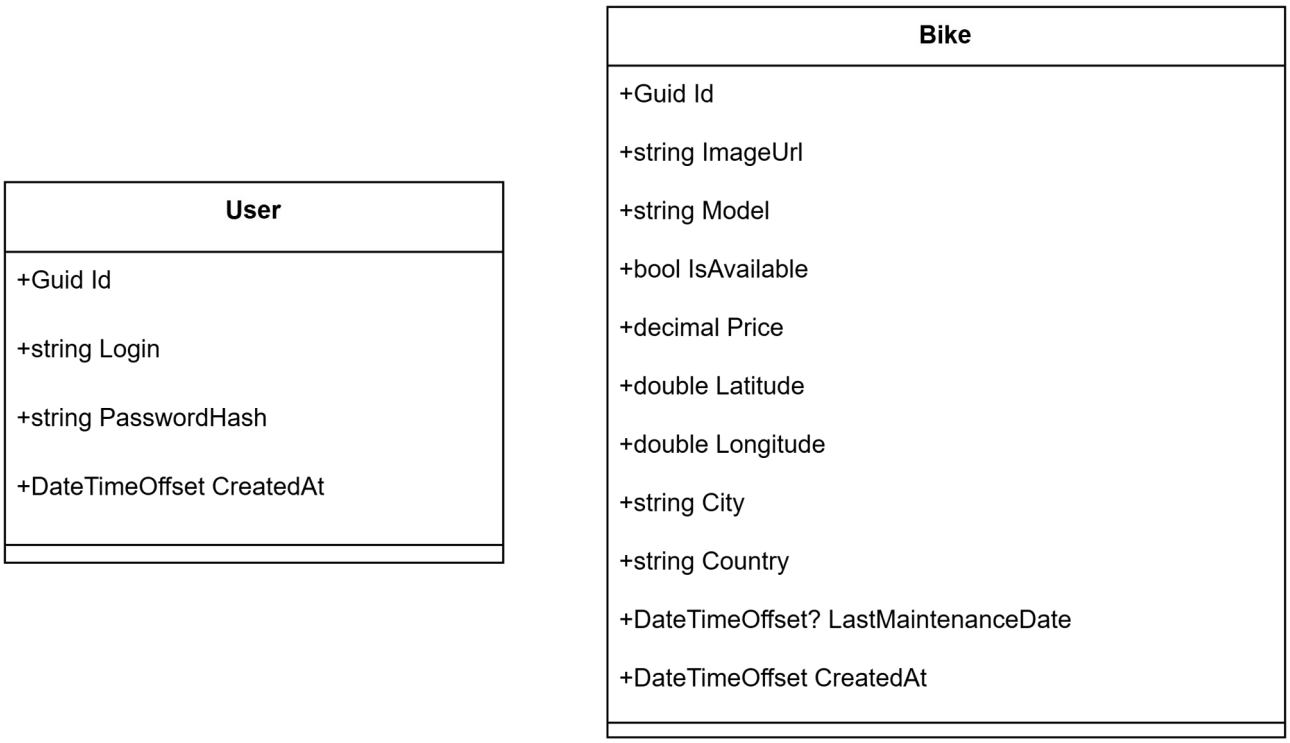



Рисунок 2.4 – Діаграма UML сутностей домену користувача та велосипеда

Використовуючи сутність користувача веб-додаток реалізує власноруч створену систему автентифікації та авторизації на основі JWT (JSON Web Tokens), що забезпечує безпечне зберігання та перевірку облікових даних кожного користувача [16, 17]. Паролі ніколи не зберігаються у вигляді звичайного тексту; натомість вони хешуються за допомогою безпечного алгоритму хешування, який

створює представлення пароля фіксованої довжини, яке неможливо обчислити [16].

```
public class PasswordHasher : IPasswordHasher
{
    private const int SaltSize = 16;
    private const int HashSize = 32;

    private const int Iterations = 10000;

    private static readonly HashAlgorithmName Algorithm = HashAlgorithmName.SHA512;

    7+3 usages  Vitalii
    public string Hash(string password)
    {
        byte[] salt = RandomNumberGenerator.GetBytes( count: SaltSize);
        byte[] hash = Rfc2898DeriveBytes.Pbkdf2(password, salt, Iterations, Algorithm,  outputLength: HashSize);

        return $"{Convert.ToHexString(hash)}-{Convert.ToHexString(salt)}";
    }

    4+3 usages  Vitalii
    public bool Verify(string password, string passwordHash)
    {
        string[] parts = passwordHash.Split('-');

        byte[] salt = Convert.FromHexString(parts[1]);
        byte[] hash = Convert.FromHexString(parts[0]);

        byte[] inputHash = Rfc2898DeriveBytes.Pbkdf2(password, salt, Iterations, Algorithm,  outputLength: HashSize);

        return hash.SequenceEqual(inputHash);
    }
}
```

Рисунок 2.5 – Код сервісу хешування паролей

Під час входу в систему додаток порівнює хеш введеного пароля із збереженим хешем, щоб перевірити особу користувача. Після аутентифікації генерується токен JWT, що містить заявки користувача, який потім передається з наступними запитами для авторизації доступу до захищених кінцевих точок [16, 17]. Такий підхід відокремлює питання аутентифікації від бізнес-логіки, забезпечує бездержавне управління сеансами та дозволяє здійснювати детальний контроль доступу користувачів.

```
public class AuthService(IUsersRepository repository, IPasswordHasher hasher, IJwtProvider jwtProvider) : IAuthService
{
    3+3 usages  Vitalii
    public Result<string> Login(string login, string password)
    {
        User? user = repository.GetByLogin(login);

        if (user == null)
        {
            return Result<string>.Fail($"User under @{login} is not found.");
        }

        UserDto userDto = new()
        {
            Id = user.Id,
            Login = user.Login,
            PasswordHash = user.PasswordHash
        };

        return hasher.Verify(password, userDto.PasswordHash)
            ? Result<string>.Ok(jwtProvider.Generate(userDto.Id, userDto.Login))
            : Result<string>.Fail("User password is incorrect.");
    }
```

Рисунок 2.6 — Код сервісу аутентифікації з методом входу

Результати роботи будь-якого сервісу, як от попередній сервіс аутентифікації та хешування паролей, в бекенді інкапсулюються за допомогою спеціального загального типу Result<T>, невеликого, але важливого будівельного блоку, який допомагає уніфікувати потік успішних і невдалих результатів у всіх операціях. Замість того, щоб генерувати винятки глибоко в шарах домену або репозиторію, операції повертають структуровані об'єкти результатів, які обробляються централізовано в шарі Minimal API [3, 5]. Це покращує передбачуваність і спрощує перетворення помилок у відповіді HTTP. Незважаючи на те, що повна глобальна обробка винятків і стратегії версійності ще не впроваджені, поточний підхід вже відповідає хорошим практикам RESTful API, де очікувані та несподівані умови повідомляються детерміновано [15]. Короткий список коду, що демонструє структуру цього шаблону, можна представити тут, щоб показати його склад і логіку.

```
public class Result<T>(bool success, T? value = default, string? error = null)
{
    18 usages
    public bool Success { get; init; } = success;

    6 usages
    public T? Value { get; init; } = value;

    11 usages
    public string? Error { get; init; } = error;

    5 usages  Vitalii
    public static Result<T> Ok() => new( success: true);
    6 usages  Vitalii
    public static Result<T> Ok(T value) => new( success: true, value);

    12 usages  Vitalii
    public static Result<T> Fail(string? error = null) => new( success: false, default, error);
}
```

Рисунок 2.7 – Код типу даних передачі результату операції

Бекенд надає функціональність суворо за принципами RESTful, зосереджуючи кінцеві точки на окремих відповідальностях і передбачуваній поведінці, орієнтованій на ресурси. Поточний обсяг охоплює такі основні операції, як автентифікація користувачів (реєстрація та вхід), пошук велосипедів за різними стратегіями фільтрування (всі, за ID, за містом і країною) та основні дії з управління запасами, включаючи додавання, видалення, оновлення метаданих та оновлення операційного статусу. Ці кінцеві точки внутрішньо координують доступ до сховища, перевірку DTO, відображення сутностей та побудову кінцевого результату. Оскільки архітектура орієнтована на сегменти, кожна кінцева точка залишається незалежною, одночасно використовуючи спільні служби для загальної логіки, такої як хешування, перевірка та взаємодія з базою даних. Невеликий ілюстративний список, що показує склад однієї такої кінцевої точки, може бути вставлений тут, щоб підкріпити описану структуру, не розкриваючи деталей, чутливих до реалізації.

```
public static class GetBikeById
{
    1 usage  Vitalii
    public static void MapGetBikeById(this RouteGroupBuilder group)
    {
        group.MapGet( pattern: "/{bikeId:guid}", handler: Handle)
            .WithSummary("Get certain bike by it's Id")
            .AllowAnonymous();
    }

    2 usages  Vitalii
    public record Bike(Guid Id, string ImageUrl, string Model, string Availability, string City, string Country, DateTimeOffset LastMaintenanceDate);

    5 usages  Vitalii
    public record Response(Bike Bike);

    3 usages  Vitalii
    public static Results<Ok<Response>, NotFound<string>> Handle(Guid bikeId, IBikesService bikesService)
    {
        BikeDto? bike = bikesService.ReadById(bikeId);

        if (bike == null)
        {
            return TypedResults.NotFound("Reading error: Cannot find bike behind this Id");
        }

        return TypedResults.Ok(
            new Response(
                new Bike(
                    Id: bike.Id,
                    ImageUrl: bike.ImageUrl,
                    Model: bike.Model,
                    Availability: bike.IsAvailable ? "Available" : "Rented",
                    City: bike.City,
                    Country: bike.Country,
                    LastMaintenanceDate: bike.LastMaintenanceDate)));
    }
}
```

Рисунок 2.8 – Код кінцевої точки отримання даних по конкретному велосипеду

Збереження даних обробляється за допомогою Entity Framework Core, що дозволяє бекенду покладатися на зрілий ORM і уникати шаблонного SQL на ранніх етапах розробки. EF Core покращує зручність обслуговування, забезпечуючи сильну типізацію, підтримку міграції та інструменти для розвитку бази даних. Репозиторії об'єднують логіку ORM, щоб зберегти узгодженість шаблонів запитів і абстрагувати їх від вищих рівнів. Це гарантує, що сервісний рівень працює тільки з чітко визначеними методами домену, а не з необробленими викликами бази даних. У всій структурі бекенду відображення між об'єктами домену та DTO наразі реалізовано за допомогою легких методів відображення, а в майбутньому планується вдосконалення за допомогою статичних допоміжних засобів відображення або спеціальних класів перевірки. На цьому етапі можна вставити другу діаграму UML, щоб показати взаємодію між

репозиторіями, сервісами та обробниками запитів на рівні API, що допоможе проілюструвати потік від кінцевої точки до бази даних і назад.

