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

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка Full-stack застосунку для обміну туристичним досвідом на основі бази відгуків про місця інтересу»

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

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

_____ Богдан ВЛАСЕНКО
(підпис)

Виконав: здобувач вищої освіти групи ПД-41
Богдан ВЛАСЕНКО

Керівник: Юрій ЗАДОНЦЕВ
к. т. н.

Рецензент:

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ

ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

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

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

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Власенку Богдану Євгенійовичу

1. Тема кваліфікаційної роботи: «Розробка Full-stack застосунку для обміну туристичним досвідом на основі бази відгуків про місця інтересу»
керівник кваліфікаційної роботи к.т.н., доцент кафедри ІПЗ Юрій ЗАДОНЦЕВ,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «24» лютого 2025 р. № 56.

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

3. Вихідні дані до кваліфікаційної роботи: аналітичні матеріали щодо архітектури web-застосунків, документація технологій React, TypeScript, Node.js, Express та MongoDB, стандарти реалізації REST API та WebSocket, технічні описи сервісів автентифікації з використанням JWT, принципи інтеграції з картографічними API (Mapbox), вимоги OWASP щодо безпеки вебсистем, приклади реалізації клієнт-серверних систем для обміну користувацьким контентом у сфері туризму.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Аналіз існуючих засобів та постановка задачі
2. Проєктування та реалізація програмного забезпечення
3. Тестування та перспективи розвитку програми

5. Перелік графічного матеріалу: презентація
1. Аналіз аналогів

2. Вимоги до програмного забезпечення
3. Програмні засоби реалізації
4. Діаграма варіантів використання
5. Архітектура програмної системи
6.. Вибір технологій та інструментів реалізації
7-8. Екранні форми
9. Апробація результатів дослідження

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

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	25.02.-04.03.2025	
2	Аналіз та дослідження існуючих аналогів	05.03-13.03.2025	
3	Огляд сучасних технологій веброзробки та архітектурних підходів до створення Full-stack застосунків із синхронізацією в реальному часі	14.03 -24.03.2025	
4	Проєктування програмної архітектури застосунку для обміну туристичним досвідом з інтерактивною картою та захищеною автентифікацією	25.03-05.04.2025	
5	Програмна реалізація клієнтської та серверної частин застосунку з використанням React, Node.js, MongoDB та WebSocket	06.04-15.04.2025	
6	Тестування функціональності, безпеки та інтерактивної карти в умовах різного навантаження	16.04-28.04.2025	
7	Оформлення роботи: вступ, висновки, реферат	29.04-11.05.2025	
8	Розробка демонстраційних матеріалів	12.05-18.05.2025	
9	Попередній захист роботи	19.05-30.05.2025	

Здобувач вищої освіти _____ Богдан ВЛАСЕНКО
(підпис)

Керівник
кваліфікаційної роботи _____ Юрій ЗАДОНЦЕВ
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 62 стор., 3 табл., 13 рис., 50 джерел.

Мета роботи: допомога та спрощення обміну туристичним досвідом на базі web-застосунку.

Об'єкт дослідження: процес обміну інформацією між користувачами туристичних платформ.

Предмет дослідження: веб-застосунки з реалізацією обміну туристичним досвідом.

Короткий зміст роботи: у роботі проведено аналіз сучасних web-платформ для обміну туристичним досвідом (TripAdvisor, Google Maps, Maps.me), виявлено їхні недоліки й сформульовано вимоги до нової системи. Розроблено архітектуру клієнт-серверного застосунку на основі технологій React, TypeScript, Node.js, Express, MongoDB, Mapbox і socket.io. Реалізовано функціонал реєстрації, авторизації через JWT, створення, редагування та перегляду маркерів на інтерактивній карті, обмін повідомленнями в реальному часі. Забезпечено захист персональних даних, обробку помилок, фільтрацію контенту та дотримання рекомендацій OWASP. Проведено тестування функціональності та розроблено рекомендації щодо масштабування системи.

КЛЮЧОВІ СЛОВА: FULL-STACK, WEB-ЗАСТОСУНОК, REACT, NODE.JS, MONGODB, MAPBOX, JWT, REST API, WEBSOCKET

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API – програмний інтерфейс взаємодії між клієнтом і сервером, зокрема REST API.

JWT (JSON Web Token) – стандарт для передачі інформації між сторонами у вигляді зашифрованого токена.

UI (User Interface) – інтерфейс користувача, що забезпечує взаємодію з веб-застосунком.

UX (User Experience) – досвід користувача під час роботи із застосунком.

HTTP – протокол передачі гіпертексту, що використовується у REST API.

HTTPS – захищений варіант HTTP із шифруванням з’єднання.

CRUD – базові операції з даними: створення (Create), читання (Read), оновлення (Update), видалення (Delete).

MongoDB – документо-орієнтована база даних NoSQL, використана в проєкті.

ODM (Object Document Mapping) – спосіб взаємодії з документною БД через об’єктні моделі.

Express – фреймворк для побудови серверної частини на Node.js.

React – бібліотека JavaScript для побудови інтерфейсу користувача.

WebSocket – протокол, що забезпечує двосторонню комунікацію між клієнтом і сервером у реальному часі.

Socket.IO – бібліотека для реалізації WebSocket у Node.js і React.

TypeScript – надмножина JavaScript, що забезпечує статичну типізацію.

Mapbox GL JS – бібліотека для інтерактивної роботи з картами в браузері.

XSS (Cross-Site Scripting) – тип атаки, пов'язаний із виконанням сторонніх скриптів у браузері жертви.

2FA (Two-Factor Authentication) – двофакторна автентифікація.

Rate-limiting – обмеження кількості запитів для запобігання зловживанню ресурсами.

Axios – клієнтська бібліотека для надсилання HTTP-запитів.

Vite – сучасний інструмент для швидкої збірки фронтенду.

Mongoose – бібліотека ODM для роботи з MongoDB у Node.js.

ВСТУП

У сучасну епоху цифрової трансформації особливого значення набувають програмні рішення, орієнтовані на спільноти користувачів та інтерактивний обмін досвідом. Однією з таких сфер є туризм — галузь, що не лише активно розвивається, а й демонструє високий рівень взаємодії з цифровими технологіями, зокрема з web-застосунками, які дозволяють мандрівникам ділитися враженнями, рекомендаціями та фото з відвіданих місць. Поява подібних сервісів стала можливою завдяки розвитку Full-stack архітектур, які поєднують фронтенд- і бекенд-технології для забезпечення повного циклу взаємодії користувача з інформаційною системою.

Актуальність теми дослідження зумовлена декількома ключовими чинниками. По-перше, зростаюча кількість користувачів, які прагнуть отримати персоналізовану інформацію про туристичні локації, актуалізує потребу у створенні надійних та зручних платформ, орієнтованих на обмін досвідом. По-друге, сучасні web-застосунки повинні відповідати підвищеним вимогам до безпеки даних, зокрема при зберіганні персональної інформації та реалізації автентифікації. По-третє, необхідність інтерактивної візуалізації (зокрема інтеграція з картографічними сервісами, як-от Mapbox) вимагає розробки гнучкої та масштабованої архітектури, здатної підтримувати обмін інформацією в реальному часі. І нарешті, широке використання сучасного технологічного стеку — таких як TypeScript, React, Express і MongoDB — створює сприятливі умови для реалізації продуктивних і адаптивних рішень, що відповідають викликам сьогодення у сфері інформаційних систем.

Метою дослідження є розробка функціонального та безпечного Full-stack web-застосунку для обміну туристичним досвідом, який передбачає зберігання, обробку та візуалізацію відгуків про місця інтересу, забезпечуючи інтерактивну взаємодію з користувачами в реальному часі.

Досягнення поставленої мети передбачає вирішення таких **завдань**:

1. Провести аналіз сучасних web-застосунків, що реалізують функціонал обміну туристичними враженнями, і виявити їх ключові особливості та недоліки.
2. Визначити потреби кінцевих користувачів і сформулювати функціональні та нефункціональні вимоги до системи.
3. Обґрунтувати вибір інструментів і технологій для реалізації клієнтської та серверної частин застосунку.
4. Розробити архітектуру програмного забезпечення, що поєднує REST API та WebSocket для підтримки синхронізації даних.
5. Реалізувати механізми автентифікації та захисту даних користувачів з використанням JWT та алгоритмів хешування.
6. Побудувати інтерфейс користувача з інтерактивною картою на основі Mapbox, забезпечивши можливість додавання, редагування та перегляду маркерів.
7. Провести тестування застосунку, виявити потенційні помилки та оцінити стійкість до типових вразливостей.
8. Розробити пропозиції щодо вдосконалення та масштабування системи, враховуючи перспективи інтеграції додаткового функціоналу.

Об’єктом дослідження є процес обміну туристичним досвідом у цифровому середовищі за допомогою web-застосунків.

Предметом дослідження виступають архітектурні, функціональні та безпекові аспекти розробки Full-stack застосунку з базою даних відгуків і інтерактивною картою на основі сучасного технологічного стеку (TypeScript, React, Express, MongoDB).

Джерельна база дослідження охоплює офіційну документацію розробницьких платформ (React, Express, MongoDB), статті та аналітичні огляди з безпеки web-застосунків (зокрема OWASP Top 10), результати тестування подібних систем, а також наукові публікації з питань архітектури інформаційних систем, реалізації REST API, роботи з WebSocket і впровадження протоколів авторизації.

Фактичний матеріал дослідження сформовано на основі самостійної розробки експериментального web-застосунку, що включає клієнтську частину на

React із використанням Mapbox, серверну логіку на Express, підключення до MongoDB через Mongoose, а також реалізацію двосторонньої синхронізації через socket.io. До складу системи увійшли механізми автентифікації користувачів, маршрутизація, додавання та редагування маркерів на карті, а також обмін даними в реальному часі між кількома клієнтами. Окрему увагу приділено реалізації обробки помилок, захисту від дублювання записів і обмеженню доступу до ресурсів залежно від ролі користувача.

Методи дослідження, застосовані в роботі, включають:

- **Аналіз і узагальнення** — для вивчення аналогічних web-застосунків та існуючих рішень у сфері туристичних платформ.
- **Моделювання архітектури** — для побудови логіки клієнт-серверної взаємодії та обґрунтування розподілу функцій між компонентами системи.
- **Проєктування і реалізація** — для створення прототипу програмного забезпечення із подальшим програмуванням.
- **Емпіричне тестування** — для перевірки функціонування реалізованих модулів у середовищі Vite та Node.js, а також перевірки обробки HTTP-запитів і WebSocket-підключень.
- **Метод граничних умов** та **статичний аналіз коду** — для виявлення логічних помилок та оцінки захищеності застосунку.

Наукова новизна роботи полягає в такому:

- **Вперше визначено** комплекс вимог до захищених web-застосунків для обміну туристичним досвідом із підтримкою синхронного обміну даними на основі WebSocket.
- **Вдосконалено** механізм автентифікації через поєднання JWT та ролей користувачів у контексті туристичних платформ, що дозволяє підвищити рівень безпеки доступу до інформації.
- **Дістало подальший розвиток** використання типізованого середовища TypeScript для реалізації frontend-частини із збереженням повної інтеграції з REST API та сокетами.

- **Створено нову концепцію**, що узагальнює досвід роботи з інтерактивними картами на базі Mapbox та функціоналом реального часу (real-time sync) для розробки платформ соціального обміну.
- **Розроблено нову систему**, яка демонструє гнучке поєднання модульного дизайну React-компонентів, типізованого контролю доступу на сервері та обробки запитів до MongoDB через Mongoose.
- **З використанням відомого принципу MVC** (Model-View-Controller) реалізовано розділення логіки клієнта, сервера й бази даних, що дозволило забезпечити масштабованість і зрозумілу підтримку коду.

Теоретична цінність дослідження полягає в узагальненні архітектурних підходів до побудови сучасних web-застосунків із синхронною взаємодією клієнтів, а також у розробці структурованої моделі для захищеної обробки користувацьких даних у системах з геолокаційною візуалізацією. Запропонована модель може бути використана для створення подібних програмних продуктів у галузях соціальних мереж, освітніх сервісів, урбаністичних платформ або систем внутрішнього моніторингу територій.

У підсумку, актуальність, мета, глибина технічної реалізації та рівень практичної апробації обраної теми підтверджують доцільність і значущість дослідження. Отримані результати можуть стати підґрунтям для подальших наукових розвідок у сфері розробки масштабованих, безпечних і продуктивних Full-stack застосунків, а також для впровадження аналогічних систем у практичну діяльність туристичних організацій, інформаційних сервісів і платформ соціальної взаємодії.

1 АНАЛІЗ ІСНУЮЧИХ ЗАСОБІВ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Аналіз сучасних web-застосунків для обміну туристичним досвідом

Платформи TripAdvisor, Google Maps і Maps.me становлять приклади реалізації різних підходів до організації цифрової взаємодії у сфері туризму. Їхнє порівняння дозволяє виявити сильні й слабкі сторони кожного з рішень, а також обґрунтувати вибір архітектурних рішень, функціонального наповнення та моделі взаємодії користувачів при розробці нового застосунку. TripAdvisor є централізованою системою управління користувацьким контентом, що забезпечує накопичення великої кількості відгуків, рейтингів і фотографій, пов’язаних із готелями, ресторанами та туристичними об’єктами. Основу функціоналу становлять модулі пошуку, фільтрації, ранжування, а також інтерактивне сортування за категоріями, які враховують тип закладу, його розташування, середню вартість і рівень обслуговування. До переваг системи належать значна кількість відгуків, що забезпечує репрезентативність, і багаторівнева система оцінювання, яка дозволяє розглядати об’єкти в розрізі різних характеристик. Утім, механізми модерації й сортування в TripAdvisor не завжди є прозорими: алгоритми рекомендування можуть ґрунтуватися як на фактичній поведінці користувачів, так і на спонсорському просуванні. Крім того, відсутність інтеграції з картографічними модулями високого рівня обмежує можливість просторового аналізу туристичних маршрутів або зручної побудови геоприв’язаних сценаріїв. Для отримання додаткової інформації користувач часто змушений перемикатися між зовнішніми сервісами, що погіршує когерентність взаємодії.

Google Maps реалізує концепцію геопросторової багатошарової платформи, яка поєднує навігацію, рекомендації, рецензії, відгуки й динамічні дані про трафік, розклади транспорту і часові витрати. Цей сервіс є частиною екосистеми Google, що забезпечує йому доступ до аналітики пошукової поведінки, геолокаційної історії, активності в інших додатках (наприклад, Gmail, YouTube), що дозволяє

формувати персоналізовані рекомендації. Завдяки тісній інтеграції з Google My Business, користувач має доступ до офіційних описів об’єктів, контактної інформації, графіків роботи та актуальних фотографій. У системі реалізовано багаторівневу фільтрацію, підтримку категорій POI, формування маршрутів із кількома точками та можливість прокладання доріг із урахуванням реального трафіку. Проте надлишок інформації, велика кількість візуальних елементів, рекламних блоків та автоматичних підказок іноді створюють когнітивне навантаження на користувача. Крім того, оцінки й відгуки не проходять чіткої модерації або фільтрації за достовірністю, що призводить до наявності контенту сумнівної якості. Для сторонніх розробників, які використовують Google Maps API у власних проєктах, діє комерційна модель із платою за кількість запитів, що обмежує доступність ресурсу для некомерційних або студентських ініціатив.

На відміну від попередніх сервісів, Maps.me орієнтований на автономне використання з акцентом на офлайн-доступ до географічних даних. Платформа базується на картографічній основі OpenStreetMap, що надає можливість редагування карт спільнотою, а також забезпечує високий рівень деталізації для окремих регіонів. Основна функціональність зосереджена навколо пошуку POI, планування маршрутів (пішохідних, велосипедних, автомобільних), додавання закладок і створення власних колекцій. Система оптимізована для роботи в умовах обмеженого або відсутнього інтернет-з’єднання, що є важливою перевагою для мандрівників у сільських або гірських районах. Разом із тим, обмежена інтеграція з соціальними функціями — відсутність системи коментарів, рейтингової моделі, а також слабка персоналізація контенту — знижують ефективність застосунку як платформи для обміну досвідом. Система не підтримує двосторонню взаємодію між користувачами або функції рекомендацій, що значно зменшує динамічність вмісту. Крім того, складність імпорту або експорту маршрутів між платформами, а також слабка підтримка API, обмежують можливості використання Maps.me у сторонніх системах.

Порівняння вказаних сервісів засвідчує, що попри широке функціональне покриття, жоден із них не поєднує одночасно автономність, гнучку кастомізацію, зручну синхронізацію даних, захищену автентифікацію та взаємодію в режимі реального часу. Водночас у всіх системах простежується тенденція до централізованого керування даними, відсутність інструментів для побудови персоналізованої карти досвіду, а також обмежена можливість створення контенту з локальним контекстом (наприклад, додавання нетипових локацій, які ще не мають масових відгуків). Це відкриває простір для розробки застосунку, в якому реалізовано незалежне редагування маркерів, взаємодію між користувачами через WebSocket, адаптивний дизайн на базі React та використання бази даних MongoDB для гнучкого зберігання напівструктурованої інформації. Інтеграція таких функцій дозволяє створити систему нового типу — відкриту, гнучку, інтерактивну, з акцентом на актуальність, просторову точність та безпеку персональних даних.

Порівняльний аналіз функціональності, користувацького досвіду (UX) та архітектури платформ TripAdvisor, Google Maps і Maps.me дає змогу ідентифікувати ключові відмінності у структурній організації, логіці взаємодії з користувачем та технічній реалізації систем, що виконують подібні завдання у сфері туристичної інформатизації. З функціонального погляду, TripAdvisor є платформою з орієнтацією на соціальну взаємодію, що надає користувачам можливість залишати текстові відгуки, оцінки за категоріями (на кшталт «чистота», «сервіс», «розташування»), публікувати фотографії, формувати добірки улюблених місць, а також вести власний профіль. Основним функціональним вузлом виступає система рейтингу, яка ґрунтується на зібраних рецензіях, і формує списки рекомендованих об’єктів за регіонами, категоріями або стилем подорожі. TripAdvisor також має систему бронювання та прямі посилання на партнерські сервіси, що розширює функціональність, але одночасно ускладнює навігацію. Google Maps реалізує ширшу функціональність, де окрім стандартного пошуку POI, можна прокладати маршрути з урахуванням реального часу, трафіку та громадського транспорту, переглядати 360-градусні панорами (Street View), додавати власні фото та залишати

короткі оцінки. Інтеграція з іншими сервісами Google забезпечує високу ефективність у доступі до контактної інформації, графіків роботи, посилань на сайти та навіть динаміки популярності локацій. Сервіс також підтримує роботу в офлайн-режимі, хоча лише після попереднього завантаження карти. Maps.me демонструє найбільш вузьку функціональність, сфокусовану на навігації, локальному зберіганні карт і прокладанні маршрутів без інтернету. Доступ до інформації про заклади, пам’ятки та інші об’єкти здійснюється переважно через OpenStreetMap, а відгуки реалізовані лише частково — без підтримки коментарів, рейтингової системи або механізмів спільного обговорення. При цьому платформа дозволяє створювати закладки та ділитися геопозиціями, проте відсутні механізми персоналізації та історії взаємодії з контентом. Усі ці відмінності функціональності напряму впливають на користувацький досвід, що в кожній з платформ сформовано відповідно до домінуючої мети системи.

UX-аналіз демонструє, що Google Maps забезпечує найбільш адаптивний інтерфейс з гнучким перемиканням між мобільною та десктопною версіями, широкою підтримкою мов, високим рівнем інтеграції з іншими застосунками та логічною структурою розміщення елементів керування. Водночас надлишок функціональних компонентів (транспортні шари, вкладки «поруч», рекламні вставки, Smart Suggestions) призводить до когнітивного перевантаження користувача, особливо в умовах слабкого з’єднання або на мобільних пристроях зі старим апаратним забезпеченням. TripAdvisor застосовує модель класичного каталогу з вертикальним скролінгом і значною кількістю текстового контенту. Основний акцент зроблено на відгуках і коментарях, які часто мають довгий обсяг і вимагають часу на прочитання. Користувачі нерідко стикаються з проблемами навігації між окремими вкладками (відгуки, фото, послуги, мапа), особливо в мобільній версії, де відсутня контекстна панель. Ієрархія інтерфейсу є негнучкою, адаптивність обмежена, а загальна візуальна структура виглядає перевантаженою при перегляді великої кількості елементів на одній сторінці. У Maps.me UX побудовано на мінімалізмі — основний екран містить карту, декілька

функціональних кнопок (пошук, маршрут, закладки), а решта опцій винесені в бічне меню. Це спрощує навігацію, але обмежує гнучкість: користувач не має змоги персоналізувати відображення контенту, застосовувати фільтри або візуально налаштовувати структуру карти. UX-рішення в Maps.me оптимізовано для офлайн-використання, однак недоступність розширених режимів фільтрації або відображення додаткових даних (рейтингів, тегів, коментарів) знижує ефективність при пошуку нових, незнайомих місць. Усі три сервіси мають принципово різні UX-підходи: Google Maps — інструмент професійного рівня з високою інтерактивністю, TripAdvisor — контентно-центричний застосунок для аналітичного ознайомлення з відгуками, Maps.me — простий навігатор для автономного використання без зайвих функцій.

З архітектурної точки зору, Google Maps реалізовано як високонавантажену мікросервісну систему з великою кількістю зовнішніх інтеграцій, доступом до хмарної інфраструктури Google Cloud Platform, підтримкою кешування, потокової обробки геолокаційних даних, аналітики користувацької поведінки, гнучкої побудови маршрутів і високої доступності. Система підтримує масштабування в реальному часі, швидке реагування на запити та паралельну обробку великої кількості подій. TripAdvisor має архітектуру, що наближена до централізованої клієнт-серверної моделі з використанням RESTful API, системою автентифікації користувачів, розділенням прав доступу та базами даних, які оптимізовані під контентний пошук (наприклад, Elasticsearch). Система активно застосовує механізми кешування запитів, lazy loading і пошарову побудову контенту, що дозволяє зменшити час завантаження сторінки. Проте обмежена інтерактивність (відсутність обробки подій у реальному часі, як у WebSocket) знижує реактивність інтерфейсу. Maps.me реалізує архітектуру, побудовану на офлайн-картографії з передзавантаженням даних і локальною обробкою маршрутів. У додатку використовується монолітна структура з інтегрованим модулем рендерингу карт і окремими підсистемами для пошуку POI, маршрутизації та обробки запитів користувача. Відсутність динамічного підключення до серверної частини зумовлює

низький рівень інтерактивності, але забезпечує стабільність і швидкість на низько-потужному обладнанні. Кожна з архітектур має свої переваги: мікросервісна модель Google Maps дозволяє створювати масштабовані розподілені системи з інтеграцією аналітики та AI, централізована модель TripAdvisor спрощує керування контентом і доступами, монолітна офлайн-архітектура Maps.me гарантує автономність та мінімальну залежність від інфраструктури. Проведене порівняння дозволяє сформулювати підхід до побудови власного застосунку, який об’єднає елементи масштабованої клієнт-серверної архітектури з підтримкою WebSocket для синхронізації даних у реальному часі, використанням NoSQL-бази даних (MongoDB) для зберігання напівструктурованої інформації, та реалізацією адаптивного фронтенду з React і TypeScript з інтеграцією картографічного API (Mapbox) та інтерфейсами REST. Такий підхід забезпечить баланс між продуктивністю, інтерактивністю, безпекою і можливістю масштабування.

Таблиця 1.1

Аналіз сучасних web-застосунків для обміну туристичним досвідом

Платформа	**Основне призначення**	**Функціональні можливості**	**UX / Інтерфейсні особливості**	**Архітектура / Технічна база**	**Основні переваги**	**Основні недоліки**
TripAdvisor	Відгуки, рейтинги та бронювання туристичних об'єктів	Рецензії, фото, оцінки, пошук, фільтрація, персональні списки, пряме бронювання	Каталожний інтерфейс, орієнтований на текст; мобільна версія менш оптимізована	Централізована архітектура з REST API; пошукові бази типу Elasticsearch	Велика база користувацького контенту, багаторівнева оцінка, підтримка бронювання	Обмежена інтерактивність; наявність рекламних пріоритетів; перевантаженість UI
Google Maps	Геонавігація та рекомендації місць інтересу	Прокладання маршрутів, відгуки, фото, Street View, популярність, інтеграція з іншими сервісами Google	Інтерактивний багатошаровий інтерфейс, адаптивність, персоналізація	Мікросервісна архітектура; GCP; масштабованість; кешування; AI-аналітика	Потужна навігація, зручна інтеграція, офлайн-режим, персоналізовані поради	Надмірна складність; реклама; платна модель API при великому навантаженні
Maps.me	Офлайн-навігація для туристів	Завантаження карт, пошук POI, прокладання маршрутів, закладки	Мінімалістичний дизайн, простота, оптимізовано під офлайн	Монолітна офлайн-архітектура на OpenStreetMap; локальне збереження даних	Повноцінна робота без інтернету, простота, висока швидкість	Відсутність коментарів, рейтингів, слабка персоналізація, відсутність API

1.2 Визначення потреб користувачів та постановка проблеми

Цільова аудиторія цифрових рішень у сфері туризму формується на перетині декількох сегментів користувачів, об’єднаних спільною потребою — доступ до актуальної, локалізованої, достовірної та персоналізованої інформації про місця інтересу. Основну частину аудиторії становлять незалежні мандрівники віком від 18 до 45 років, які активно користуються мобільними пристроями, орієнтуються на самостійне планування подорожей, відкриті до нових вражень і покладаються на цифрові платформи як основне джерело туристичної інформації. Ця категорія охоплює як внутрішніх туристів, що досліджують свою країну, так і міжнародних мандрівників, для яких важливою є можливість швидко отримати інформацію про об’єкти в незнайомому середовищі, не витрачаючи ресурси на довготривале порівняння. Аудиторія характеризується високою цифровою грамотністю, орієнтацією на зручність та ефективність, очікує інтерфейсу, який не потребує додаткових пояснень або навчання.