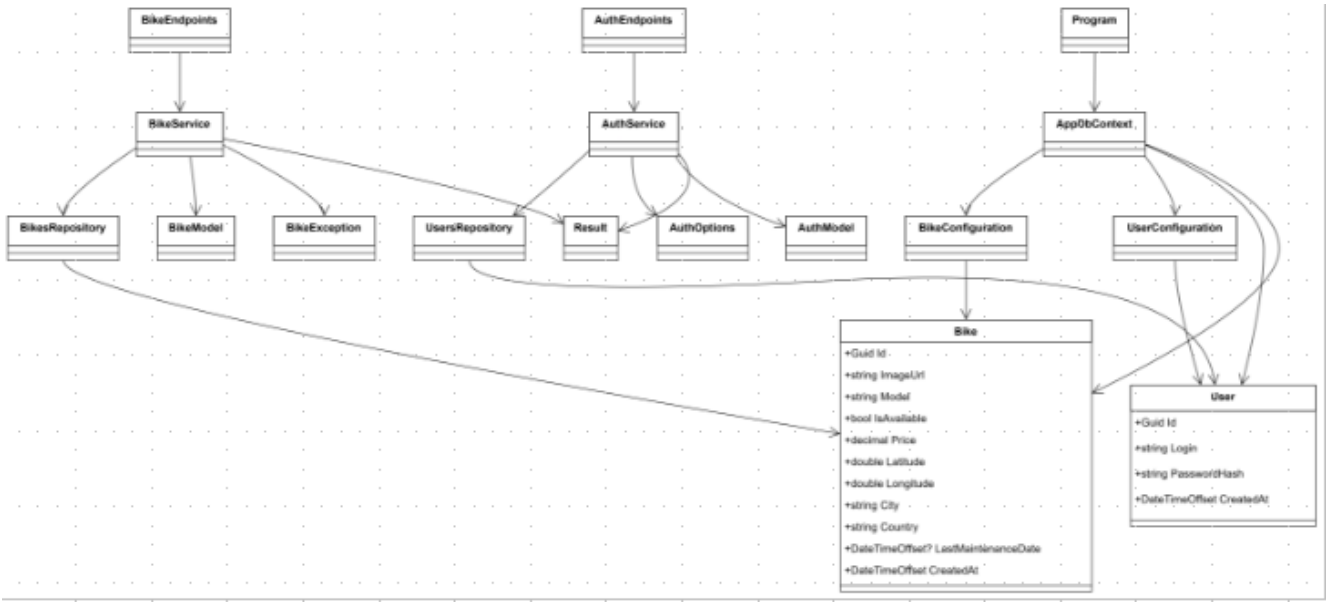



Рисунок 2.9 – Структурна UML-діаграма компонентів бекенду

Безпека забезпечується за допомогою аутентифікації JSON Web Token. Для безпечного зберігання облікових даних користувачів використовується спеціальний механізм хешування, а аутентифікація JWT Bearer налаштована для авторизації доступу до захищених операцій [16, 17]. Завдяки інтеграції авторизації в кінцеві точки Minimal API, дизайн забезпечує чітке розділення між публічними діями (реєстрація та вхід) і приватними операціями (управління велосипедами, операції з профілем). Це гарантує, що неавтентифіковані користувачі не зможуть змінювати системні дані, при цьому зберігаючи легкість процесу автентифікації та можливість його розширення в майбутньому за допомогою ролей або додаткових заяв.

Документація API генерується автоматично за допомогою , що спрощує візуалізацію потоків запитів-відповідей, тестування кінцевих точок під час розробки та підтримку узгодженості між завданнями інтеграції інтерфейсу. У поєднанні з робочим процесом Result<T> OpenAPI чітко повідомляє про очікувані

формати успіху та невдачі для кожної операції. Тут можна вставити компактний список коду, що ілюструє налаштування мінімального API, сконфігурованого за допомогою .

```
builder.Services.AddAuthentication()
    .AddJwtBearer(
        options =>
        {
            options.TokenValidationParameters = new TokenValidationParameters
            {
                IssuerSigningKey =
                    new SymmetricSecurityKey(Encoding.UTF8.GetBytes(builder.Configuration["JWT:Key"]!)),
                ValidateIssuer = false,
                ValidateAudience = false,
                ValidateLifetime = true,
                ValidateIssuerSigningKey = true,
                ClockSkew = TimeSpan.Zero
            };
        });
builder.Services.AddAuthorization();

builder.Services.Configure<JwtOptions>(builder.Configuration.GetSection( key: "JWT"));

builder.Services.AddOpenApi();
```

Рисунок 2.10 – Код конфігурації аутентифікації та документації

В результаті архітектура бекенду забезпечує стабільну, розширювану та прозору основу для системи. Вона підтримує чітке розділення обов'язків, пропонує передбачувані операційні потоки та забезпечує плавну інтеграцію з фронтендом Angular. Поєднання EF Core для збереження даних, JWT для автентифікації, логіки сервісу, орієнтованої на результат, та гібридної структури, орієнтованої на сегменти, гарантує, що бекенд залишається достатньо простим для швидкої розробки зараз, але при цьому достатньо надійним для майбутнього масштабування, модулізації та вдосконалення.

### 2.3 Розробка фронтенду з використанням Angular

Клієнтська частина системи реалізована за допомогою Angular, фреймворку, що забезпечує чітко структуроване середовище для створення масштабованих

веб-додатків. Його модульна архітектура, основа TypeScript та інтегровані інструменти роблять його придатним для довгострокових проектів, де важливими є зручність обслуговування та передбачуване зростання [9]. У контексті додатку для прокату велосипедів Angular забезпечує чітке розділення між логікою презентації, обробкою даних, компонентами інтерфейсу користувача та комунікацією з REST API бекенду [14, 15]. Використання структурування на основі функцій дозволяє додатку залишатися організованим навіть при введенні додаткових модулів, таких як профіль користувача, сторінка деталей велосипеда або адміністративні утиліти, в майбутніх ітераціях. Архітектурне рішення побудувати фронтенд за допомогою Angular відповідає загальній трирівневій структурі системи та забезпечує стабільну, розширювану основу для користувацького інтерфейсу.

Проект використовує структуру каталогів на основі функцій, де кожен функціональний блок додатка містить власні компоненти, шаблони, стилі та пов'язані служби. Структура зазвичай включає каталог /features для доменного інтерфейсу користувача (такого як аутентифікація та велосипеди), область /core для спільних клієнтів, моделей та утиліт, а також область /shared для елементів інтерфейсу користувача, що можуть бути використані повторно. Така організація допомагає ізолювати проблеми та уникнути питань взаємозалежності, а також спрощує спільну розробку. На рисунку 2.11 показано загальне представлення цієї структури та те, як модулі клієнтського додатка згруповані в ієрархію тек.

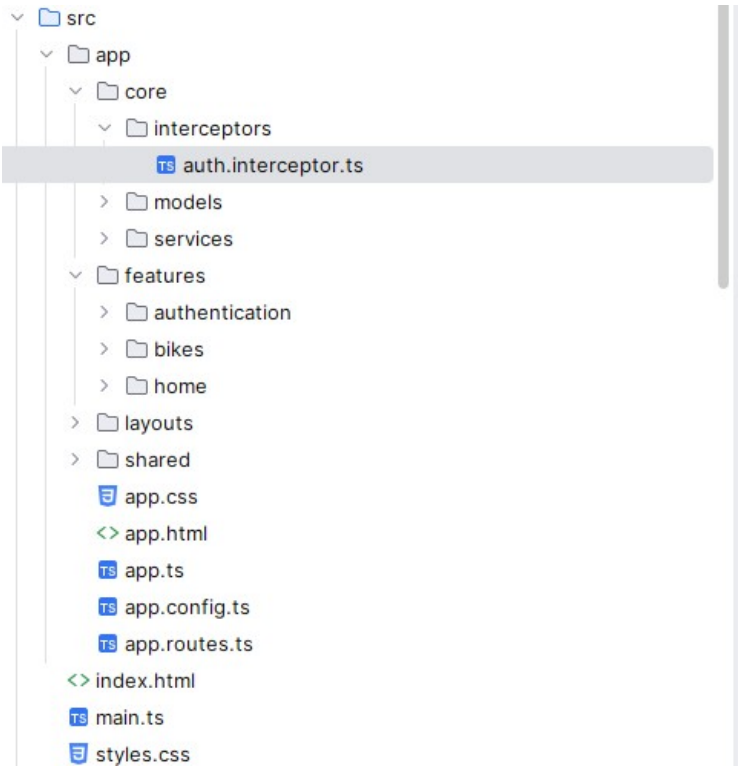



Рисунок 2.11 – Структура тек фронтенд додатку

Основні інтерактивні елементи інтерфейсу реалізовані як компоненти Angular, які представляють незалежні блоки інтерфейсу користувача. Компоненти, такі як форми входу та реєстрації, обробляють реєстрацію користувачів, а компонент списку велосипедів управляє візуалізацією доступних велосипедів, отриманих із сервера. Заплановані компоненти, такі як детальний перегляд велосипеда та сторінка профілю користувача, розширять ці можливості та запровадять більш глибокі сценарії взаємодії. Синтаксис шаблонів Angular у поєднанні з новим API сигналів використовується для відображення оновлень в інтерфейсі користувача [12]. Такий підхід гарантує, що інтерфейс користувача реагує на оновлення даних без необхідності використання складних ланцюжків спостережуваних змінних, хоча RxJS залишається присутнім у комунікаціях на рівні сервісів. Типовий приклад компонента списку велосипедів наведено на рисунку 2.12, який демонструє, як сигнали використовуються для підтримки реактивного стану інтерфейсу користувача наведеного на рисунку 2.13.

```
@Component({
  selector: 'app-bikes-list',
  templateUrl: './bikes-list.component.html',
  imports: [
    BikeCardComponent
  ],
  styleUrls: ['./bikes-list.component.css']
})
export class BikesListComponent implements OnInit {
  bikes = signal<Bike[]>([]);
  loading = signal(false);

  constructor(private bikeService: BikeService) {}
```

Рисунок 2.12 – Код компоненту списку велосипедів

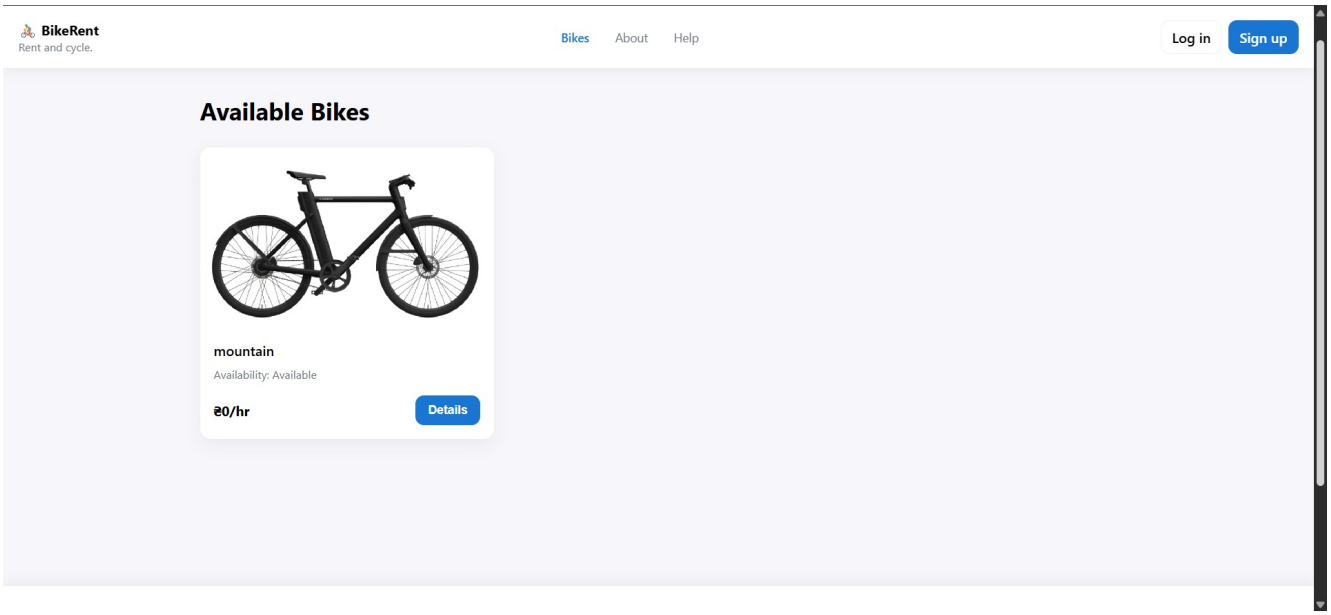



Рисунок 2.13 – Сторінка списку велосипедів

Комунікація з сервером здійснюється за допомогою спеціальних служб Angular, які інкапсулюють всю логіку HTTP. Такий рівень абстракції запобігає прямій залежності компонентів від деталей бекенду та забезпечує відповідне місце для управління автентифікацією, перетворення запитів та операцій з даними, що можуть бути використані повторно. Система використовує HttpClient для надсилання запитів і покладається на інтерцептор HTTP для автоматичного додавання токенів JWT до всіх вихідних запитів, забезпечуючи безпечний доступ до захищених кінцевих точок [9, 10]. Реактивне програмування за допомогою