Другу групу становлять так звані «локальні дослідники» — користувачі, які використовують платформу не стільки для подорожей, скільки для планування вільного часу у своєму регіоні: пошуку кав’ярень, пам’яток, маршрутів вихідного дня. Цей сегмент зазвичай очікує швидкого доступу до мапи з позначками, рекомендацій, фільтрів за відгуками друзів або спільноти. Значну частку аудиторії також становлять фахівці суміжних галузей — гіди, працівники туризму, культурні менеджери, які можуть використовувати платформу як інструмент представлення певного простору або залучення туристичних потоків у конкретні локації. В окремий сегмент варто виділити користувачів із обмеженими технічними можливостями або повільним інтернетом — для них критично важливою є підтримка офлайн-режиму, економія трафіку та збереження ключового функціоналу навіть за мінімальної пропускної здатності мережі.

Очікування цільової аудиторії щодо функціонування системи зосереджуються навколо трьох головних векторів: інформативність, інтерфейсна ефективність і функціональна довершеність. З погляду інформативності,

користувачі вимагають від системи відображення не лише загальної інформації про об’єкт, а й відгуків, фотографій, рейтингів, актуальності даних, наявності локального контексту — наприклад, вказівок, як дістатися, годин роботи, сезонності, особливостей маршруту. Цінним також вважається розширення опису через коментарі інших користувачів, рекомендації місцевих мешканців, інтерактивні мітки, що додаються в реальному часі. З точки зору інтерфейсної ефективності, користувачі очікують інтуїтивного розміщення елементів, швидкої навігації, відсутності надмірної кількості підрозділів або рівнів вкладеності. Адаптивність до різних типів екранів, можливість взаємодії однією рукою, підтримка жестів, мінімалізм у візуальному представленні — усе це є важливими аспектами зручності.

У межах функціональної довершеності користувачі очікують безпечної автентифікації (через email або соціальні мережі), наявності системи закладок, можливості створення власних маршрутів, додавання нотаток або міток на карті, обміну враженнями з іншими користувачами без прив’язки до соціальних платформ. Окремо слід виділити вимогу до швидкого оновлення контенту в реальному часі, синхронізації між пристроями, підтримки персоналізованої стрічки або рекомендацій. Очікування також включають збереження даних для офлайн-доступу, опції пошуку за фільтрами (відстань, тип локації, середній рейтинг), а також функціонал зворотного зв’язку — можливість повідомити про помилки, неточності, дезінформацію. Для частини аудиторії критичним є питання безпеки — наявність політик захисту персональних даних, можливість контролю над тим, які дані публікуються, які залишаються приватними, хто має доступ до інформації, що зберігається у профілі.

У комплексі ці очікування формують вимоги до системи, здатної забезпечити рівновагу між гнучкістю, продуктивністю, простотою використання та безпекою. Врахування цих чинників є необхідною передумовою для створення релевантного застосунку, що відповідає реальним запитам і сценаріям користування в межах туристичного середовища.

Існуючі web-застосунки для туристичних потреб, незважаючи на значну популярність і широке охоплення функціоналу, мають низку типових проблем, які суттєво обмежують ефективність їх використання в реальних умовах.

Однією з найбільш поширених є надмірна централізація управління контентом, коли ключові елементи (відгуки, рейтинги, рекомендації) або контролюються адміністраторами системи, або формуються на основі алгоритмів, які не є прозорими для користувачів. У результаті формується ситуація, коли нові або некомерційні об'єкти практично не мають шансу потрапити до поля зору мандрівників, незалежно від їхньої фактичної цінності. Алгоритми сортування, як правило, надають перевагу об’єктам із високою кількістю переглядів, активністю в коментарях або спонсорською підтримкою, що створює викривлену картину об’єктивності.

Другою суттєвою проблемою є слабка підтримка локального контенту. У багатьох випадках платформи зосереджуються на популярних туристичних напрямках, тоді як інформація про маловідомі локації залишається неповною, застарілою або відсутньою. Це особливо актуально для внутрішнього туризму або подорожей у менш урбанізовані регіони, де користувачі не можуть знайти маршрутів, відгуків, попереджень чи порад, що ускладнює планування поїздок. Крім того, більшість систем не дозволяє додавати нові об’єкти без попередньої модерації або затвердження, що уповільнює оновлення бази та знижує оперативність контенту.

До технічних проблем слід віднести обмеженість роботи в умовах нестабільного інтернет-з’єднання. Лише окремі сервіси пропонують офлайн-функціонал, і той є обмеженим — користувачі часто не мають доступу до фотографій, відгуків, оновлень карти або точок інтересу без попереднього завантаження великих обсягів даних. Також спостерігається слабка інтеграція між мобільною та десктопною версіями, що ускладнює перехід між пристроями, порушує логіку навігації та веде до дублювання дій.

Ще одним обмеженням є відсутність повноцінної взаємодії між користувачами в режимі реального часу. Більшість застосунків передбачає одностороннє додавання контенту (відгуків, оцінок), але не надає можливості обговорення, уточнення або колективного створення маршрутів. Це значно знижує динаміку системи та не дозволяє формувати спільноти, що могли б підтримувати локальний туризм через обмін досвідом у реальному часі.

У контексті UX-архітектури слід зазначити перевантаженість інтерфейсів, надмірну кількість реклами та інформаційного шуму, що перешкоджає швидкому доступу до потрібних функцій. Багато користувачів змушені витрачати час на навігацію серед великої кількості вкладок, банерів, рекламних блоків і опцій, що дублюють одна одну або малозрозумілі з першого погляду. Існуючі застосунки також демонструють обмеження у використанні інструментів персоналізації: рідко реалізується можливість фільтрування контенту за власними вподобаннями, приховування непотрібної інформації, створення власного інтерфейсу перегляду мапи або вибір рівня деталізації. Окремою проблемою виступає недостатня гнучкість збереження та відновлення даних — наприклад, неможливість швидко відновити збережені маршрути після зміни пристрою або втрати доступу до облікового запису.

Перелічені проблеми обґрунтовують потребу у створенні нового підходу, який базується на принципах відкритості, інтерактивності, автономності та адаптивності. Такий підхід передбачає, з одного боку, залучення користувачів до генерації контенту в реальному часі — через додавання маркерів, коментарів, уточнень, сповіщень без попередньої модерації або затримки публікації, а з іншого — забезпечення технічної основи для гнучкого обміну даними через WebSocket, підтримки офлайн-функціоналу, використання легко масштабованої NoSQL-бази (наприклад, MongoDB) для зберігання напівструктурованої інформації.

Новий застосунок має враховувати сценарії використання в умовах низької пропускної здатності, мінімізації енергоспоживання, а також забезпечення приватності даних — через контроль над доступом до геолокації, публікаціями та

автентифікацією. Усе це вимагає формування нової концепції взаємодії, де кожен користувач виступає не лише споживачем, а й активним учасником формування туристичного ландшафту. Вимоги до цієї концепції продемонстровані на рисунку 1.1.

Рис. 1.1 Визначення потреб користувачів та постановка проблеми

1.3 Формулювання цілей та функціональних вимог до системи

Функціональні вимоги до web-застосунку для обміну туристичним досвідом визначаються на основі реальних сценаріїв користування, очікувань цільової

аудиторії та аналізу недоліків існуючих платформ. Основною функцією системи є надання користувачам можливості швидко та зручно обмінюватися інформацією про місця інтересу, включаючи текстові відгуки, оцінки, координати, фото та рекомендації щодо маршрутів. Застосунок повинен реалізовувати функціонал реєстрації та автентифікації користувача з підтримкою JWT-токенів, а також забезпечувати базову рольову модель, яка розрізняє щонайменше два типи користувачів: звичайний користувач (User) та адміністратор (Admin або Moderator). Для кожної ролі мають бути визначені дозволені дії: створення, перегляд, редагування або видалення маркерів, доступ до чужих відгуків, участь в обговореннях, управління вмістом.

Користувач повинен мати змогу додавати маркер на карту, вказуючи назву локації, її координати, короткий опис, одну або кілька фотографій, категорію (природа, історія, гастрономія, інше), рейтинг за шкалою, а також додаткові опції — наприклад, складність маршруту, сезонність або доступність. Усі маркери мають зберігатися в базі даних із можливістю подальшої фільтрації та пошуку за ключовими полями. Обов’язковою є підтримка CRUD-операцій (створення, читання, оновлення, видалення) для кожного типу контенту, який додає користувач: це стосується маркерів, коментарів, маршрутів та особистих добірок. Додатково має бути реалізовано пошук за назвою, тегами, категоріями та геолокацією, а також фільтрація за популярністю, новизною або рейтингом.

Ключовим компонентом функціоналу є інтерактивна карта, яка дозволяє переглядати всі створені маркери, здійснювати навігацію по території, масштабувати, фокусуватись на певних регіонах, отримувати спливаючі підказки при наведенні або кліку. Важливо, щоб карта підтримувала відображення у вигляді різних шарів (базова карта, супутникова, топографічна), а також дозволяла перемикатися між категоріями об’єктів. Інтерфейс карти повинен бути інтегрований з компонентами перегляду детальної інформації про маркер (відгуки, оцінки, фото, автор), а також надавати змогу переходити до розділів редагування або створення маршрутів. Передбачено, що система має дозволяти створення та

збереження персональних маршрутів — із можливістю додавання кількох точок, розрахунку відстані, збереження у профілі користувача, подальшого експорту або публікації на платформі. Крім цього, має бути реалізовано модуль комунікації, який дозволяє користувачам залишати коментарі під відгуками інших користувачів, відповідати на них, а також ставити вподобання (upvote/downvote) — для виявлення найбільш релевантного вмісту.

У разі активної спільноти необхідним є функціонал сповіщень — про відповіді, згадки, додавання об’єктів у близькому радіусі. Функціональні вимоги також охоплюють модуль профілю: кожен користувач повинен мати доступ до особистого кабінету, де зберігаються його активність, створені маркери, маршрути, обране, а також налаштування приватності. Повинна бути передбачена можливість видалення профілю, зміни аватару, перегляду статистики (кількість доданих точок, отриманих вподобань, тощо).

У межах функціональних вимог має бути реалізовано механізм обробки помилок та валідації введених даних. Усі форми (створення маркерів, написання відгуків, оновлення профілю) мають бути захищені від некоректного введення, включати обмеження на довжину полів, тип файлів, допустимі значення. Система повинна забезпечувати виведення інформативних повідомлень у разі невдачі — з кодом помилки, її описом та можливими рішеннями. Для підвищення безпеки доцільно реалізувати обмеження на кількість запитів із одного IP-адресу, підтвердження реєстрації через email, підтримку багатофакторної автентифікації.

Додатково має бути реалізована підтримка багатомовності інтерфейсу (на базі i18n), адаптивність до мобільних пристроїв, а також кросбраузерна сумісність. Система має масштабуватись для одночасної роботи великої кількості користувачів, із підтримкою кешування та оптимізованого завантаження компонентів. Також необхідно врахувати, що усі функціональні модулі повинні взаємодіяти з backend-частиною за допомогою REST API або WebSocket (для обміну в реальному часі), а дані зберігатися в структурованому форматі з підтримкою гнучкого запиту через

MongoDB. Усі вимоги повинні реалізовуватися з урахуванням розширюваності та подальшого розвитку функціоналу.

Нефункціональні вимоги до системи обміну туристичним досвідом визначають загальні характеристики програмного забезпечення, які не залежать безпосередньо від його функціональних можливостей, але є критичними для стабільної, безпечної та зручної експлуатації. У сфері туристичних web-застосунків особливу увагу необхідно приділяти масштабованості, безпеці та доступності, оскільки ці параметри впливають на можливість обробки великої кількості користувачів, збереження довіри до платформи та її здатність працювати в різноманітних технічних умовах.

Масштабованість передбачає, що застосунок повинен функціонувати без зниження продуктивності при зростанні навантаження, включаючи одночасні підключення користувачів, обробку запитів, зберігання великої кількості даних та оновлення карти в режимі реального часу. Для цього архітектура системи повинна бути модульною, з чітким розмежуванням між клієнтською та серверною частинами, підтримувати горизонтальне масштабування бази даних MongoDB, розподілення навантаження між сервісами, кешування часто використовуваних запитів і використання CDN для статичних ресурсів. Важливим є застосування асинхронної обробки даних і WebSocket-підключень для забезпечення ефективної взаємодії в реальному часі. У випадку зростання трафіку система повинна дозволяти додавання нових екземплярів серверів без зміни логіки взаємодії між компонентами.

Безпека є ключовим нефункціональним аспектом, оскільки застосунок оперує чутливою інформацією користувача: геолокацією, обліковими записами, електронною поштою, активністю. Передбачено використання шифрування всіх HTTP-з’єднань через HTTPS, захист паролів шляхом хешування з використанням bcrypt із динамічною генерацією солі, авторизацію через JWT-токени з контрольованим терміном дії, а також обмеження доступу до ресурсів на основі ролей користувачів. Серверна частина повинна містити механізми перевірки прав

доступу до кожного запиту (middleware), систему виявлення підозрілих дій (часті запити, автоматизовані атаки), логування подій та оповіщення про помилки без розголошення внутрішньої структури сервера. Користувачі повинні мати змогу контролювати приватність свого профілю, обираючи, які дані є публічними, а які — приватними. Особливу увагу варто приділити захисту від основних вразливостей OWASP (ін’єкції, XSS, CSRF), реалізувати механізми перевірки вхідних даних, очищення запитів, обмеження доступу до внутрішніх API. Крім того, застосунок має відповідати вимогам GDPR або аналогічних регламентів щодо обробки персональних даних.

Доступність застосунку охоплює як технічний аспект (гарантований аптайм, підтримка різних типів пристроїв і мереж), так і когнітивний (легкість сприйняття інтерфейсу, логічна структура, мультимовність). Система повинна коректно відображатися на різних пристроях — від мобільних телефонів до десктопів із високою роздільною здатністю, підтримувати адаптивний дизайн, а також залишатися функціональною за умов низької швидкості з’єднання або тимчасової відсутності Інтернету.

Це передбачає реалізацію офлайн-режиму із кешуванням базової карти, можливістю створення та редагування контенту локально з подальшою синхронізацією. Застосунок має підтримувати кросбраузерну сумісність, працювати стабільно в середовищах Chrome, Firefox, Safari, Edge, а також враховувати доступність для осіб із обмеженими можливостями — наприклад, через застосування ARIA-атрибутів, можливість масштабування шрифтів, керування з клавіатури. Інтерфейс має бути мінімалістичним, з високим контрастом, доступною навігацією та зрозумілою системою повідомлень.

Забезпечення високої доступності також включає моніторинг працездатності сервісів, автоматичне відновлення після збоїв (self-healing), регулярне резервне копіювання даних і використання надійної інфраструктури для хостингу з гарантією високої відмовостійкості.

Усі зазначені нефункціональні вимоги повинні бути враховані на етапі проєктування системи, оскільки саме вони визначають реальну здатність платформи масштабуватись, залишатися стабільною, адаптивною, безпечною та доступною для широкої аудиторії в умовах сучасного динамічного інформаційного середовища. Важливість кожної з функціональних вимог зображено на рисунку 1.2.

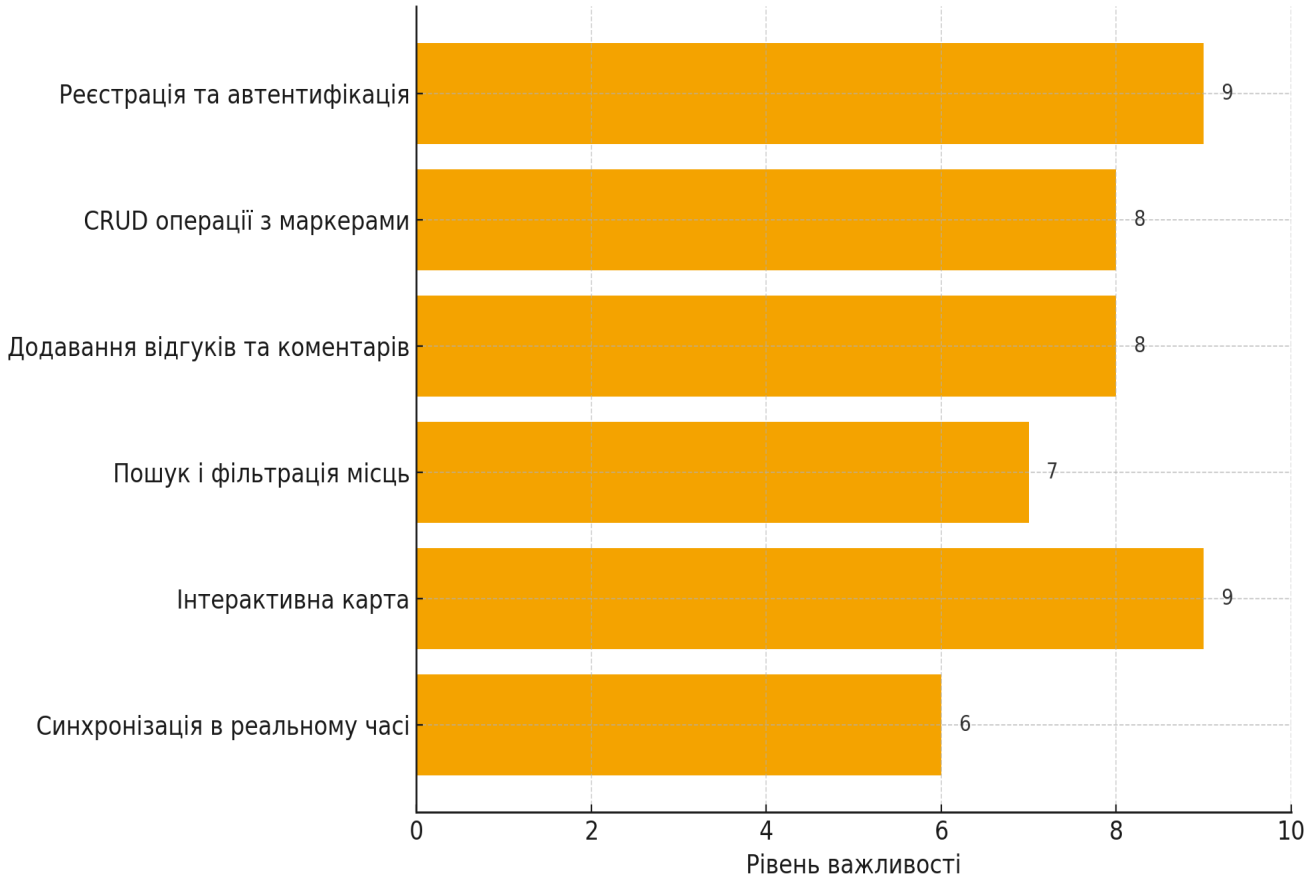


Рис. 1.2 Рівень важливості функціональних вимог до системи обміну туристичним досвідом

Висновки до розділу 1

У межах першого розділу було здійснено комплексний аналіз існуючих web-застосунків для обміну туристичним досвідом, серед яких розглянуто TripAdvisor, Google Maps і Maps.me. Кожен із зазначених сервісів демонструє унікальні підходи до побудови інтерфейсу, реалізації функціоналу та організації архітектури, однак

жоден із них не забезпечує повноцінної гнучкості, автономності й відкритості, необхідної для сучасного користувача.

TripAdvisor має розвинену систему відгуків і рейтингу, проте страждає від недостатньої інтерактивності та непрозорості алгоритмів. Google Maps забезпечує високу точність навігації, інтеграцію з іншими сервісами та персоналізовані рекомендації, але його функціональність перевантажена і залежить від стабільного інтернет-з’єднання. У свою чергу, Maps.me орієнтований на офлайн-користування, проте має обмежені можливості взаємодії між користувачами та слабку персоналізацію контенту.

Аналіз цільової аудиторії показав наявність стійкого запиту на рішення, яке поєднує переваги всіх трьох підходів: простоту та автономність Maps.me, багатофункціональність Google Maps і соціальну взаємодію TripAdvisor. Виявлені проблеми – централізація контенту, низька підтримка локальної інформації, відсутність взаємодії в реальному часі, перевантаженість інтерфейсів – обґрунтовують доцільність створення нового застосунку.

Сформульовані функціональні та нефункціональні вимоги передбачають побудову масштабованої системи з інтерактивною картою, підтримкою WebSocket, безпечною автентифікацією, офлайн-режимом і адаптивним інтерфейсом. Отже, результати аналізу є основою для подальшого проєктування та реалізації web-застосунку, орієнтованого на обмін туристичним досвідом у реальному часі з урахуванням потреб різних категорій користувачів.

2 ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Архітектура програмної системи

Клієнт-серверна модель є фундаментальною архітектурною парадигмою, що лежить в основі розробки web-застосунку для обміну туристичним досвідом. Вона передбачає розподіл системи на два логічно відокремлені компоненти — клієнт, який забезпечує взаємодію з користувачем, і сервер, який відповідає за обробку запитів, виконання бізнес-логіки, управління даними та захист інформації. У межах цієї архітектури клієнтська частина реалізована за допомогою фреймворку React з використанням мови програмування TypeScript, що забезпечує типізацію, зниження ризиків помилок і покращення передбачуваності поведінки коду.

Основна відповідальність клієнта полягає у відображенні інтерфейсу, обробці подій взаємодії користувача (натискання, заповнення форм, перемикання між сторінками), формуванні HTTP-запитів до сервера та обробці отриманих відповідей. Клієнт не зберігає критичних даних — усі операції з базою, автентифікація, авторизація і логіка перевірок виконуються на серверній стороні, що унеможливлює несанкціонований доступ до ключових функцій шляхом маніпуляцій з клієнтським кодом.

Серверна частина побудована з використанням Node.js і Express, що дозволяє реалізувати модульну, масштабовану систему, яка обробляє вхідні HTTP-запити, взаємодіє з базою даних MongoDB через бібліотеку Mongoose і відповідає за верифікацію даних, перевірку прав доступу та логіку формування відповідей. Принципи розділення відповідальностей реалізовано шляхом розділення коду на кілька незалежних рівнів: маршрутизація (routes), контролери (controllers), сервіси (services), моделі даних (models) та middleware. Кожен з цих компонентів виконує чітко визначену роль. Роутери приймають запити з клієнта і передають їх до відповідного контролера залежно від HTTP-методу та URL. Контролери

обробляють логіку обробки запиту, викликають відповідні сервіси, які реалізують бізнес-логіку (наприклад, створення нового маркера, перевірка унікальності email, генерація токена), а сервіси вже працюють з моделями, які описують структуру документів у базі даних. Middleware-компоненти застосовуються для перевірки автентичності токенів, логування запитів, валідації вхідних даних, обмеження доступу до захищених маршрутів або обробки помилок. Така модульна побудова підвищує читаність, тестованість і розширюваність коду, дає змогу змінювати окремі частини без впливу на всю систему.

Взаємодія між клієнтом і сервером відбувається через HTTP-запити типу GET, POST, PUT, DELETE, які відправляються з клієнтської частини до відповідних REST API-ендпоінтів. Наприклад, при реєстрації користувача клієнт формує POST-запит із введеними даними, які передаються в тілі запиту на сервер, де обробляються контролером, перевіряються на унікальність, хешуються за допомогою bcrypt, після чого зберігаються в базі. У відповідь сервер повертає токен доступу (JWT), який клієнт зберігає у локальному сховищі та надсилає в кожному наступному запиті через заголовок Authorization.