RxJS залишається частиною рівня сервісу, дозволяючи додатку обробляти відповіді API та трансформувати дані [9]. Для аутентифікації клієнт зберігає токени в локальному сховищі та перевіряє стан авторизації за допомогою легкого механізму. Рисунок 2.14 ілюструє службу аутентифікації, а Рисунок 2.15 ілюструє службу, відповідальну за отримання інформації про велосипеди.

```
@Injectable({ providedIn: 'root' })
export class AuthService {
  private authUrl : string = 'http://localhost:5047/api/auth';

  Show usages  Vitalii
  constructor(private http: HttpClient) { }

  Show usages  Vitalii
  login(credentials: UserCredentialsModel) : Observable<any> {
    return this.http.post<{ token: string }>( url: `${this.authUrl}/login`, { model: credentials } )
      .pipe(tap(response : { token: string } => localStorage.setItem( key: 'token', response.token)));
  }

  Show usages  Vitalii
  register(credentials: UserCredentialsModel) : Observable<any> {
    return this.http.post<{ token: string }>( url: `${this.authUrl}/register`, { model: credentials })
      .pipe(tap(response : { token: string } => localStorage.setItem( key: 'token', response.token)));
  }
```

Рисунок 2.14 – Код сервісу аутентифікації фронтенду

```
@Injectable({
  providedIn: 'root',
})
export class BikeService {
  private apiUrl : string  = 'http://localhost:5047/api/bikes';

  Show usages  Vitalii
  constructor(private http: HttpClient) {}

  Show usages  Vitalii
  getAll(): Observable<Bike[]> {
    return this.http.get<{ bikes: Bike[] }>(this.apiUrl).pipe(
      map(response : { bikes: Bike[] } => response.bikes.map(bike : Bike => ({
        id: bike.id,
        model: bike.model,
        price: 0.0,
        availability: bike.availability,
        imageUrl: 'bike.png',
      })))
    );
  }
}
```

Рисунок 2.15 – Код сервісу зчитування даних про велосипеди

Навігація в додатку забезпечується механізмом маршрутизації Angular, який зіставляє URL-шляхи з конкретними компонентами. Такий підхід створює чітке визначення потоків користувачів додатка. Основні маршрути включають домашню сторінку, форми входу та реєстрації, а також перегляд списку велосипедів. Ця структура вже дозволяє легко розширювати детальні сторінки або розділи, обмежені правилами автентифікації. Рисунок 2.16 показує конфігурацію маршрутизації, яка керує поточною моделлю навігації.

```
export const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: "login", component: LoginComponent },
  { path: "signup", component: SignupComponent },
  { path: "bikes", component: BikesListComponent },
];
```

Рисунок 2.16 – Код маршрутизації об’єктів

Елементи користувацького інтерфейсу, що повторюються в системі, реалізовані як спільні компоненти. Сюди входять настроювані кнопки, макети карток та інші блоки інтерфейсу, що можуть бути використані повторно. Інкапсуляція цих шаблонів забезпечує узгодженість стилю та поведінки в різних частинах системи. Рисунок 2.17 надає приклад компонента кнопки, що може бути використаний повторно, який визначає невелику систему дизайну, що підтримує кілька варіантів і інтегрована в декілька частин додатка.

```
@Component({
  selector: 'app-btn',
  templateUrl: './button.component.html',
  imports: [
    NgClass
  ],
  styleUrls: ['./button.component.css']
})
export class ButtonComponent {
  @Input() variant: 'primary' | 'outline' | 'ghost' = 'primary';
  @Input() disabled : boolean  = false;
}
```

Рисунок 2.17 – Код загальноприйнятого компонента кнопки


Рисунок 2.18 – Компонент кнопки на прикладі кнопки реєстрації

Обраний фреймворк надає низку переваг, які сильно підтримують довгострокові цілі системи. TypeScript забезпечує безпеку типів під час компіляції, зменшуючи кількість помилок і покращуючи зручність обслуговування. Структурована модель компонентів Angular забезпечує послідовну архітектуру додатка та передбачуваний розвиток проекту. Вбудовані інструменти, такі як CLI, система маршрутизації, HttpClient та ReactiveForms, сприяють професійному робочому процесу розробки та скорочують цикли впровадження [9]. Водночас слід визнати недоліки Angular, такі як більший розмір початкового пакета, розлогий синтаксис та більш крута крива навчання порівняно з легшими фреймворками. Однак для проекту, що націлений на довгострокову підтримку, передбачувану складність та інтеграцію з технологіями бекенду корпоративного рівня, такими як .NET, ці компроміси є прийнятними і навіть вигідними.

Загалом, архітектура фронтенду на базі Angular забезпечує стабільність, хороший потенціал масштабованості та чітке розділення між компонентами додатка. Підхід, використаний у цьому проекті, закладає основу для майбутньої реалізації додаткових функцій, таких як панель адміністратора, динамічна фільтрація велосипедів, інтерактивні карти, розширена функціональність профілю та багатші взаємодії в режимі реального часу. Структура може бути розширена без значної переробки, що відповідає довгостроковій перспективі створення надійного, підтримуваного та розширюваного веб-додатку.

### 2.4 Інтеграція фронтенду та бекенду

Інтеграція Angular-додатку з бекендом на .NET є серцем всієї системи. Вона відповідає за те, щоб запити користувачів оброблялися правильно і дані передавалися без помилок. Взаємодія будується за принципами REST: кожна операція має свій чітко визначений endpoint і працює через стандартні HTTP-методи – GET, POST, PUT, DELETE [15]. Фронтенд формує запити, пакує введені користувачем дані у DTO і отримує відповіді у JSON, які потім обробляє. Це дає чітке розділення: презентаційний шар працює з користувачем, а бекенд керує логікою, валідацією, автентифікацією та доступом до бази через репозиторії.

У типовій роботі додатку Angular запити йдуть через сервіси, де вся логіка комунікації зібрана в одному місці. Наприклад, коли користувач логіниться, AuthService формує запит з обліковими даними і відправляє його на бекенд. Там endpoint приймає Request, викликає метод IAuthService.Login і повертає Result<string> – або успішний результат із JWT-токеном, або повідомлення про помилку. API обгортає це у HTTP-відповідь, і фронтенд зберігає токен для подальших авторизованих запитів.

```
public static Results<Ok<Response>, ProblemHttpResult> Handle([FromBody] Request request, IAuthService authService)
{
    Result<string> result = authService.Login(request.Model.Login, request.Model.Password);

    if (result is { Success: true, Value: null })
    {
        return TypedResults.Problem(
            detail: "Login error: We are working onto fixing this problem",
            statusCode: StatusCodes.Status500InternalServerError);
    }

    return result.Success
        ? TypedResults.Ok(new Response(result.Value))
        : TypedResults.Problem(detail: result.Error);
}
```

Рисунок 2.19 – Код обробки запиту на вхід

Авторизація на фронтенді працює автоматично через Angular HTTP Interceptor: він бере токен із localStorage і додає його до заголовка Authorization для всіх захищених запитів [9]. Бекенд перевіряє токен і підтверджує авторизацію. Так користувач може взаємодіяти із сервером безперервно, без зайвих повторних входів.

```
intercept(req: HttpRequest<any>, next: HttpHandler) : Observable<HttpEvent<any>> {
  const token : string | null = localStorage.getItem( key: 'token');

  if (token) {
    req = req.clone({
      setHeaders: { Authorization: `Bearer ${token}` }
    });
  }

  return next.handle(req);
}
```

Рисунок 2.20 – Код перехоплення та ін’єкції токену

Управління велосипедами працює за тією ж схемою. Фронтенд викликає методи BikeService, вони відправляють запити на бекенд, а сервіс BikesService на сервері обробляє бізнес-логіку: перевіряє дані, мапить сутності на DTO, працює з

репозиторієм і повертає JSON. Наприклад, при отриманні всіх велосипедів сервіс тягне сутності, перетворює на DTO, а фронтенд реактивно оновлює інтерфейс через Angular Signals [9].

Помилки обробляються строго: валідаційні або репозиторні винятки не летять на фронтенд просто так. Вони загортаються в Result<T>.Fail і перетворюються на відповідні HTTP-відповіді. DTO гарантують, що внутрішні сутності, як Bike або User, не виставляються прямо, що дає змогу змінювати бекенд без шкоди для UI.

Для наочності на рисунку 2.21 показано життєвий цикл запиту до між фронтендом та бекендом.

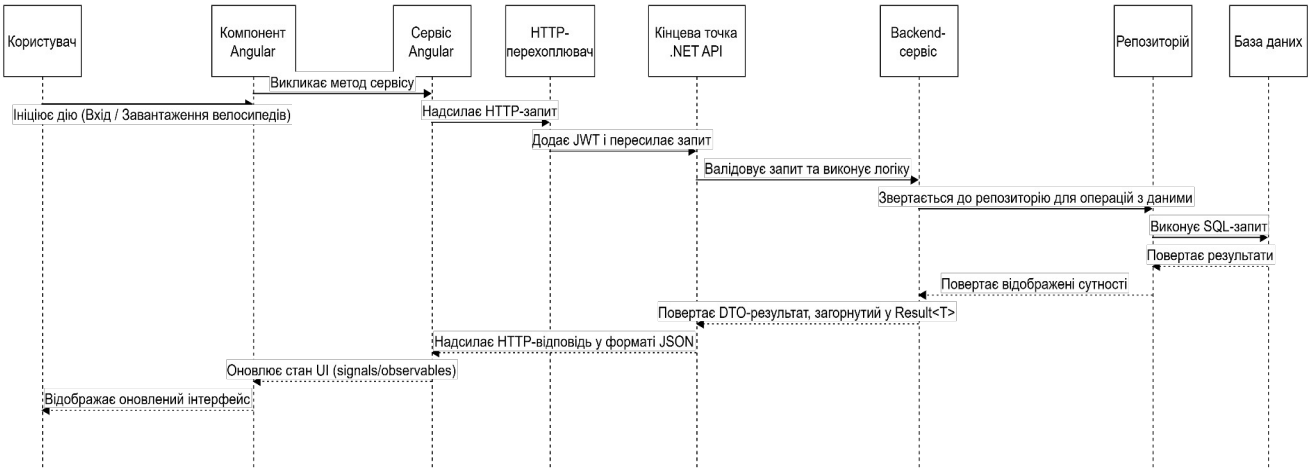



Рисунок 2.21 – Діаграма послідовності із запитом користувача

В цілому, така інтеграція створює міцний і гнучкий зв’язок між Angular і .NET. REST, DTO, токени, обробка помилок і реактивне оновлення UI формують цілісну модель, що дозволяє системі рости, зберігаючи передбачувану і надійну взаємодію між клієнтом і сервером [12].