Цей токен використовується для ідентифікації користувача та контролю доступу до захищених маршрутів — наприклад, створення нового маркера або редагування власного відгуку. Клієнт не має змоги виконати жодну операцію, не отримавши позитивного підтвердження від сервера. Крім REST-запитів, застосунок використовує WebSocket через бібліотеку socket.io для двосторонньої синхронізації подій у реальному часі. Це дозволяє оновлювати маркери на карті одразу після їх створення або редагування іншими користувачами без перезавантаження сторінки.

Отже, клієнт-серверна модель у даному застосунку забезпечує чітке розмежування логіки представлення (frontend) і логіки обробки даних (backend), що дозволяє дотримуватися принципів масштабованості, безпеки, повторного використання коду та гнучкості розробки. Кожна сторона системи виконує свою специфічну функцію: клієнт надає інтерфейс користувача та керує взаємодією, сервер опрацьовує запити, реалізує бізнес-логіку, гарантує безпеку та цілісність

даних, а база MongoDB забезпечує зберігання, фільтрацію та пошук структурованих об’єктів — маршрутів, відгуків, профілів і маркерів. Реалізація такої моделі є методологічно виправданою та технічно ефективною для задач, пов’язаних з обміном динамічним користувацьким контентом у масштабованому середовищі.

Взаємодія між клієнтською та серверною частинами в межах розробленого web-застосунку реалізується на основі двох ключових технологій: REST API та WebSocket. Поєднання цих підходів забезпечує не лише структуровану обробку запитів, але й динамічне оновлення даних у реальному часі, що є критично важливим для інтерактивного застосунку, орієнтованого на картографічну візуалізацію, публікацію маркерів та швидкий обмін інформацією між користувачами. Архітектурно така реалізація відповідає концепції багаторівневої клієнт-серверної взаємодії, де логіка бізнес-процесів розмежована між окремими модулями, що взаємодіють через стандартизовані протоколи.

REST (Representational State Transfer) є класичним способом обміну даними між клієнтом і сервером, що реалізований у проєкті на основі HTTP-запитів. Основні дії користувача — реєстрація, вхід, створення маркерів, редагування записів — реалізовані як маршрути на сервері (Node.js + Express), кожен з яких відповідає певному методу (GET, POST, PUT, DELETE) і обробляється через спеціалізовані контролери. Наприклад, при створенні нового об’єкта (маркеру) користувач заповнює форму, яка ініціює POST-запит до ендпоінту /api/mark. У запиті передається JSON-об’єкт, що включає координати, опис, категорію, зображення та інші поля. Сервер, отримавши запит, викликає відповідний контролер, який перевіряє валідність даних, виконує операції з базою даних MongoDB через бібліотеку Mongoose, і у відповідь надсилає клієнту підтвердження з ID створеного ресурсу.

З метою безпеки всі маршрути, що стосуються особистих або захищених дій (наприклад, створення маркерів, редагування профілю, збереження маршрутів), потребують автентифікації користувача. Для цього в проєкті реалізовано JWT-

авторизацію: після успішної реєстрації або входу користувач отримує JSON Web Token, який зберігається на клієнті (наприклад, у localStorage) і автоматично включається в заголовок Authorization у наступних HTTP-запитах. Серверна логіка включає middleware для перевірки токена, що дозволяє обмежити доступ до внутрішніх ресурсів і контролювати права користувача. Якщо токен недійсний або відсутній, запит відхиляється з відповідним статусом помилки, що реалізує захищену логіку маршрутизації.

Окрім REST API, застосунок активно використовує WebSocket — протокол, що забезпечує двосторонній обмін даними між клієнтом і сервером у реальному часі. Це дозволяє реалізувати динамічні оновлення карти без потреби у ручному перезавантаженні сторінки або повторному запиті даних. У застосунку WebSocket реалізовано за допомогою бібліотеки socket.io, що підтримується як на фронтенді (React) так і на бекенді (Express-сервер). Принцип роботи полягає в тому, що після створення або редагування маркера, сервер передає сигнал через канал socket.io всім підключеним клієнтам, які автоматично оновлюють відображення карти. Таким чином, якщо один користувач додав нову точку, усі інші бачать її без затримки, що підвищує інтерактивність та створює ефект спільної роботи з простором.

Технічно клієнт встановлює WebSocket-з’єднання з сервером одразу після завантаження застосунку. Сторона клієнта підписується на певні події (наприклад, "marker:created", "marker:updated"), і в разі отримання повідомлення тригерить оновлення локального стану карти. Сервер же, у свою чергу, після завершення обробки REST-запиту, додатково викликає відповідну подію WebSocket через спеціальний контролер (наприклад, socketController.js), яка транслюється всім клієнтам або за необхідності лише певній групі користувачів. Така модель є ефективною для масштабованих додатків, де дані змінюються часто, і від користувачів очікується швидке реагування.

Слід наголосити, що REST API і WebSocket у системі не дублюють функції, а доповнюють одне одного. REST використовується для передачі структурованих

даних з підтвердженням (наприклад, при створенні ресурсу), а WebSocket — для передачі сигналів про зміну стану системи. Така гібридна модель дозволяє оптимізувати навантаження: зменшити кількість запитів на сервер за рахунок push-оновлень і водночас зберегти контроль над логікою збереження та обробки даних. У випадку втрати з'єднання з WebSocket, клієнт може використовувати резервний механізм — періодичне оновлення через REST-запити (polling), що забезпечує надійність у нестабільних мережевих умовах.

Архітектурно це забезпечено через чітке розділення компонентів: REST-ендпоінти розташовані в папці routers/(mark.js, user.js), логіка WebSocket — у controllers/socketController.js, а клієнтська частина підписується на сокет-події в основному компоненті App.tsx або відповідному map-компоненті. Це дозволяє масштабувати кодову базу, додавати нові канали подій (наприклад, для повідомлень, лайків, маршрутів) без порушення вже реалізованої логіки.

Додатковою перевагою поєднання REST і WebSocket є зниження навантаження на серверну частину. Завдяки тому, що повторні запити на отримання списку маркерів замінено підпискою на події, зменшується частота запитів, що критично при великій кількості одночасно активних користувачів. У разі масштабування застосунку — наприклад, із підключенням тисяч користувачів — WebSocket дозволяє реалізувати подієву модель, яка не потребує кожного разу нової обробки запиту, а працює на основі централізованої трансляції змін.

Ще одним важливим аспектом є безпека WebSocket-з'єднання. У проєкті реалізовано підключення сокетів лише після автентифікації, коли клієнт передає токен у момент ініціалізації з'єднання. Сервер перевіряє токен за допомогою middleware перед тим, як дозволити підписку на події. Таким чином, навіть у WebSocket-каналі немає анонімного доступу, і кожна подія, що транслюється, контролюється з точки зору авторизації.

Клієнтська частина, реалізована на основі React + TypeScript, забезпечує типізовану передачу всіх структур, що надходять як через REST, так і через WebSocket. Це дозволяє зменшити кількість логічних помилок, забезпечити

передбачуваність коду, а також створити централізовану систему управління станом (через useContext або Redux), яка синхронізує зміни між компонентами. Наприклад, після отримання повідомлення про новий маркер, його обробка відбувається у глобальному стані, з подальшим оновленням карти, списків та пов’язаних з ними UI-елементів.

З погляду користувача, така система дає змогу одночасно взаємодіяти з мапою, бачити додавання нових точок іншими мандрівниками, редагувати власні записи та отримувати миттєвий зворотний зв’язок. У поєднанні з адаптивною верстkою, офлайн-режимом, JWT-автентифікацією та масштабованою базою даних MongoDB, це формує стійке середовище для обміну туристичним досвідом із високою інтерактивністю та стабільністю.

У підсумку, застосування REST API для базової логіки запитів і WebSocket для реактивної синхронізації забезпечує оптимальну архітектуру для web-застосунку з реальним навантаженням і великою кількістю взаємодій.

Такий підхід зображено на рисунку 2.1. Він дозволяє одночасно досягнути структурної організованості, відповідності принципам безпеки та гнучкості обміну даними між клієнтами в режимі реального часу, що є ключовим для цифрової платформи туристичного спрямування.

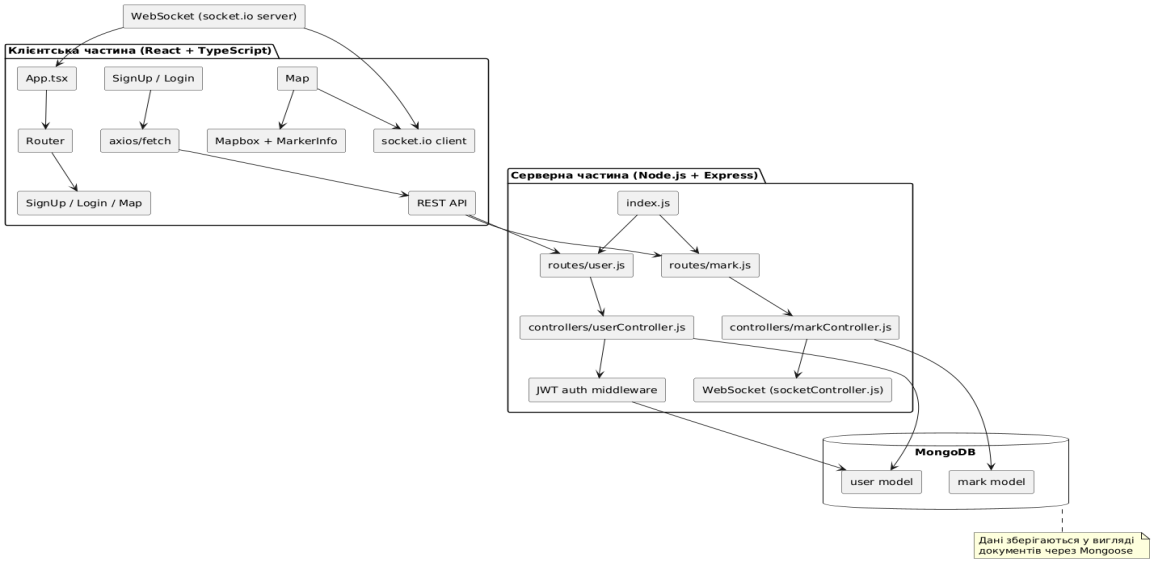


Рис. 2.1 Архітектура програмної системи

2.2 Вибір технологій та інструментів реалізації

Розробка сучасних web-застосунків вимагає ретельного вибору технологічного стеку, який не лише відповідає функціональним та нефункціональним вимогам системи, а й забезпечує масштабованість, зручність підтримки коду, швидку розробку та інтеграцію з зовнішніми API.

У межах створення застосунку для обміну туристичним досвідом обґрунтований вибір було зроблено на користь трьох ключових технологій: TypeScript, React і Express. Така комбінація дозволяє реалізувати повноцінну клієнт-серверну архітектуру з чітким розподілом відповідальностей між фронтендом і бекендом, а також інтегрувати додаткові сервіси (наприклад, WebSocket, Mapbox, MongoDB) із мінімальними витратами часу на конфігурацію.

Першим компонентом обраного стеку є TypeScript — надмножина JavaScript, що додає до мови строго типізовану модель. На відміну від традиційного JavaScript, який дозволяє гнучку, але неформалізовану обробку змінних, TypeScript змушує розробника заздалегідь визначати типи даних, структури об'єктів, інтерфейси та узгодження компонентів. У контексті великого проєкту з багатьма формами введення (наприклад, реєстрація, додавання маркерів, фільтрація карти), типізація дає змогу уникнути критичних помилок на етапі виконання, полегшує рефакторинг і зменшує кількість логічних багів.

Застосування TypeScript дозволяє виявити конфлікти типів ще під час компіляції, а також значно покращує досвід роботи в середовищах розробки (IDE), де автоматичне завершення коду, підказки та інтелектуальний аналіз структури проєкту прискорюють розробку. У межах нашого застосунку TypeScript застосовується на клієнтській частині, забезпечуючи безпечну взаємодію з REST API, правильну передачу даних у WebSocket-каналах та перевірку структур введення користувача (наприклад, об’єкти типу Marker, User, RouteInput). Це не лише знижує ризик некоректної роботи інтерфейсу, а й формує єдину систему очікувань щодо типів на стороні клієнта, що особливо важливо у багатокомпонентних інтерфейсах з використанням React.

Фреймворк React було обрано як основу для побудови інтерфейсу користувача. Його ключовою перевагою є компонентний підхід, який дозволяє розділити інтерфейс на логічні блоки (наприклад, Navbar, MapComponent, MarkerInfo, AddForm), кожен з яких відповідає за власний стан, логіку відображення та обробку подій. React дозволяє організувати гнучку архітектуру, де повторно використовувані компоненти зберігають ізольовану поведінку, що особливо важливо при масштабуванні застосунку.