## 2.5 Висновок до розділу 2

У цьому розділі описано процес розробки системи прокату велосипедів та показано, як трирівнева архітектура використовується для інтеграції сервісів Angular, .NET та репозиторіїв EF Core. Ця архітектура забезпечує організованість системи, її легке обслуговування та розширення в майбутньому.

Бекенд використовує функціональні сервіси разом із шаблоном Result<T> для виконання уніфікованих завдань автентифікації та управління велосипедами, а фронтенд застосовує реактивні форми, спостережувані об'єкти та перехоплювачі Angular для забезпечення безпечного та реактивного користувацького досвіду.

Фронтенд взаємодіє з серверною частиною за допомогою кінцевих точок RESTful, добре структурованих об'єктів передачі даних (DTO) та автентифікації на основі токенів, що забезпечує безперебійну та миттєву комунікацію. Загалом, рішення, розглянуті в цьому розділі, закладають надійну та адаптивну системну основу, яка дозволяє легко впроваджувати нові функції та підтримувати стабільність системи в міру розширення проекту.

# РОЗДІЛ 3. ТЕСТУВАННЯ ПРОГРАМНОЇ СИСТЕМИ

Розділ охоплює аспекти тестування, впровадження та обслуговування веб-додатку для прокату велосипедів. У ньому викладено загальний підхід до тестування, детально описано перевірку компонентів бекенду та фронтенду, а також розглянуто стратегії підтримки та розвитку додатку в майбутньому. У розділі наголошується на практиках, що забезпечують надійність, зручність використання та довгострокову підтримку системи.

## 3.1 Загальний підхід до тестування

Тестування програмної системи було сплановано як безперервний процес, що супроводжує розробку на всіх її етапах. Такий підхід відповідає сучасним уявленням про забезпечення якості програмного забезпечення, згідно з якими тестування розглядається не як інструмент пошуку помилок після завершення розробки, а як засіб запобігання їх виникненню. Вихідною логікою було визначити необхідність перевіряти кожен компонент якомога раніше, виконувати перевірки регулярно та не допускати неконтрольованого внесення змін у кодову базу [2, 3]. Основною метою було забезпечити не лише відповідність реалізованої функціональності початковим вимогам, а й стабільність системи в умовах її поступового функціонального розширення [5]. У контексті сучасного цифрового середовища, де програмні продукти еволюціонують і оновлюються безперервно, такий підхід є базовою умовою підтримки якості [6].

Процес тестування не було заплановано як окремий фінальний етап, відокремлений від основної розробки, оскільки подібна модель характерна для застарілих каскадних підходів і не відповідає сучасним ітеративним методологіям. Натомість було прийнято рішення інтегрувати тестування безпосередньо в життєвий цикл створення програмного забезпечення. У ряді випадків було свідомо призупинено реалізацію функціональності з метою

попереднього опису логіки тестів, після чого подальший розвиток системи здійснювався відповідно до підходу Test Driven Development [12]. Дана методологія передбачає написання тесту до реалізації функціональності, що дозволяє формалізувати вимоги у вигляді перевірюваних умов. Цикл Test Driven Development наведено на рисунку 3.1.

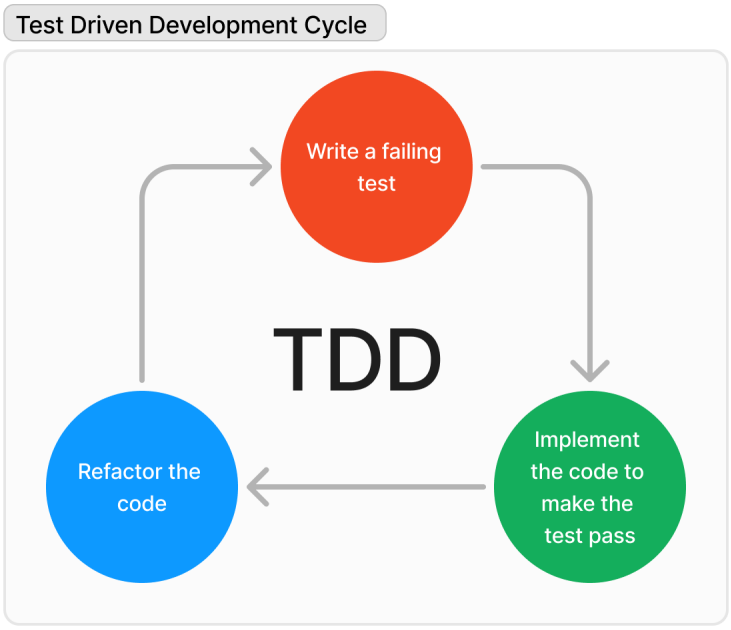



Рисунок 3.1 – Цикл розробки згідно методології Test Driven Development

Застосування TDD було обґрунтовано його здатністю зменшувати кількість логічних помилок, спрощувати подальший рефакторинг коду та підтримувати чіткий зв’язок між вимогами і реалізацією. У такій моделі тестування виконує роль специфікації поведінки системи, а код – засобу її реалізації. Очікуваним наслідком було зменшення кількості дефектів на пізніх етапах розробки та скорочення витрат часу на їх усунення.

Паралельно з використанням TDD було заплановано застосування інших взаємодоповнюючих підходів до тестування. Загальну структуру тестів було вирішено будувати відповідно до принципів піраміди тестування, що є загальноприйнятою моделлю організації тестових перевірок у програмних

системах. Згідно з цією моделлю, більша частина тестів повинна припадати на нижні рівні абстракції, оскільки саме вони є найменш затратними та найбільш стабільними. Графічне представлення даної концепції наведено на рисунку 3.2.

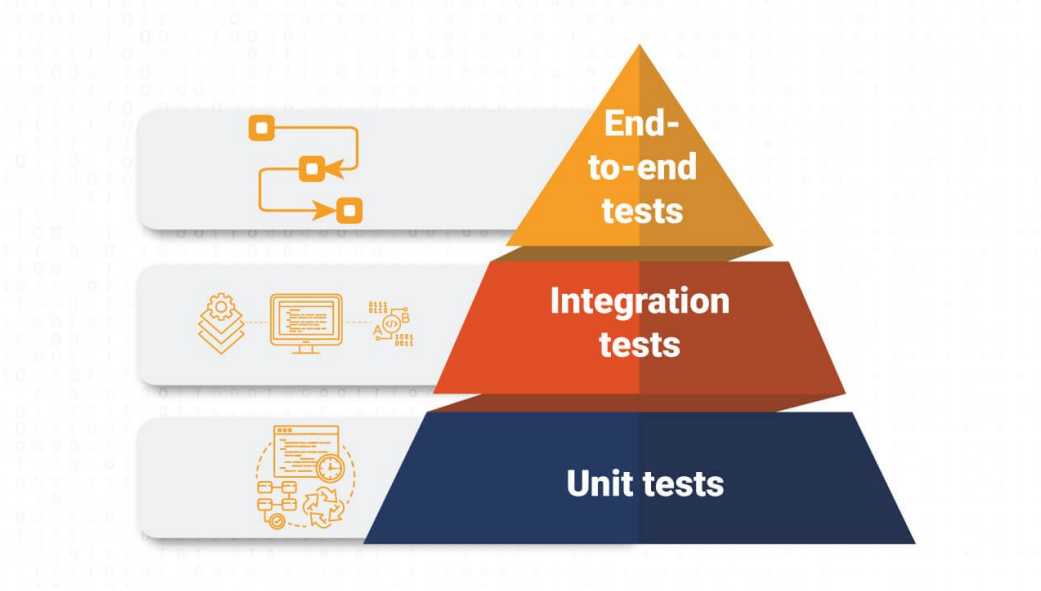



Рисунок 3.2 – Піраміда тестування

На базовому рівні було реалізовано модульне тестування, що включало велику кількість швидких та ізольованих перевірок. Модульні тести у теорії програмної інженерії розглядаються як фундамент системи тестування, оскільки дозволяють перевіряти коректність окремих функцій або класів без залежності від зовнішнього середовища. Було передбачено використовувати ці тести для підтвердження бізнес-логіки серверної частини на платформі .NET, а також ключових функцій клієнтського застосунку, реалізованого з використанням Angular. Основною перевагою цього рівня було визначено можливість швидкого зворотного зв’язку та мінімальні витрати ресурсів на виконання тестів.

Наступним етапом було заплановано проведення інтеграційного тестування, яке в теоретичному аспекті спрямоване на перевірку взаємодії між модулями системи. На цьому рівні окремі компоненти розглядаються вже не ізольовано, а як частини єдиного механізму. Було необхідно підтвердити коректність спільної роботи контролерів, сервісів, бізнес-логіки та шару доступу до даних. Інтеграційні

тести дозволяли виявляти помилки, які не проявляються на рівні окремих модулів, але виникають під час їх взаємодії.

Окремо було заплановано зосередитися на тестуванні API, що в сучасних клієнт-серверних системах розглядається як критичний рівень перевірки. REST API виступає контрактом між клієнтською та серверною частинами, а його стабільність безпосередньо впливає на узгодженість роботи всієї системи. Було передбачено перевіряти коректність відповідей сервера, обробку помилкових запитів, а також відповідність структури даних визначеним контрактам. Такий підхід дозволяє мінімізувати ризик порушення взаємодії між частинами системи під час подальших змін.

Завершальним рівнем стратегії було визначено тестування «від кінця до кінця» (end-to-end), яке в теорії тестування розглядається як спосіб перевірки системи з позиції кінцевого користувача. Даний підхід було охарактеризовано як найбільш ресурсозатратний, проте таким, що дозволяє оцінити працездатність системи в умовах реального використання. У межах проєкту було вирішено не впроваджувати автоматизовані E2E-тести через обмежений масштаб системи, а зосередитися на ручному моделюванні ключових користувацьких сценаріїв [11].

Узагальнюючи, стратегія тестування була сформована як послідовний і теоретично обґрунтований процес, що відповідає сучасним підходам до забезпечення якості програмного забезпечення. Було закладено принципи автоматизації, повторюваності та постійної верифікації результатів. Попри відсутність повноцінного CI/CD-конвеєра, підхід до тестування було організовано дисципліновано та наближено до практик, характерних для комерційних програмних продуктів.

### 3.2 Огляд тестування бекенду та фронтенду

Тестування серверної та клієнтської частин програмної системи було організовано як єдиний безперервний процес, у межах якого кожен рівень

перевірки розглядався окремо, а згодом оцінювався в контексті реальних сценаріїв використання. Такий підхід дозволяв поєднати формальну перевірку окремих компонентів із практичною оцінкою їх взаємодії у цілісному програмному середовищі. Для серверної частини було заплановано більш структуроване тестування бізнес-логіки та поведінки API, тоді як для клієнтської частини основний акцент було зроблено на перевірці станів компонентів, коректності відображення даних і реакції інтерфейсу на дії користувача.

Для практичної оцінки працездатності системи було виконано повторювані зразкові виклики та сценарії обробки даних, результати яких порівнювалися з очікуваною поведінкою. Такий підхід дозволяв не лише зафіксувати правильність роботи окремих функцій, а й простежити, як система поводиться в умовах, наближених до реального використання. У межах модульного тестування серверної частини було перевірено коректність ізольованої логіки сервісів ще до виконання реальних HTTP-викликів, що відповідало загальним принципам раннього виявлення помилок.

```
[Fact]
Vitalii
public void Login_ShouldFail_WhenUserNotFound()
{
    // Arrange
    _usersRepo.Setup(repository ⇒ repository.GetByLogin("user")) // ISetup<IUsersRepository,User?>
        .Returns((User?)null);

    // Act
    Result<string> result = _service.Login("user", password: "password");

    // Assert
    Assert.False(result.Success);
    Assert.Contains("not found", result.Error);
}
```

Рисунок 3.3 – Код одного з модульних тестів бекенд частини

Для оцінки роботи серверних служб було виконано сценарій аутентифікації, у межах якого перевірялася відповідність HTTP-кодів стану та структура JSON-відповідей під час подання коректних облікових даних [15]. Такі перевірки мали

принципове значення, оскільки саме через API клієнтська частина взаємодіє з сервером, а будь-яка нестабільність на цьому рівні безпосередньо впливає на користувацький досвід. Аналогічні підходи було застосовано до основних кінцевих точок системи, зокрема отримання списків велосипедів, створення нових записів і оновлення інформації про бронювання. Послідовність виконання запитів та перевірок узагальнено на рисунку 3.4.


Рисунок 3.4 – Потік тестування аутентифікації бекенду

Окремий акцент було зроблено на перевірці рівня доступу до даних. Для цього було виконано цілеспрямоване тестування репозиторіїв, метою якого було підтвердити коректність повернення результатів запитів і збереження змін у сховищі. Використання фіктивних наборів даних або бази даних у пам’яті дозволяло ізолювати цей рівень від решти системи та зосередитися безпосередньо на логіці створення, фільтрування та оновлення записів. Такий підхід сприяв виявленню дрібних неточностей на ранніх етапах і запобігав їх подальшому впливу на клієнтську частину.

```
[Fact]
Vitalii
public void Add_ShouldPersistUser()
{
    // Arrange
    UsersRepository repo = CreateRepository();
    var user = new User
    {
        Id = Guid.NewGuid(),
        Login = "testuser",
        PasswordHash = "hash"
    };

    // Act
    repo.Add(user);

    // Assert
    User stored = Context.Users.Single();
    Assert.Equal("testuser", stored.Login);
}
```

Рисунок 3.5 – Код модульного тесту користувацького репозиторію

На стороні користувацького інтерфейсу увагу було зосереджено на підтвердженні коректного відображення станів компонентів Angular і адекватної реакції інтерфейсу на дії користувача. Компонент входу в систему було перевірено вручну з метою оцінки роботи повідомлень про помилки, станів елементів керування та механізмів валідації форм. Такий підхід дозволяв виявити проблеми, пов’язані не лише з логікою, а й із сприйняттям інтерфейсу, які часто залишаються поза межами автоматизованих перевірок.

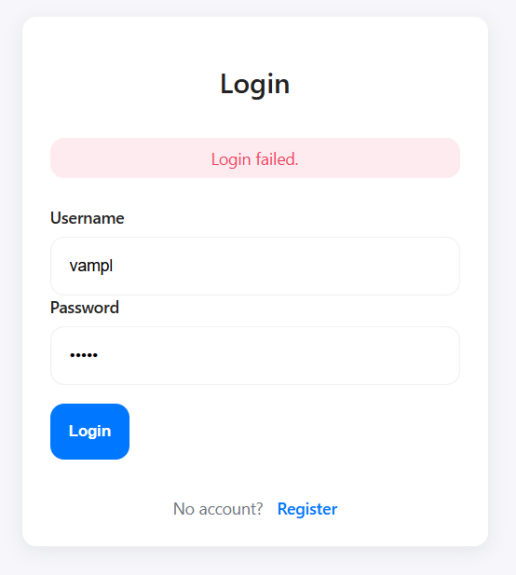



Рисунок 3.6 – Ручне тестування перевірки форми входу в систему

Окрему увагу було приділено тестуванню сервісів Angular, які виконують роль проміжної ланки між API та компонентами інтерфейсу. Було виконано тестування сервісів із використанням імітованих HTTP-викликів для перевірки коректної обробки відповідей сервера та реакції на помилкові сценарії [9, 11]. Це дозволяло зменшити залежність клієнтської частини від внутрішніх змін API та підтримувати стабільну поведінку інтерфейсу в процесі активної розробки.

```
describe( name: 'BikeService', () : void  => {
  let httpClientMock: { get: ReturnType<typeof vi.fn> };
  let service: BikeService;

  beforeEach(() : void  => {
    httpClientMock = {
      get: vi.fn()
    };
    service = new BikeService(httpClientMock as unknown as HttpClient);
  });

  it( name: 'should call the correct API URL in getAll', () : void  => {
    httpClientMock.get.mockReturnValue(of({ bikes: [] }));
    service.getAll().subscribe();
    expect(httpClientMock.get).toHaveBeenCalledWith('http://localhost:5047/api/bikes');
  });

  it( name: 'should transform the bikes response correctly', () : Promise<void>  => {
    const apiResponse = {
      bikes: [
        { id: 1, model: 'A', price: 123, availability: true, imageUrl: 'other.png' },
        { id: 2, model: 'B', price: 456, availability: false, imageUrl: 'other2.png' }
      ]
    };
    httpClientMock.get.mockReturnValue(of(apiResponse));
    return service.getAll().toPromise().then(bikes : Bike[] | undefined  => {
      expect(bikes).toEqual([
        { id: 1, model: 'A', price: 0.0, availability: true, imageUrl: 'bike.png' },
        { id: 2, model: 'B', price: 0.0, availability: false, imageUrl: 'bike.png' }
      ]);
    });
  });
});
```

Рисунок 3.7 – Код модульного тестування сервісів фронтенд частини

У підсумку тестування серверної та клієнтської частин було вибудувано як взаємодоповнюючу систему перевірок. Автоматизовані модульні тести, перевірки поведінки API та доступу до даних забезпечували стабільність і передбачуваність серверної логіки, тоді як тестування інтерфейсу дозволяло оцінити зручність і

логічну цілісність взаємодії користувача з системою. Поєднання цих підходів створювало структурований і прозорий процес контролю якості, узгоджений із сучасними практиками розробки програмного забезпечення.

### 3.3 Висновки до розділу 3

У цьому розділі було розглянуто підхід до тестування програмної системи як невід'ємної складової процесу розробки. Тестування було організовано як безперервний процес, інтегрований у життєвий цикл створення програмного забезпечення, що дозволило поєднати розробку функціональності з її постійною перевіркою та верифікацією відповідності вимогам.

У межах розділу було обґрунтовано використання методології Test Driven Development та принципів піраміди тестування як теоретичної основи побудови стратегії перевірок. Застосування цих підходів дало змогу раціонально розподілити тестові сценарії між різними рівнями системи, зосередивши основну увагу на модульному тестуванні та поступово переходячи до інтеграційних і прикладних перевірок.

Окрему увагу було приділено тестуванню серверної та клієнтської частин системи. Для бекенду було виконано перевірку бізнес-логіки, поведінки API та доступу до даних, тоді як тестування фронтенду було спрямовано на підтвердження коректності роботи сервісів і компонентів інтерфейсу, а також передбачуваності взаємодії користувача з системою. Поєднання автоматизованих і ручних перевірок дозволило оцінити систему як з технічної, так і з користувацької точки зору.

Загалом підходи, розглянуті в цьому розділі, сформували структурований і прозорий процес контролю якості програмного продукту. Попри відсутність повноцінного CI/CD-конвеєра, реалізована стратегія тестування була наближена до практик комерційної розробки та створила надійну основу для подальшого розвитку, підтримки та масштабування системи.

# РОЗДІЛ 4. ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

У розділі розглядаються заходи з охорони праці та безпеки для розробників, які працюють над веб-додатком для прокату велосипедів. У ньому викладено вимоги до дотримання нормативних актів та практик для забезпечення безпечних та ергономічних умов праці.

## 4.1 Охорона праці

При розробці програмного забезпечення для системи прокату велосипедів було враховано вимоги охорони праці та пожежної безпеки, що регламентують умови безпечної праці працівників, які здійснюють інтелектуальну діяльність з використанням персональних електронно-обчислювальних машин. Забезпечення безпечних і нешкідливих умов праці розглядалося як обов’язкова складова організації робочого процесу під час проєктування, реалізації та тестування програмної системи [23].

Нормативною основою забезпечення охорони праці в межах даного проєкту є Закон України «Про охорону праці», який визначає основні положення щодо реалізації конституційного права працівників на належні, безпечні та здорові умови праці [23]. Відповідно до статей 13 і 14 зазначеного Закону, роботодавець зобов’язаний забезпечити безпечні умови праці, а працівник – дотримуватися вимог нормативно-правових актів з охорони праці. Хоча розробка програмного забезпечення не пов’язана з виробничими ризиками підвищеної небезпеки, вона належить до видів діяльності, що мають специфічні фактори ризику, зокрема тривале перебування за комп’ютером, статичні навантаження, психоемоційне напруження та вплив електромагнітного випромінювання [24].

Організація робочого місця розробника програмного забезпечення здійснювалася з урахуванням НПАОП 0.00-7.15-18 “Вимог щодо безпеки та

захисту здоров’я працівників під час роботи з екранними пристроями” [24]. Відповідно до цих норм було враховано вимоги до розміщення периферії, що дозволило зменшити навантаження на органи зору, опорно-руховий апарат і центральну нервову систему.

Робоче місце було організовано таким чином, щоб верхній край екрана монітора знаходився на рівні очей або трохи нижче, а відстань від очей до екрана становила не менше 50–70 см. Клавіатура розміщувалася на відстані 10–30 см від краю робочого столу, що забезпечувало зручне положення кистей рук. Стілець мав регулювання по висоті та куту нахилу спинки, що відповідало ергономічним вимогам та зменшувало ризик розвитку професійних захворювань [31] опорно-рухового апарату [25].

Освітлення робочого місця було організовано відповідно до вимог ДБН В.2.5-28:2018 «Природне і штучне освітлення» [25]. Було забезпечено достатній рівень природного та штучного освітлення, при якому уникалося пряме потрапляння світла на екран монітора та виникнення відблисків. Використання комбінованого освітлення дозволяло підтримувати комфортні умови праці.

Особливу увагу було приділено режиму праці та відпочинку. Відповідно до рекомендацій санітарних норм, під час роботи з комп’ютером було передбачено регулярні перерви для зменшення зорового та нервово-емоційного навантаження [26]. Такий режим сприяв збереженню працездатності, концентрації уваги та запобіганню перевтомі, що є важливим чинником при виконанні складних аналітичних та інженерних завдань.

З урахуванням того, що під час розробки програмного забезпечення використовується електротехнічне обладнання, було враховано вимоги Правил улаштування електроустановок (ПУЕ) [27] та Правил безпечної експлуатації електроустановок споживачів [28]. Робоче місце було підключене до електромережі з використанням справних розеток, заземлення та сертифікованих подовжувачів, що зменшувало ризик ураження електричним струмом. Перед

початком роботи здійснювалася візуальна перевірка стану кабелів живлення та відсутності пошкоджень ізоляції.