Використання React у поєднанні з хуками (useState, useEffect, useContext) та системою маршрутизації (React Router) дає змогу ефективно керувати переходами між сторінками, зберігати стан користувача між сесіями та організовувати логіку відображення залежно від авторизації. У нашому випадку React використовується для побудови форм автентифікації (Login, SignUp, ForgotPassword), основного мап-компоненту з інтеграцією Mapbox, а також системи модальних вікон для редагування або перегляду маркерів. Поєднання React із SCSS-модулями забезпечує гнучке стилізування компонентів з урахуванням адаптивного дизайну, що особливо важливо при перегляді карти на мобільних пристроях. Окрему роль відіграє ефективна обробка подій: додавання маркера по подвійному кліку, валідація форм, реакція на оновлення WebSocket — усе це реалізовано засобами React, що дозволяє зберігати високу продуктивність і швидкість відгуку інтерфейсу.

Для серверної частини системи використано фреймворк Express, який є мінімалістичним і водночас потужним інструментом для побудови backend-логіки в середовищі Node.js. Express дозволяє створювати REST API з гнучкою системою маршрутизації, використанням middleware-функцій та підключенням до бази даних MongoDB через бібліотеку Mongoose. Вибір Express обумовлений кількома факторами. По-перше, він має низький поріг входу, широке ком’юніті та добре задокументовані практики. По-друге, Express дозволяє чітко організувати структуру коду — окремі папки для маршрутів, контролерів, моделей, конфігурацій — що полегшує підтримку й масштабування проєкту.

У застосунку Express відповідає за обробку запитів з клієнта (login, signup, create marker, update marker), валідацію введених даних, перевірку авторизації користувачів через JWT, а також за зв’язок із MongoDB, де зберігаються документи користувачів і маркерів. Крім того, Express інтегровано з бібліотекою socket.io, яка забезпечує двосторонню WebSocket-комунікацію для оновлення карти в режимі реального часу. Після обробки запиту, наприклад створення маркера, Express-трейл активує відповідну подію через сокет, яка миттєво передається іншим підключеним клієнтам, забезпечуючи синхронність відображення.

Технологічна сумісність між TypeScript, React і Express також виступає ключовою перевагою обраного стеку. Усі три компоненти є частиною JavaScript-екосистеми, що дозволяє використовувати однакові принципи розробки, стилі кодування, інструменти дебагінгу, засоби тестування та CI/CD. Наприклад, єдина структура DTO (Data Transfer Object) може бути спільною як для клієнта, так і для сервера, що усуває дублювання описів і забезпечує типову узгодженість. Також важливо зазначити, що всі обрані технології мають активну спільноту розробників, регулярні оновлення, відкриті репозиторії, що дозволяє швидко інтегрувати зовнішні модулі, вирішувати нетипові задачі, а також адаптувати застосунок до вимог безпеки (наприклад, захист від ін’єкцій, CORS, обмеження доступу).

Обраний стек є не лише технічно обґрунтованим, а й педагогічно доцільним, оскільки демонструє класичний повний цикл розробки від створення інтерфейсу до реалізації backend-логіки, обміну через API та зберігання даних у документно-орієнтованій базі. Це дозволяє розробнику набувати навичок роботи з сучасними фреймворками, вивчати архітектурні патерни, зокрема MVC (Model-View-Controller), реалізовувати авторизацію, кешування, масштабування, оптимізацію запитів.

У процесі реалізації сучасного web-застосунку для обміну туристичним досвідом одним із пріоритетних напрямів розробки стала оптимізація продуктивності клієнтської та серверної частин. Для досягнення високої швидкодії, мінімізації часу відповіді, плавності роботи інтерфейсу та стабільного

функціонування при підвищеному навантаженні було використано низку бібліотек та фреймворків, кожен з яких відіграє специфічну роль в архітектурі проєкту.

На фронтенді продуктивність оптимізується завдяки застосуванню Vite як сучасного інструменту збирання та сервера розробки, який замінює традиційний Webpack у багатьох нових проєктах. Vite використовує модульну систему ESBuild, що забезпечує практично миттєве перезбирання коду навіть при великій кількості компонентів, та дозволяє динамічно підвантажувати лише необхідні частини застосунку. У реальному застосунку це проявляється у швидкому рендерінгу карт, мінімальному часі завантаження сторінки входу, реєстрації або форми створення маркера. Крім цього, завдяки модульному імпорту SCSS-стилів та автоматичному поділу коду, Vite сприяє зменшенню обсягу JavaScript-пакета при першому завантаженні сторінки, що критично важливо для користувачів із мобільних пристроїв або повільним з'єднанням.

Ще одним інструментом оптимізації клієнтської частини виступає бібліотека React.memo у поєднанні з useCallback та useMemo, які застосовуються для запобігання повторному рендеру компонентів без змін у props або стані. Наприклад, при масштабуванні мапи або перемиканні між маркерами, завдяки застосуванню React.memo для компонентів MapMarker, Popup, MarkerInfo, система не виконує повторне відображення всіх елементів, а лише тих, у яких справді відбулися зміни. Це суттєво знижує навантаження на віртуальний DOM і покращує продуктивність при роботі з великою кількістю точок на карті. Важливу роль також відіграє бібліотека React Router, яка реалізує оптимізовану маршрутизацію із lazy-loading для сторінок. Завдяки цьому основний JavaScript-бандл не включає код усіх сторінок одночасно, а завантажує лише необхідне для поточної взаємодії (наприклад, карта або форма відновлення паролю).

Для асинхронної взаємодії з бекендом застосовується бібліотека Axios, яка дозволяє реалізувати HTTP-запити з підтримкою перехоплення помилок (interceptors), кастомних заголовків (для токена авторизації), тайм-аутів і глобального керування станами завантаження. Axios також забезпечує гнучку

обробку кешу та дозволяє повторно використовувати конфігурації запитів без надлишкового дублювання. У поєднанні з кастомним хендлінгом відповідей це дозволяє зменшити кількість дубльованих рендерів і зробити обробку помилок більш контрольованою. Наприклад, при відсутності з'єднання з сервером користувач отримує інформативне повідомлення замість довгого очікування.

На серверній частині одним із ключових інструментів оптимізації продуктивності виступає Express middleware. Усі запити проходять через попередньо налаштовані middleware-функції, що дозволяє централізовано обробляти валідацію, логування, авторизацію та обробку помилок без дублювання логіки у кожному маршруті. Наприклад, middleware для JWT-автентифікації (authMiddleware) перевіряє токен у кожному захищеному запиті, забезпечуючи при цьому високу швидкість обробки завдяки кешуванню перевіреного токена у пам'яті. Аналогічно middleware для логування (loggerMiddleware) дає змогу швидко діагностувати вузькі місця в запитах, фіксуючи час обробки кожного маршруту.

Особливої уваги в контексті продуктивності заслуговує використання бібліотеки socket.io, яка реалізує WebSocket-протокол для обміну даними між сервером і клієнтом у реальному часі. На відміну від періодичного опитування сервера (polling), WebSocket дозволяє підтримувати постійне двостороннє з'єднання, що є оптимальним для оновлення карти при появі нових маркерів. Завдяки використанню socket.emit та socket.broadcast сервер надсилає лише релевантні повідомлення активним клієнтам, не витрачаючи ресурси на нецільові запити. Крім цього, socket.io підтримує автоматичне перепідключення, що мінімізує втрати трафіку при обриві зв'язку.

Для роботи з базою даних використовується Mongoose, яка не тільки забезпечує об'єктно-документне відображення (ODM) MongoDB-документів, але й реалізує вбудовані механізми кешування, валідації, індексації та lean-запитів. Застосування lean() у запитах до бази значно прискорює обробку, оскільки дозволяє отримати чисті JSON-об'єкти без обгортання в моделі Mongoose. Наприклад, при

масовому рендері маркерів на карті lean-запит до колекції забезпечує швидке отримання лише необхідних полів без зайвих обчислень.

Ще одним елементом продуктивності є асинхронна обробка запитів, реалізована за допомогою async/await у всіх функціях доступу до бази та зовнішніх API. Це дозволяє уникнути блокування подій, обробляти велику кількість одночасних запитів і зменшити середній час відповіді сервера навіть при зростанні навантаження. Сервер реагує лише після отримання результату, не витрачаючи ресурси на очікування.

Додатково застосовується бібліотека dotenv для динамічного конфігурування змінних середовища, що дозволяє ізолювати конфіденційні параметри (ключі бази, секрети токенів, порти), а також швидко перемикатися між середовищами розробки, тестування та продакшену без перезбирання проєкту. Це опосередковано сприяє стабільності та продуктивності, оскільки дозволяє уникнути помилок конфігурації та забезпечує ізоляцію змін.

З метою додаткового контролю за ресурсами при розгортанні проєкту реалізовано підтримку кешування статичних ресурсів через HTTP-заголовки Cache-Control, що дозволяє браузеру зберігати стилі, шрифти, зображення локально і не звертатися до сервера при кожному завантаженні сторінки. У поєднанні з ефективною структурою директорій та стисненням відповідей (compression middleware), це зменшує обсяг трафіку та покращує час рендерингу.

Отже, оптимізація продуктивності в системі досягається не за рахунок одного інструменту, а шляхом поєднання низки спеціалізованих бібліотек і фреймворків на всіх рівнях архітектури. Від Vite та React.memo на фронтенді до lean-запитів Mongoose, Express middleware та socket.io — кожен елемент інтегрується з урахуванням вимог продуктивності, асинхронності й масштабованості. Цей підхід, який презентовано на рисунку 2.2, гарантує стабільну роботу застосунку при реальному навантаженні, забезпечує швидку взаємодію з користувачем і формує позитивний досвід роботи з платформою.

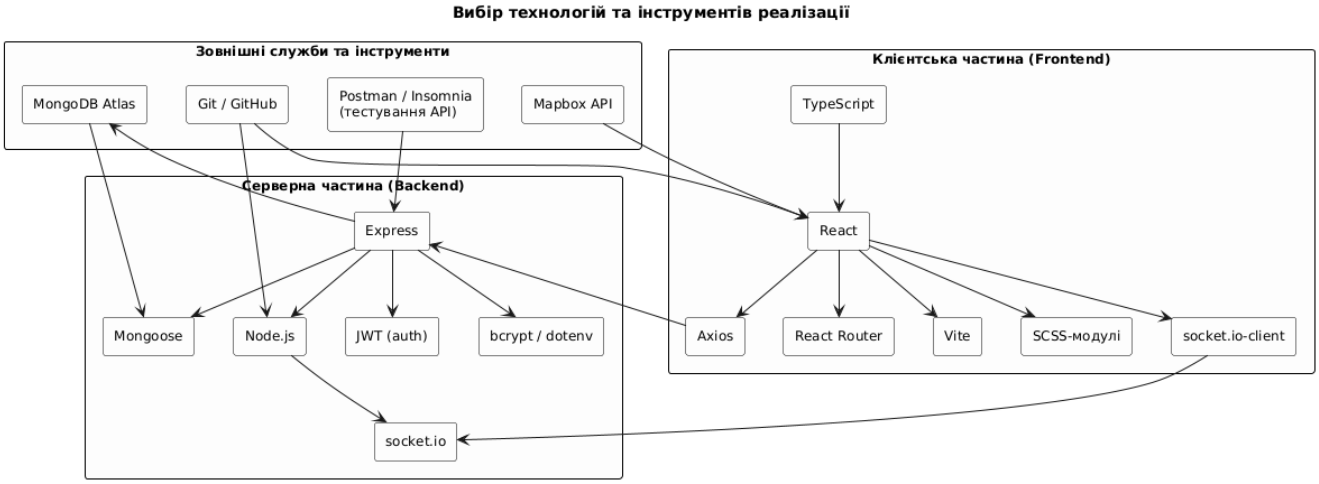


Рис. 2.2 Вибір технологій та інструментів реалізації

2.3 Реалізація захищеної автентифікації та роботи з даними

Організація реєстрації, входу користувача, токенізації та контролю доступу є фундаментальними елементами безпеки та індивідуалізації взаємодії в будь-якому сучасному web-застосунку. У розробленій системі обміну туристичним досвідом реалізовано повноцінну систему автентифікації та авторизації з використанням токенів доступу на основі стандарту JWT (JSON Web Token), хешуванням паролів через bcrypt, валідацією вхідних даних на стороні сервера, а також перевіркою прав доступу до захищених маршрутів та ресурсів. Реєстрація та вхід користувача побудовані на класичному підході через електронну пошту й пароль, з подальшою генерацією токена, який використовується для ідентифікації в усіх подальших запитах до серверу.

Процес реєстрації користувача починається з фронтенд-форми, створеної за допомогою React та TypeScript. Інтерфейс форми включає поля для введення email, паролю, підтвердження паролю та, за потреби, додаткових атрибутів (наприклад, ім’я користувача). На рівні клієнта реалізована базова валідація формату email (через регулярні вирази), мінімальної довжини пароля, а також перевірка співпадіння пароля з полем підтвердження. Після успішної перевірки дані

передаються на бекенд через HTTP POST-запит до ендпоінту /api/auth/register, з використанням бібліотеки Axios. На сервері цей запит обробляється контролером authController, де дані додатково перевіряються за допомогою бібліотеки express-validator або вручну. Пароль перед записом до бази даних шифрується за допомогою бібліотеки bcrypt, яка генерує соль і застосовує багаторазове хешування, унеможливлюючи розшифрування пароля навіть у разі компрометації бази даних.

Після створення користувача сервер генерує JWT-токен, який містить зашифровану інформацію про користувача: його ID, email та роль. Цей токен підписується секретним ключем (який зберігається у .env через dotenv) і надсилається клієнту у відповідь. Клієнт зберігає токен у локальному сховищі (наприклад, localStorage або sessionStorage) або в HTTP-only cookie (залежно від реалізації), і автоматично додає його до всіх подальших запитів у заголовку Authorization: Bearer <token>. Такий механізм дозволяє уникати повторної автентифікації при кожному переході між сторінками й водночас забезпечує високий рівень захисту за рахунок обмеженого строку дії токена та його цифрового підпису.

Процес входу в систему реалізовано за схожим принципом. Користувач вводить email і пароль, які через форму надсилаються на серверний ендпоінт /api/auth/login. Контролер перевіряє наявність користувача в базі даних MongoDB, і якщо користувач знайдений, то введений пароль звіряється з тим, що збережено у базі, за допомогою методу bcrypt.compare(). У разі збігу сервер знову генерує JWT-токен, який містить ідентифікатор користувача, його роль і строк дії (наприклад, 1 година), та відправляє його клієнту. У випадку помилок — неправильний email, невірний пароль, невалідний формат даних — користувач отримує відповідь з кодом помилки (наприклад, 401 або 403) та поясненням у форматі JSON, що дозволяє клієнту відобразити відповідне повідомлення.

Система токенізації на базі JWT має кілька рівнів захисту. По-перше, токен є зашифрованим і підписаним — його не можна підробити без доступу до секретного

ключа сервера. По-друге, кожен запит, що потребує автентифікації (наприклад, створення нового маркера, редагування профілю, надсилання маршруту), перевіряється middleware-функцією authMiddleware. Ця функція розпізнає токен у заголовку запиту, розшифровує його, перевіряє підпис та строк дії, після чого прикріплює об’єкт користувача до запиту (через req.user). Це дозволяє контролерам точно знати, хто виконує дію, і реалізувати логіку авторизації — наприклад, заборонити редагування чужого маркера або надання адміністративного доступу до списку користувачів. У разі відсутності або недійсності токена запит одразу припиняється, і повертається код 401.

Окрему увагу приділено контролю доступу, який базується не лише на перевірці токена, але й на ролях. У системі передбачено щонайменше два рівні доступу: звичайний користувач (User) і адміністратор (Admin або Moderator). Для цього в базі зберігається атрибут role, який декодується із JWT та перевіряється у middleware або контролерах. Наприклад, лише користувач з правами адміністратора має доступ до ендпоінтів типу /api/admin/users, які повертають список усіх зареєстрованих облікових записів. Аналогічно, для кожного ресурсу (маркер, відгук, маршрут) зберігається поле authorId, яке звіряється з req.user.id, що дозволяє реалізувати правила доступу до редагування або видалення лише власного контенту.

Забезпечення безпеки при передачі токенів реалізовано через обов’язкове використання HTTPS на всіх рівнях комунікації. Токени з обмеженим строком дії дозволяють зменшити ризики у разі витоку (наприклад, на публічних пристроях), а для ще більшої безпеки може бути реалізовано механізм refresh token, що дозволяє оновлювати основний токен без повторної автентифікації, зберігаючи при цьому контроль над сесією.

Хоча ця функціональність є додатковою, вона сумісна з обраною архітектурою й може бути додана через окремий маршрут /api/auth/refresh.Інтеграція з WebSocket (через socket.io) також враховує автентифікацію. Під час встановлення з’єднання клієнт надсилає токен як

параметр, який перевіряється сервером. Усі події, що передаються через WebSocket — наприклад, створення або оновлення маркера — супроводжуються інформацією про користувача, що дозволяє перевірити права на оновлення, транслювати зміни лише певним користувачам (наприклад, за географічним фільтром) та забезпечити контроль над подіями в реальному часі.

На клієнтській частині вся логіка автентифікації реалізована централізовано через контекст або хук useAuth(), що дозволяє зберігати стан поточного користувача в глобальному об’єкті, а також автоматично додавати токени до всіх запитів через налаштування Axios. При кожному оновленні сторінки виконується перевірка наявності токена у сховищі, його декодування та передача до глобального стану. Якщо токен відсутній або недійсний — користувач автоматично перенаправляється на сторінку входу. Такий механізм забезпечує безперервну роботу застосунку та дозволяє зберігати контекст користувача навіть після перезавантаження сторінки.

У розробленому web-застосунку для обміну туристичним досвідом зберігання всіх даних реалізовано на основі нереляційної бази даних MongoDB, що дозволяє ефективно працювати з напівструктурованими об’єктами, забезпечуючи високу гнучкість, масштабованість та швидкість доступу. Для взаємодії з базою даних використовується бібліотека Mongoose, яка реалізує механізм ODM (Object Document Mapping) і дозволяє працювати з MongoDB у вигляді моделей, схем та типізованих документів. У межах системи було розроблено декілька основних моделей: User, Marker, Comment, Route, кожна з яких виконує чітко визначену роль в архітектурі та містить поля, що відповідають вимогам як функціональності, так і захисту інформації.

Модель User містить атрибути, які визначають обліковий запис користувача: email, password, role, createdAt, isActive, а також додаткові поля, як-от ім’я або зображення профілю. Однією з головних вимог до безпечного зберігання користувацьких даних є недопущення збереження паролів у відкритому вигляді. Для цього перед збереженням документа виконується middleware-функція pre('save'), яка за допомогою бібліотеки bcrypt хешує пароль, додаючи соль

і використовуючи багаторазове шифрування. У разі спроби повторного збереження профілю без зміни пароля перевірка забезпечує уникнення повторного хешування. Таким чином, навіть у разі витоку бази даних атака типу "reverse hash" є практично неможливою. Крім того, модель користувача не повертає поле password у відповідях API, оскільки його виключено через select: false, а при використанні lean() — дані повертаються як простий об’єкт без зайвих властивостей.

Модель Marker відповідає за зберігання геолокаційних об’єктів, які створюють користувачі. Структура моделі включає такі ключові поля: title, description, coordinates (масив з широтою і довготою), category, authorId, images, createdAt, rating, а також внутрішні службові поля — статус публікації, перевірку модератором тощо. Для підвищення ефективності запитів по геолокації реалізовано індексацію координат як 2dsphere index, що дозволяє MongoDB виконувати просторові запити (наприклад, "знайти всі маркери в радіусі 5 км"). Це особливо актуально при використанні карти, де користувач може фільтрувати маркери за близькістю до поточного місця або регіону. Кожен маркер прив’язаний до конкретного користувача через поле authorId, що дозволяє реалізувати контроль доступу (право на редагування або видалення) на основі порівняння з ідентифікатором, отриманим з JWT-токена.

Модель Comment прив’язана до Marker і містить посилання на коментарі користувачів. Її структура включає поля text, authorId, markerId, createdAt, а також додаткову інформацію про рейтинг коментаря (наприклад, лайки або репорти). Для захисту від надмірного навантаження (спам-коментарі, атаки на базу) реалізовано обмеження кількості коментарів від одного користувача за одиницю часу (rate-limiting), яке впроваджується на рівні контролера через кеш або в поєднанні з Redis. Водночас самі коментарі можуть фільтруватися як у маршрутах, так і на клієнті — залежно від мови, довжини або наявності заборонених слів (фільтрація може реалізовуватись через бібліотеки типу bad-words або кастомні фільтри).

Дані зберігаються в хмарному сервісі MongoDB Atlas, що забезпечує автоматичне шифрування на рівні зберігання (storage-level encryption), резервне копіювання та інструменти моніторингу навантаження. Серверна частина взаємодіє з базою даних через спеціальні репозиторії моделей, де виклики методів (.find(), .findById(), .updateOne(), .aggregate()) виконуються через асинхронні функції з перевіркою помилок та логуванням. Для захисту доступу до бази використовується змінна середовища з рядком з'єднання (MONGO_URI), яка зберігається у .env і не потрапляє до публічного репозиторію. Додатково в MongoDB реалізовано обмеження доступу за IP-адресами, а також параметри користувацьких ролей (наприклад, readWrite, readOnly), що дозволяє уникнути несанкціонованого доступу навіть у разі компрометації сервера.

На рівні запитів реалізовано вибіркове повернення даних, що дозволяє клієнту отримувати лише необхідні поля. Наприклад, при запиті до /api/markers використовуються конструкції .select('title coordinates rating'), що дозволяє мінімізувати трафік і підвищити продуктивність застосунку, особливо в мобільних умовах. Аналогічно, використання методу .lean() дозволяє повертати дані як прості об'єкти, що пришвидшує серіалізацію й зменшує навантаження на пам'ять. У разі потреби більш складної обробки застосовуються агрегатні запити (aggregate pipeline), наприклад, для отримання топ-10 популярних місць у певному регіоні за останній тиждень.

Важливою частиною захисту даних є контроль прав доступу до записів у базі. Усі маршрути сервера, які оперують з ресурсами, мають проміжний шар перевірки: автентифікація за токеном, перевірка наявності об'єкта в базі, перевірка відповідності authorId до ID поточного користувача. Наприклад, користувач не може видалити чужий маркер, навіть якщо знає його ID, оскільки перед виконанням .deleteOne() контролер перевіряє умову authorId === req.user.id. Такий рівень контролю дозволяє захистити дані навіть у разі підробки запиту.

Серверна частина додатково захищена від загроз типу NoSQL injection, оскільки всі запити формуються строго типізовано, без прямого підставлення

змінних у фільтри. Дані, отримані з запитів, проходять перевірку (express-validator, Joi або кастомна валідація), що дозволяє уникнути атак із підміною структури фільтрів ({ $ne: null }, { $gt: "" } тощо). Всі поля, що використовуються у запитах до бази, попередньо перевіряються на тип, довжину, формат, наявність дозволених значень.

Таблиця 2.1

Реалізація захищеної автентифікації та роботи з даними

Компонент	**Технологія / Метод**	**Функціональне призначення**
Реєстрація користувача	bcrypt, Mongoose, Express	Хешування паролів, збереження користувача у базі, валідація полів
Вхід до системи	JWT, bcrypt, Axios	Перевірка пароля, генерація токена, автентифікація клієнта
Токенізація	jsonwebtoken, dotenv, middleware	Формування JWT-токена, підпис, строк дії, передача в заголовку Authorization
Контроль доступу	JWT middleware, role check	Обмеження доступу до маршрутів, перевірка прав (User/Admin), контроль доступу до об’єктів
Захищені запити	Express, authMiddleware	Захист REST API: тільки авторизовані користувачі мають доступ до певних дій
Структура даних	Mongoose models	Схеми для User, Marker, Comment з типізацією, валідацією, референсами
Захист паролів	bcrypt.hash()	Сольове хешування паролів при збереженні в MongoDB
Безпека передачі даних	HTTPS, Axios, JWT	Шифрування при передачі, безпечна авторизація
Захист від SQL/NoSQL атак	Валідація, типізація, express-validator	Очищення вхідних даних, перевірка типів, недопущення підставлення операторів у запити
Приватність у відповідях	select: false, .lean()	Приховання поля паролю, мінімізація чутливої інформації в API-відповідях

2.4 Інтерфейс користувача та інтерактивна карта на базі Mapbox

Інтерфейс користувача розробленого web-застосунку є ключовим елементом, що забезпечує зручність навігації, інтуїтивну взаємодію та ефективний обмін інформацією про туристичні об’єкти. Його побудовано з використанням компонентної структури React у поєднанні з TypeScript, SCSS-модулями та адаптивним дизайном. У рамках реалізації було визначено набір основних компонентів, які відіграють центральну роль у побудові користувацького досвіду. До таких компонентів належать Navbar, MapMarker та AddForm — кожен з яких реалізує окрему логіку відображення та взаємодії в межах системи.