Вимоги пожежної безпеки також були враховані під час організації робочого процесу. Нормативною базою в цій сфері є Кодекс цивільного захисту України [29] та Правила пожежної безпеки в Україні [30]. Робоче приміщення було обладнане первинними засобами пожежогасіння, а електричне обладнання використовувалося відповідно до інструкцій виробника. Заборонялося перевантаження електромережі та використання несправних електроприладів.

Під час виконання робіт не використовувалися горючі або вибухонебезпечні матеріали, однак було враховано ризик загоряння електрообладнання внаслідок короткого замикання або перегріву. З цією метою було забезпечено вільний доступ до вентиляційних отворів комп’ютерної техніки, а також дотримано вимог щодо відстаней між обладнанням і стінами приміщення.

З погляду психофізіологічних факторів, розробка програмного забезпечення пов’язана з високим рівнем розумового навантаження, відповідальності за результат та необхідністю тривалої концентрації уваги. Тому було враховано рекомендації щодо зменшення психоемоційного напруження шляхом раціонального планування робочого часу, чергування видів діяльності та дотримання оптимального мікроклімату в приміщенні [26].

Таким чином, у межах даного проєкту було враховано основні вимоги охорони праці та пожежної безпеки, що регламентують умови праці при розробці програмного забезпечення. Дотримання чинної нормативно-правової бази дозволило забезпечити безпечні, комфортні та здорові умови праці, знизити ризики професійних захворювань і створити передумови для ефективної та стабільної роботи над програмним продуктом.

## 4.2 Ергономічні вимоги до робочого місця користувача персональним комп’ютером (ПК)

Ергономічні вимоги до робочого місця користувача персональним комп’ютером є складовою системи безпеки праці та цивільного захисту, оскільки безпосередньо впливають на функціональний стан людини, рівень професійних ризиків і здатність персоналу до адекватних дій у разі виникнення надзвичайних ситуацій. Відповідно до Закону України «Про охорону праці», роботодавець зобов’язаний забезпечити безпечні та нешкідливі умови праці, у тому числі шляхом дотримання ергономічних і санітарно-гігієнічних вимог до робочих місць [23].

Організація робочого місця користувача ПК повинна відповідати вимогам Державних санітарних норм і правил щодо роботи з візуальними дисплейними терміналами електронно-обчислювальних машин, які встановлюють обов’язкові параметри розміщення обладнання, меблів і засобів відображення інформації [24]. Робоче місце має забезпечувати можливість підтримання раціональної робочої пози, при якій зменшується статичне м’язове навантаження та запобігається розвитку функціональних порушень опорно-рухового апарату.

Робочий стіл повинен мати достатню площу для розміщення монітора, клавіатури, маніпулятора та допоміжних засобів, а його висота — відповідати антропометричним характеристикам користувача. Сидіння користувача ПК повинно бути регульованим по висоті та куту нахилу спинки і забезпечувати підтримку поперекового відділу хребта, що відповідає вимогам санітарного законодавства та ергономічних стандартів [24, 25]. Недотримання цих вимог розглядається як фактор підвищення ризику професійних захворювань.

Засоби відображення інформації повинні розміщуватися таким чином, щоб зорове навантаження не перевищувало допустимих норм. Відповідно до санітарних правил, відстань від очей користувача до екрана монітора повинна становити не менше 50 см, а кут огляду забезпечувати комфортне сприйняття

інформації без надмірного напруження м’язів шиї та органів зору [24, 31]. Раціональне розміщення екрана та робочої поверхні користувача персонального комп’ютера наведено на рисунку 4.2.

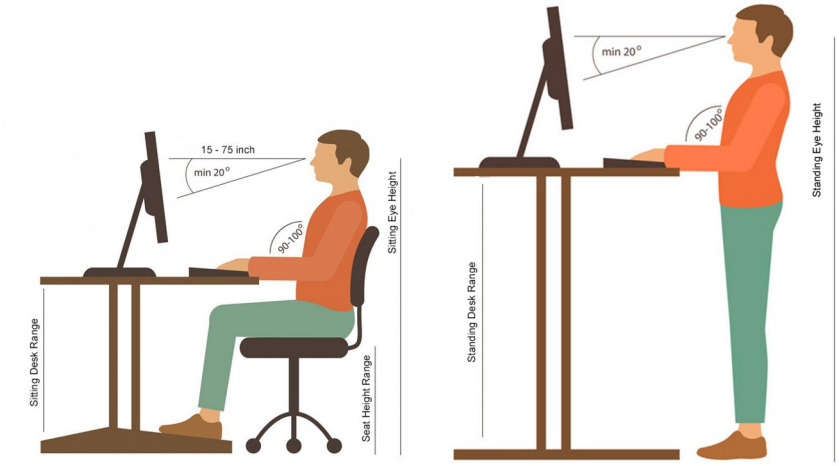



Рисунок 4.2 – Ергономіка робочого місця у сидячому та стоячому положеннях

Клавіатура та пристрої керування повинні розташовуватися у зоні досяжності рук без необхідності надмірного нахилу або напруження плечового поясу. Таке розміщення відповідає принципам ергономіки праці та спрямоване на профілактику професійних порушень верхніх кінцівок, що підтверджується вимогами чинних нормативних документів з охорони праці [25].

Освітлення робочого місця повинно відповідати вимогам державних будівельних норм і санітарних правил щодо природного та штучного освітлення. Освітленість у зоні роботи з документами та клавіатурою повинна бути рівномірною, без відблисків на екрані монітора, що є обов’язковою умовою безпечної роботи з ПК [25, 31, 32]. Приклад організації робочого місця з урахуванням вимог до освітлення та розміщення обладнання наведено на рисунку 4.3.


Рисунок 4.3 – Правильна та неправильна ергономіка освітлення робочого місця

Мікроклімат виробничого приміщення, у якому експлуатується персональний комп’ютер, повинен відповідати нормативним параметрам температури, відносної вологості та швидкості руху повітря, визначеним санітарними нормами [26]. Відхилення від встановлених значень негативно впливає на працездатність користувачів і може знижувати рівень готовності персоналу до дій у разі виникнення надзвичайних ситуацій [31, 32].

З точки зору безпеки в надзвичайних ситуаціях ергономічно організоване робоче місце розглядається як елемент підвищення стійкості функціонування об’єкта [32]. Відповідно до Кодексу цивільного захисту України, умови праці та розміщення обладнання повинні забезпечувати можливість безпечної евакуації, зменшення ризику травмування та швидкого реагування персоналу у разі пожежі, аварій або інших небезпечних подій [29].

Таким чином, дотримання ергономічних вимог до робочого місця користувача персональним комп’ютером є нормативно обґрунтованою необхідністю, яка поєднує вимоги охорони праці, санітарного законодавства та системи цивільного захисту [31, 32]. Виконання цих вимог сприяє зниженню професійних ризиків, збереженню здоров’я персоналу та підвищенню рівня безпеки в умовах надзвичайних ситуацій.

# ВИСНОВКИ

У кваліфікаційній роботі було розглянуто питання проектування та реалізації сучасної веб-системи прокату велосипедів із використанням актуальних підходів і технологій програмної інженерії. У пояснювальній записці увагу було зосереджено не лише на реалізації функціональних можливостей системи, але й на обґрунтуванні архітектурних рішень, забезпеченні надійності, підтримуваності та потенціалу подальшого розвитку програмного продукту. Обрана тематика є актуальною в умовах зростання ролі цифрових сервісів у сфері міської мобільності та поширення моделей спільного використання транспортних засобів.

На початковому етапі було виконано аналіз предметної області та визначено основні вимоги до системи. Сформовано перелік функціональних вимог, що охоплюють автентифікацію користувачів, перегляд і пошук велосипедів, управління даними та відображення доступності ресурсів. Окрему увагу було приділено нефункціональним вимогам, зокрема питанням безпеки, продуктивності та масштабованості. Проведений аналіз дозволив сформувати цілісне бачення системи та забезпечив логічну основу для прийняття подальших проектних рішень. Дослідження існуючих програмних рішень у цій сфері дало змогу виявити типові підходи та обмеження, що було враховано при виборі технологічної платформи.

У процесі проектування було обґрунтовано використання трирівневої архітектури, яка поєднує клієнтську частину на базі Angular, серверну частину у вигляді .NET REST API та реляційну базу даних із доступом через Entity Framework Core. Такий підхід забезпечив чітке розмежування відповідальностей між рівнями системи, підвищив зрозумілість структури та створив передумови для подальшого розширення функціональності. Серверна частина була організована з використанням функціонально-орієнтованого підходу та концепції вертикального розрізу, що сприяло логічній ізоляції бізнес-функцій. Клієнтська

частина була реалізована з урахуванням принципів модульності та повторного використання компонентів.

Взаємодія між фронтендом і бекендом реалізована на основі RESTful-підходу із застосуванням об’єктів передачі даних та уніфікованих механізмів обробки помилок. Використання автентифікації на основі токенів забезпечило контроль доступу до захищених ресурсів, а застосування реактивних механізмів обробки даних на стороні клієнта підвищило передбачуваність та стабільність роботи інтерфейсу користувача. Сукупність цих рішень дозволила сформувати узгоджену та логічно цілісну модель взаємодії компонентів системи.

Перевірка працездатності програмного рішення підтвердила коректність реалізованої логіки та відповідність системи визначеним вимогам. Тестування серверної частини охоплювало перевірку бізнес-логіки та кінцевих точок API, тоді як тестування клієнтської частини було зосереджене на поведінці компонентів і коректності користувацьких сценаріїв. Отримані результати дозволили виявити потенційні напрями подальшого вдосконалення, зокрема у сфері автоматизації тестування та оптимізації продуктивності.

У підсумку в межах кваліфікаційної роботи було розроблено функціональний прототип веб-додатку для прокату велосипедів, який демонструє практичне застосування сучасних архітектурних підходів, технологій інтеграції фронтенду та бекенду, а також базових механізмів безпеки. Поставлена мета роботи, що полягала у створенні масштабованої, структурованої та технічно обґрунтованої веб-системи для управління сервісом прокату велосипедів, була досягнута в повному обсязі. Отримані результати підтверджують доцільність обраних технологічних і архітектурних рішень та можуть бути використані як основа для подальшого розвитку й практичного впровадження системи.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Методичні вказівки до виконання кваліфікаційної роботи магістра для здобувачів спеціальності 121 – Інженерія програмного забезпечення, усіх форм навчання [Електронний ресурс] – Режим доступу: https://elartu.tntu.edu.ua/handle/lib/50316 (дата звернення: 14.12.2025).

2. ECMA-334. C# Language Specification.. Чинний від 15.11.2023. 2023. 671 с.

3. Tour of C# [Електронний ресурс] – Режим доступу: https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/ (дата звернення: 14.12.2025).

4. .NET GitHub Repository [Електронний ресурс] – Режим доступу: https://github.com/dotnet/ (дата звернення: 14.12.2025).

5. AltexSoft. The Good and the Bad of .NET Framework Programming [Електронний ресурс] – Режим доступу: https://www.altexsoft.com/blog/the-good-and-the-bad-of-net-framework-programming/ (дата звернення: 14.12.2025).

6. ASP.NET Overview [Електронний ресурс] – Режим доступу: https://learn.microsoft.com/en-us/aspnet/overview (дата звернення: 14.12.2025).

7. Gayathri M. The Importance of .NET Core and MVC in Modern Software Development : Bachelor of Technology. 2020. 3 с.

8. Juraj K. Progressive Web App with Angular 2 and ASP.NET. : Bachelor Information and Communications Technology / School of Technology, Communication and Transport. 2017. 61 с.

9. Angular Documentation [Електронний ресурс] – Режим доступу: https://angular.dev/overview (дата звернення: 14.12.2025).

10. Angular HTTP Guide [Електронний ресурс] – Режим доступу: https://angular.dev/guide/http (дата звернення: 14.12.2025).

11. Angular CLI End-to-End Testing [Електронний ресурс] – Режим доступу: https://angular.dev/tools/cli/end-to-end (дата звернення: 14.12.2025).

12. Tutorial: ASP.NET Core with Angular [Електронний ресурс] – Режим доступу: https://learn.microsoft.com/en-us/visualstudio/javascript/tutorial-asp-net-core-with-angular (дата звернення: 14.12.2025).

13. Vertical Slice Architecture [Електронний ресурс] – Режим доступу: https://github.com/nadirbad/VerticalSliceArchitecture (дата звернення: 14.12.2025).

14. REST (Representational State Transfer) [Електронний ресурс] – Режим доступу: https://techterms.com/definition/rest (дата звернення: 14.12.2025).

15. HackerNoon. RESTful API Designing Guidelines and Best Practices [Електронний ресурс] – Режим доступу: https://surl.li/bennfe (дата звернення: 14.12.2025).

16. Curity. JWT Best Practices [Електронний ресурс]. – Режим доступу: https://curity.io/resources/learn/jwt-best-practices/ (дата звернення: 14.12.2025).

17. Token-based Authentication Overview [Електронний ресурс] – Режим доступу: https://surl.li/mzggec (дата звернення: 14.12.2025).

18. Mobisoft Infotech. Custom Web Development Services [Електронний ресурс] – Режим доступу: https://surl.li/kxaimg (дата звернення: 14.12.2025).

19. Strava. Global Heatmap [Електронний ресурс] – Режим доступу: https://www.strava.com/maps/global-heatmap (дата звернення: 14.12.2025).

20. Donkey Republic. Bike Sharing Platform [Електронний ресурс] – Режим доступу: https://www.donkey.bike/ (дата звернення: 14.12.2025).

21. Spokeo Bike. Bike Rental Service [Електронний ресурс] – Режим доступу: https://spokeo.bike/ (дата звернення: 14.12.2025).

22. Citi Bike NYC. Official Website [Електронний ресурс] – Режим доступу: https://citibikenyc.com/ (дата звернення: 14.12.2025).

23. Закон України «Про охорону праці» [Електронний ресурс] – Режим доступу: https://zakon.rada.gov.ua/laws/show/2694-12#Text (дата звернення: 14.12.2025).

24. НПАОП 0.00-7.15-18 “Вимог щодо безпеки та захисту здоров’я працівників під час роботи з екранними пристроями” [Електронний ресурс] –

Режим доступу: https://zakon.rada.gov.ua/laws/show/z0508-18#Text (дата звернення: 14.12.2025).

25. ДБН В.2.5-28-2018. Природне і штучне освітлення [Електронний ресурс] – Режим доступу: https://e-construction.gov.ua/laws_detail/3718602641056466807 (дата звернення: 14.12.2025).

26. Державні санітарні норми мікроклімату виробничих приміщень (ДСН 3.3.6.042-99) [Електронний ресурс] – Режим доступу: https://zakon.rada.gov.ua/rada/show/va042282-99#Text (дата звернення: 14.12.2025).

27. Правила улаштування електроустановок [Електронний ресурс] – Режим доступу: https://mev.gov.ua/storinka/pravyla-ulashtuvannya-elektroustanovok (дата звернення: 14.12.2025).

28. Правила безпечної експлуатації електроустановок споживачів [Електронний ресурс] – Режим доступу: https://zakon.rada.gov.ua/laws/show/z0093-98 (дата звернення: 14.12.2025).

29. Кодекс цивільного захисту України [Електронний ресурс] – Режим доступу: https://zakon.rada.gov.ua/laws/show/5403-17 (дата звернення: 14.12.2025).

30. Правила пожежної безпеки в Україні (НАПБ А.01.001-2014) [Електронний ресурс] – Режим доступу: https://zakon.rada.gov.ua/laws/show/z0252-15#Text (дата звернення: 14.12.2025).

31. Стручок В.С. Методичний посібник для здобувачів освітнього ступеня «магістр» всіх спеціальностей денної та заочної (дистанційної) форм навчання «БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ» / Тернопіль: ФОП Паляниця В. А., –156 с.

32. Стручок В.С. Навчальний посібник «ТЕХНОЕКОЛОГІЯ ТА ЦИВІЛЬНА БЕЗПЕКА. ЧАСТИНА «ЦИВІЛЬНА БЕЗПЕКА»» / Тернопіль: ФОП Паляниця В. А., – 156 с.

# ДОДАТКИ

Лістинг А.1 – Визначення сутності користувача

```
namespace BikesRental.API.Features.Auth.Entities;

public class User
{
    public Guid Id { get; set; }

    public string Login { get; set; } = string.Empty;

    public string PasswordHash { get; set; } = string.Empty;

    public DateTimeOffset CreatedAt { get; set; } =
DateTimeOffset.UtcNow;
}
```

Лістинг А.2 – Визначення сутності велосипеда

```
namespace BikesRental.API.Features.Bikes.Entities;

public class Bike
{
    public Guid Id { get; set; }

    public string ImageUrl { get; set; } = string.Empty;

    public string Model { get; set; } = string.Empty;

    public bool IsAvailable { get; set; }

    public decimal Price { get; set; }

    public double Latitude { get; set; }

    public double Longitude { get; set; }

    public string City { get; set; } = string.Empty;

    public string Country { get; set; } = string.Empty;

    public DateTimeOffset? LastMaintenanceDate { get; set; }

    public DateTimeOffset CreatedAt { get; set; } =
DateTimeOffset.UtcNow;
}
```

Лістинг А.3 – Репозиторій користувача для доступу до даних

```
using BikesRental.API.Features.Auth.Entities;
using BikesRental.API.Features.Bikes.Exceptions;
using BikesRental.API.Infrastructure;
using Microsoft.EntityFrameworkCore;

namespace BikesRental.API.Features.Auth.Data;

public class UsersRepository(AppDbContext context) :
IUsersRepository
{
    public User? GetByLogin(string login)
    {
        return context.Users
            .AsNoTracking()
            .FirstOrDefault(user => user.Login == login);
    }

    public void Add(User newUser)
    {
        if (context.Users.Any(user => user.Id == newUser.Id))
        {
            throw new DuplicateRecordException(
                $"{nameof(AppContext)}.{nameof(context.Users)}
already contains record behind {newUser.Id}");
        }

        context.Users.Add(newUser);

        context.SaveChanges();
    }
}
```

Лістинг А.4 – Логіка служби автентифікації

```
using BikesRental.API.Common;
using BikesRental.API.Features.Auth.Data;
using BikesRental.API.Features.Auth.Entities;
using BikesRental.API.Features.Auth.Models;
using BikesRental.API.Features.Bikes.Exceptions;

namespace BikesRental.API.Features.Auth.Services;

public class AuthService(IUsersRepository repository,
IPasswordHasher hasher, IJwtProvider jwtProvider) : IAuthService
{
    public Result<string> Login(string login, string password)
    {
        User? user = repository.GetByLogin(login);

        if (user == null)
        {
```

```
            return Result<string>.Fail($"User under @{login} is not
found.");
        }

        UserDto userDto = new()
        {
            Id = user.Id,
            Login = user.Login,
            PasswordHash = user.PasswordHash
        };

        return hasher.Verify(password, userDto.PasswordHash)
            ? Result<string>.Ok(jwtProvider.Generate(userDto.Id,
userDto.Login))
            : Result<string>.Fail("User password is incorrect.");
    }

    public Result<string> Register(string login, string password)
    {
        UserDto userDto = new()
        {
            Login = login,
            PasswordHash = hasher.Hash(password)
        };

        try
        {
            repository.Add(new User {Id = userDto.Id, Login =
userDto.Login, PasswordHash = userDto.PasswordHash});

            return
Result<string>.Ok(jwtProvider.Generate(userDto.Id, userDto.Login));
        }
        catch (DuplicateRecordException)
        {
            return Result<string>.Fail("Creation error: Cannot
create a user that already created.");
        }
    }
}
```

Лістинг А.5 – Бізнес-логіка служби велосипедів

```
using BikesRental.API.Common;
using BikesRental.API.Features.Bikes.Data;
using BikesRental.API.Features.Bikes.Entities;
using BikesRental.API.Features.Bikes.Exceptions;
using BikesRental.API.Features.Bikes.Models;

namespace BikesRental.API.Features.Bikes.Services;
```

```
public class BikesService(IBikesRepository bikesRepository) :
IBikesService
{
    public List<BikeDto> ReadAll()
    {
        return bikesRepository.GetAll()
            .Select(
                bike => new BikeDto
                {
                    Id = bike.Id,
                    ImageUrl = bike.ImageUrl,
                    Model = bike.Model,
                    IsAvailable = bike.IsAvailable,
                    Price = bike.Price,
                    City = bike.City,
                    Country = bike.Country
                })
            .ToList();
    }

    public List<BikeDto> ReadAllInCityAndCountry(string city, string
country)
    {
        return bikesRepository.GetAllByCityAndCountry(city, country)
            .Select(
                bike => new BikeDto
                {
                    Id = bike.Id,
                    ImageUrl = bike.ImageUrl,
                    Model = bike.Model,
                    IsAvailable = bike.IsAvailable,
                    Price = bike.Price,
                    City = bike.City,
                    Country = bike.Country
                })
            .ToList();
    }

    public BikeDto? ReadById(Guid bikeId)
    {
        Bike? bike = bikesRepository.Get(bikeId);

        return bike == null
            ? null
            : new BikeDto
            {
                Id = bike.Id,
                ImageUrl = bike.ImageUrl,
                Model = bike.Model,
                IsAvailable = bike.IsAvailable,
                Price = bike.Price,
```

```
                City = bike.City,
                Country = bike.Country
            };
    }

    public Result<BikeDto> Create(BikeDto newBike)
    {
        if (newBike.Id == Guid.Empty)
        {
            throw new ArgumentException("Validation error: Cannot
create a bike with empty identifier field.");
        }

        if (string.IsNullOrWhiteSpace(newBike.ImageUrl))
        {
            throw new  ArgumentException("Validation error: Cannot
create a bike with empty image url.");
        }

        if (string.IsNullOrWhiteSpace(newBike.Model))
        {
            throw new ArgumentException("Validation error: Cannot
create a bike with nullable or white spaced model name.");
        }

        if (string.IsNullOrWhiteSpace(newBike.City))
        {
            throw new ArgumentException("Validation error: Cannot
create a bike with empty city name.");
        }

        if (string.IsNullOrWhiteSpace(newBike.Country))
        {
            throw new ArgumentException("Validation error: Cannot
create a bike with empty country name.");
        }

        if (newBike.LastMaintenanceDate.Date >
DateTimeOffset.Now.Date)
        {
            throw new ArgumentException("Validation error: Cannot
create a bike with maintenance date further than current date.");
        }

        try
        {
            bikesRepository.Add(
                new Bike
                {
                    Id = newBike.Id,
                    ImageUrl = newBike.ImageUrl,
```

```
                    Model = newBike.Model,
                    IsAvailable = newBike.IsAvailable,
                    Price = newBike.Price,
                    City = newBike.City,
                    Country = newBike.Country,
                    LastMaintenanceDate =
newBike.LastMaintenanceDate
                });

            return Result<BikeDto>.Ok(newBike);
        }
        catch (DuplicateRecordException)
        {
            return Result<BikeDto>.Fail("Creation error: Cannot
create a bike that already created.");
        }
    }

    public Result<BikeDto> Update(BikeDto updatedBike)
    {
        if (updatedBike.Id == Guid.Empty)
        {
            throw new ArgumentException("Validation error: Cannot
update a bike with new empty identifier field.");
        }

        if (string.IsNullOrWhiteSpace(updatedBike.ImageUrl))
        {
            throw new  ArgumentException("Validation error: Cannot
create a bike with empty image url.");
        }

        if (string.IsNullOrWhiteSpace(updatedBike.Model))
        {
            throw new ArgumentException("Validation error: Cannot
create a bike with new nullable or white spaced model name.");
        }

        if (string.IsNullOrWhiteSpace(updatedBike.City))
        {
            throw new ArgumentException("Validation error: Cannot
create a bike with empty city name.");
        }

        if (string.IsNullOrWhiteSpace(updatedBike.Country))
        {
            throw new ArgumentException("Validation error: Cannot
create a bike with empty country name.");
        }
```

```
        if (updatedBike.LastMaintenanceDate.Date >
DateTimeOffset.Now.Date)
        {
            throw new ArgumentException("Validation error: Cannot
create a bike with new maintenance date further than current
date.");
        }

        try
        {
            bikesRepository.Update(
                new Bike
                {
                    Id = updatedBike.Id,
                    Model = updatedBike.Model,
                    IsAvailable = updatedBike.IsAvailable,
                    LastMaintenanceDate =
updatedBike.LastMaintenanceDate
                });

            return Result<BikeDto>.Ok();
        }
        catch (NotFoundRecordException)
        {
            return Result<BikeDto>.Fail(error: "Edition error:
Cannot update a bike that doesn't exists.");
        }
    }

    public Result<BikeDto> Remove(Guid removeBikeId)
    {
        if (removeBikeId == Guid.Empty)
        {
            throw new ArgumentException("Validation error: Cannot
delete a bike by empty identifier field.");
        }

        try
        {
            bikesRepository.Delete(removeBikeId);

            return Result<BikeDto>.Ok();
        }
        catch (InvalidOperationException)
        {
            return Result<BikeDto>.Fail("Removal error: Cannot
remove a bike that doesn't exists.");
        }
    }

    public Result<BikeDto> Rent(Guid rentBikeId)
```

```
    {
        if (rentBikeId == Guid.Empty)
        {
            throw new ArgumentException("Validation error: Cannot
rent a bike by empty identifier.");
        }

        try
        {
            bikesRepository.UpdateAvailability(rentBikeId,
updatedAvailability: false);

            return Result<BikeDto>.Ok();
        }
        catch (NotFoundRecordException)
        {
            return Result<BikeDto>.Fail("Rental error: Cannot rent a
bike that doesn't exists.");
        }
    }
}
```

Лістинг А.6 – Кінцеві точки API автентифікації

```
using BikesRental.API.Common;
using BikesRental.API.Features.Auth.Services;
using Microsoft.AspNetCore.Http.HttpResults;
using Microsoft.AspNetCore.Mvc;

namespace BikesRental.API.Features.Auth.Endpoints;

public static class Login
{
    public static void MapLogin(this RouteGroupBuilder group)
    {
        group.MapPost("/login", Handle)
            .WithSummary("Let's user to get into account")
            .AllowAnonymous();
    }

    public record Model(string Login, string Password);

    public record Request(Model Model);

    public record Response(string Token);

    public static Results<Ok<Response>, ProblemHttpResult>
Handle([FromBody] Request request, IAuthService authService)
    {
        Result<string> result =
authService.Login(request.Model.Login, request.Model.Password);
```

```
        if (result is { Success: true, Value: null })
        {
            return TypedResults.Problem(
                "Login error: We are working onto fixing this
problem",
                statusCode:
StatusCodes.Status500InternalServerError);
        }

        return result.Success
            ? TypedResults.Ok(new Response(result.Value))
            : TypedResults.Problem(detail: result.Error);
    }
}

public static class Register
{
    public static void MapRegister(this RouteGroupBuilder group)
    {
        group.MapPost("/register", Handle)
            .WithSummary("Let's user create a new account")
            .AllowAnonymous();
    }

    public record Model(string Login, string Password);

    public record Request(Model Model);

    public record Response(string Token);

    public static Results<Ok<Response>, ProblemHttpResult>
Handle([FromBody] Request request, IAuthService authService)
    {
        Result<string> result =
authService.Register(request.Model.Login, request.Model.Password);

        if (result is { Success: true, Value: null })
        {
            return TypedResults.Problem(
                "Login error: We are working onto fixing this
problem",
                statusCode:
StatusCodes.Status500InternalServerError);
        }

        return result.Success
            ? TypedResults.Ok(new Response(result.Value))
            : TypedResults.Problem(detail: result.Error);
    }
}
```

Лістинг А.7 – Контекст бази даних додатка

```
using System.Reflection;
using BikesRental.API.Features.Auth.Entities;
using BikesRental.API.Features.Bikes.Entities;
using Microsoft.EntityFrameworkCore;

namespace BikesRental.API.Infrastructure;

public class AppDbContext(DbContextOptions<AppDbContext> options) :
DbContext(options)
{
    public DbSet<Bike> Bikes { get; set; }

    public DbSet<User> Users { get; set; }

    protected override void OnModelCreating(ModelBuilder
modelBuilder)
    {

modelBuilder.ApplyConfigurationsFromAssembly(Assembly.GetExecutingAs
sembly());

        base.OnModelCreating(modelBuilder);
    }
}
```

Лістинг А.8 – Налаштоване виключення «не знайдено»

```
namespace BikesRental.API.Features.Bikes.Exceptions;

public class NotFoundRecordException : Exception
{
    public NotFoundRecordException()
    {
    }

    public NotFoundRecordException(string message) : base(message)
    {
    }
}
```

Лістинг А.9 – Сервіс управління велосипедами

```
import { Injectable } from '@angular/core';
import { map, Observable } from 'rxjs';
import { HttpClient } from '@angular/common/http';
import { Bike } from '../models/bike.model'
```

```
@Injectable({
  providedIn: 'root',
})
export class BikeService {
  private apiUrl = 'http://localhost:5047/api/bikes';

  constructor(private http: HttpClient) {}

  getAll(): Observable<Bike[]> {
    return this.http.get<{ bikes: Bike[] }>(this.apiUrl).pipe(
      map(response => response.bikes.map(bike => ({
        id: bike.id,
        model: bike.model,
        price: 0.0,
        availability: bike.availability,
        imageUrl: 'bike.png',
      })))
    );
  }
}
```

Лістинг А.10 – Перехоплювач запитів

```
import { Injectable } from '@angular/core';
import { HttpHandler, HttpRequest, HttpInterceptor, HttpEvent } from
'@angular/common/http';
import { Observable } from 'rxjs';

Injectable()
export class AuthInterceptor implements HttpInterceptor {
  intercept(req: HttpRequest<any>, next: HttpHandler) :
Observable<HttpEvent<any>> {
    const token : string | null = localStorage.getItem('token');
    if (token) {
      req = req.clone({
        setHeaders: { Authorization: `Bearer ${token}` }
      });
    }
    return next.handle(req);
  }

}
```

Лістинг А.11 – Інформаційна карта велосипеда

```
import { Component, Input } from '@angular/core';
import { Bike } from '../../../core/models/bike.model';
import { ButtonComponent } from '../button/button.component';
import { RouterLink } from '@angular/router';
```

```
@Component({
  selector: 'app-bike-card',
  templateUrl: './bike-card.component.html',
  imports: [
    ButtonComponent,
    RouterLink,
  ],
  styleUrls: ['./bike-card.component.css']
})
export class BikeCardComponent {
  @Input() bike!: Bike;
}
```

Лістинг А.12 – Тестування сервісу управління велосипедами

```
import { describe, it, expect, vi, beforeEach } from 'vitest';
import { BikeService } from './bike.service';
import { HttpClient } from '@angular/common/http';
import { of } from 'rxjs';

describe('BikeService', () => {
  let httpClientMock: { get: ReturnType<typeof vi.fn> };
  let service: BikeService;
  beforeEach(() => {
    httpClientMock = {
      get: vi.fn()
    };
    service = new BikeService(httpClientMock as unknown as
HttpClient);
  });

  it('should call the correct API URL in getAll', () => {
    httpClientMock.get.mockReturnValue(of({ bikes: [] }));
    service.getAll().subscribe();

expect(httpClientMock.get).toHaveBeenCalledWith('http://localhost:50
47/api/bikes');
  });

  it('should transform the bikes response correctly', () => {
    const apiResponse = {
      bikes: [
        { id: 1, model: 'A', price: 123, availability: true,
imageUrl: 'other.png' },
        { id: 2, model: 'B', price: 456, availability: false,
imageUrl: 'other2.png' }
      ]
    };
    httpClientMock.get.mockReturnValue(of(apiResponse));
    return service.getAll().toPromise().then(bikes => {
      expect(bikes).toEqual([
```

```
        { id: 1, model: 'A', price: 0.0, availability: true,
imageUrl: 'bike.png' },
        { id: 2, model: 'B', price: 0.0, availability: false,
imageUrl: 'bike.png' }
      ]);
    });
  });
});
```

**УДК 004.41**
**Барабаш В. ст. гр. Спм-61**
**Петрик М. Р., д.ф.-м.н., проф.**
Тернопільський національний технічний університет імені Івана Пулюя

**ПРОЄКТУВАННЯ ТА РОЗРОБКА ВЕБЗАСТОСУНКУ ДЛЯ ОРЕНДИ ВЕЛОСИПЕДІВ З ВИКОРИСТАННЯМ .NET ТА ANGULAR**

**Barabash V.**
**Dr., Prof., Petryk M. R.**
Ternopil Ivan Puluj National Technical University

**DESIGN AND DEVELOPMENT OF A WEB APPLICATION FOR BICYCLE RENTAL USING .NET AND ANGULAR**

Ключові слова: веб-застосунок, оренда велосипедів, .NET, Angular, REST API, front-end, back-end, база даних, тестування.
Keywords: web application, bicycle rental, .NET, Angular, REST API, front-end, back-end, database, testing

У цій роботі досліджується проектування та впровадження повнофункціональної веб-системи для управління операціями з прокату велосипедів, з акцентом на інтеграцію високопродуктивного серверного рівня .NET з клієнтським інтерфейсом на базі Angular.
Робота спрямована на розробку архітектури, яка забезпечує надійні робочі процеси бронювання, послідовну обробку даних та інтерактивну візуалізацію активів, що здаються в прокат. У дослідженні оцінюються сучасні підходи до програмного забезпечення для спільного використання транспортних засобів, визначаються архітектурні шаблони, які найкраще підтримують масштабованість та зручність користування, і застосовуються до розробки прототипу системи.
Завдяки ітеративному проектуванню, впровадженню та тестуванню, дисертація демонструє, як сучасні веб-технології та комунікація на основі REST можуть бути поєднані для створення стабільної та орієнтованої на користувача платформи управління прокатом.

**Список використаних джерел:**

1. Bike/Car Rental App Development – “Bike Rental App Development : Cost and Process” [Електронний ресурс] – Режим доступу до ресурсу: [https://www.aalpha.net/blog/bike-rental-app-development/]