Компонент Navbar (навігаційна панель) відповідає за управління загальною структурою доступу користувача до функціональних розділів системи. Його реалізовано у вигляді фіксованого верхнього елемента інтерфейсу, який відображається на всіх сторінках після авторизації. Залежно від ролі користувача (User чи Admin) та його авторизаційного статусу, навбар динамічно змінює свій вміст. Для звичайного користувача він містить кнопки переходу на головну карту, сторінку профілю, обране та вихід із системи, тоді як для адміністратора можуть бути додані посилання на панель модерації або перегляд усіх користувачів. Реалізація навбару враховує адаптивність: при ширині екрану менше визначеного порогу застосовується мобільна версія з «гамбургер»-меню, що відкриває список пунктів у вигляді бічної панелі.

Візуальне оформлення компонента реалізовано через SCSS-модулі, з використанням змінних кольорів, медіа-запитів та гнучкої сітки. Навбар також включає індикатор авторизованого користувача — ініціали або аватар, що завантажується із профілю, а також інтеграцію зі станом автентифікації через контекст або глобальний хук useAuth, який надає доступ до імені користувача та дозволяє реалізувати вихід із системи при кліку на відповідну кнопку (logout() викликає очищення токена та редирект).

Компонент MapMarker є центральним елементом візуалізації маркерів на карті. Він відображає кожну точку, що створена користувачами або була попередньо

додана до бази даних, і розташовується на основі координат, отриманих із MongoDB та переданих через WebSocket або REST API. У межах інтеграції з Mapbox використано механізм Marker та Popup — маркери створюються за допомогою референсу до DOM-елемента, стилізованого окремим SCSS-модулем, який може змінювати вигляд залежно від категорії точки (наприклад, історія, природа, гастрономія). Після натискання на маркер активується модальне вікно або спливаюча підказка (Popup), яка містить назву, рейтинг, фотографію, дату додавання та кнопку перегляду детальнішої інформації.

Компонент використовує useEffect для підключення маркерів після завантаження карти та очищення маркерів при зміні зуму або координат карти. Крім цього, MapMarker реалізує механізм оновлення маркерів у реальному часі: при надходженні подій з каналу socket.io (наприклад, marker:created, marker:updated) стан компоненту оновлюється через useState, а нові маркери додаються до карти без перезавантаження сторінки. Для оптимізації рендерингу використано React.memo, що дозволяє уникати повторного створення DOM-елементів для маркерів, які не зазнали змін, а також ключі (keys), які відповідають унікальному id маркера в базі.

Компонент AddForm реалізує логіку створення нової туристичної точки — ключову функцію взаємодії з системою. Він відкривається у вигляді модального вікна або правої бічної панелі після кліку по кнопці «Додати точку» або подвійного кліку на мапі. У формі користувач може вказати назву локації, опис, категорію (обирається з випадаючого списку), завантажити одне або декілька зображень, а також ввести координати вручну або автоматично (через click на карту).

Компонент реалізовано як контрольовану форму з використанням хуків useForm (або кастомних хуків) і підтримкою валідації полів (обов’язковість, довжина, формат тощо). Для відправки запиту на створення маркера використовується Axios, який надсилає POST-запит на маршрут /api/markers, додаючи токен авторизації в заголовок. У разі успішної відповіді користувач отримує повідомлення про створення точки, форма закривається, а новий маркер

з’являється на карті в режимі реального часу через канал WebSocket (socket.emit('marker:created')).

У межах компонента AddForm реалізовано також механізм відображення мініатюр завантажених зображень, перевірка формату файлів (JPEG, PNG), а також обмеження на розмір і кількість (наприклад, не більше 5 фото по 2 МБ). Компонент працює з локальним станом, де кожне поле має окрему змінну стану, що дозволяє динамічно виводити повідомлення про помилки або попередження. Додатково передбачено збереження чернетки — наприклад, у localStorage — для того, щоб користувач не втратив введені дані при випадковому закритті модального вікна або оновленні сторінки.

Усі основні компоненти інтегруються через головний компонент карти (MapPage), що відповідає за загальну логіку управління станом, взаємодію з сервером, передачу даних до дочірніх елементів, а також обробку WebSocket-подій. Для зберігання глобального стану використано useContext або Redux, що дозволяє підтримувати синхронізацію даних між Navbar, MapMarker, AddForm, Sidebar та іншими елементами. Уся взаємодія з інтерфейсом реалізована згідно з принципами реактивності та одноманітності: оновлення в одному компоненті (наприклад, додавання маркера) автоматично синхронізується з іншими без необхідності ручного оновлення.

Взаємодія з інтерактивною картою є центральним сценарієм використання web-застосунку, що реалізує механізм додавання, редагування та оновлення геолокаційних маркерів у режимі реального часу. Основу картографічного відображення становить бібліотека **Mapbox GL JS**, інтегрована в React-компонент карти. У поєднанні з WebSocket, React-хуками та REST API, ця система дозволяє реалізувати повноцінну подієво-керовану взаємодію, яка охоплює створення нових точок на основі координат, редагування їхніх властивостей та автоматичне оновлення всіх змін без перезавантаження сторінки.

Додавання нового маркера ініціюється користувачем безпосередньо з карти: у стандартному режимі активується подвійний клік на відповідну ділянку карти, в

результаті чого обробник події (onDblClick) отримує координати місця натискання (lngLat). Ці координати зберігаються у стані (useState) та передаються у форму створення (AddForm), яка відкривається у вигляді модального вікна або слайдера з правого боку екрана. Користувач вводить назву, опис, обирає категорію, додає фото, а після заповнення форми натискає кнопку "Створити". На цьому етапі викликається функція handleSubmit, яка через Axios надсилає POST-запит на серверний маршрут /api/markers, передаючи координати й решту полів маркера у форматі JSON.

На сервері запит обробляється через Express-контролер, який виконує валідацію даних, перевірку автентифікації через JWT, після чого формує новий документ у колекції MongoDB (MarkerModel.create(...)). Після успішного створення нової точки, крім стандартної HTTP-відповіді, сервер викликає подію WebSocket через socket.emit('marker:created', newMarker), яка транслюється всім підключеним клієнтам. На стороні клієнта підписка на цю подію (socket.on('marker:created', callback)) запускає функцію оновлення локального стану карти: до масиву маркерів додається нова точка, яка відображається без потреби в оновленні сторінки.

Візуально новий маркер рендериться через компонент MapMarker, який отримує координати, колір, категорію та інші атрибути, і прикріплюється до карти у вигляді DOM-елемента, пов’язаного з Mapbox-інтерфейсом.

Редагування маркерів можливе лише для користувачів, які є їх авторами. При натисканні на існуючий маркер відкривається спливаюча підказка (Popup) з детальною інформацією та кнопкою «Редагувати». Натискання цієї кнопки відкриває форму редагування, де поля вже заповнені наявними даними. Користувач має змогу змінити назву, опис, зображення, категорію або координати. Після внесення змін надсилається PUT-запит на маршрут /api/markers/:id, який включає оновлені поля.

Сервер перевіряє автентичність токена, порівнює ID автора маркера з ID поточного користувача (отриманого з токена), і в разі відповідності виконує

оновлення документа в базі даних за допомогою MarkerModel.findByIdAndUpdate(...).

Одразу після оновлення, сервер надсилає повідомлення WebSocket через socket.emit('marker:updated', updatedMarker), що запускає механізм оновлення маркера на всіх клієнтських інтерфейсах. Зміни застосовуються до масиву маркерів у React-стані (setMarkers(prev => prev.map(...))), і оновлений DOM-елемент замінює старий без втрати позиції на карті або скидання масштабу.

У випадку, якщо змінилися координати маркера, маркер видаляється з карти (marker.remove()) і додається заново в новому місці. Якщо змінився лише опис або зображення, оновлення виконується лише в Popup, зберігаючи позицію на карті.

У разі видалення маркера використовується окрема кнопка у Popup, яка викликає DELETE-запит на /api/markers/:id. Після підтвердження сервер видаляє документ з бази та розсилає повідомлення WebSocket marker:deleted.

Клієнтська частина фільтрує масив маркерів і виключає елемент із відповідним ID (setMarkers(prev => prev.filter(m => m.id !== id))), після чого маркер зникає з карти. Видалення маркера можливе лише для його автора або адміністратора, і ця перевірка реалізована на рівні контролера через зіставлення req.user.id та marker.authorId.

Для підтримки продуктивності карта оптимізована за допомогою механізмів React.memo, обмеження кількості одночасно рендерених маркерів (на основі вікна огляду карти), а також використання useEffect із залежностями ([map, markers]) для уникнення зайвих повторних рендерів.

Усі події, пов’язані з картою, обробляються в одному компоненті (MapComponent.tsx), який підписаний на канали socket.io, отримує список активних маркерів із REST API при першому завантаженні, а далі підтримує синхронізацію через події WebSocket. Таким чином, навіть за великої кількості користувачів і частих змін, карта залишається стабільною, чутливою до подій і візуально актуальною.

Таблиця 2.2

Основні елементи інтерфейсу та інтерактивної карти

Компонент	Призначення	Технології / Бібліотеки
Navbar	Панель навігації; перемикання між сторінками; кнопка «Вихід»	React, SCSS-модулі, useAuth, useContext
MapPage	Основна карта з маркерами; контейнер для Mapbox	React, Mapbox GL JS, Axios, socket.io
MapMarker	Візуалізація точок на мапі; відображення Popup з деталями	Mapbox Marker, React.memo, useEffect
AddForm	Додавання маркерів; форма з назвою, описом, фото	React, Axios, SCSS, useForm, useState
Popup	Відображення деталей маркера; кнопки редагування й видалення	Mapbox Popup, React
UpdateForm	Форма редагування даних маркера	React, Axios, useState
Pages/	Реєстрація, вхід, відновлення паролю	React Router, Axios, JWT
Контекст (Auth)	Глобальний стан користувача, токени, доступ до ролей	useContext, useReducer
Інтерактивність	Додавання по кліку, оновлення через WebSocket	socket.io, REST API
Адаптивність	Підтримка мобільних пристроїв і змін розміру екрана	CSS media queries, flex, grid

Висновки до розділу 2

У другому розділі було здійснено поетапне проєктування та реалізацію web-застосунку для обміну туристичним досвідом із урахуванням сучасних архітектурних, функціональних і безпекових вимог. Основу програмної системи становить клієнт-серверна модель з чітким розмежуванням логіки між frontend- та backend-частинами. Клієнт розроблено з використанням React та TypeScript, що забезпечило високу типізованість, передбачуваність коду та ефективну побудову компонентного інтерфейсу. Сервер реалізовано на платформі Node.js із

фреймворком Express, що дало змогу сформувати модульну, масштабовану архітектуру з підтримкою middleware, REST API та WebSocket-підключення.

Особливу увагу приділено реалізації захищеної автентифікації: використано JWT-токени, bcrypt для хешування паролів, перевірку прав доступу на основі ролей користувачів, а також перевірку авторства контенту. Усі чутливі операції обмежені через middleware, що гарантує безпеку персональних даних. Зберігання інформації організовано в MongoDB із використанням ODM-бібліотеки Mongoose, що дозволило працювати з напівструктурованими документами, реалізувати геоіндексацію та забезпечити масштабованість запитів до бази.

Інтерфейс користувача побудовано з урахуванням принципів адаптивності, реактивності та зручності навігації. Ключовим компонентом є інтерактивна карта на основі Mapbox, яка взаємодіє з WebSocket-каналом для миттєвого оновлення маркерів у реальному часі. Створення, редагування та видалення маркерів реалізовано з дотриманням логіки авторизації та синхронізації між користувачами.

Як результат, реалізовано архітектурно збалансовану, безпечну та інтерактивну систему, здатну працювати зі значною кількістю динамічних об’єктів. Такий підхід дозволяє ефективно вирішувати завдання обміну туристичною інформацією та забезпечує основу для подальшого масштабування функціоналу.

3 ТЕСТУВАННЯ ТА ПЕРСПЕКТИВИ РОЗВИТКУ ПРОГРАМИ

3.1 Методика тестування та виявлення помилок

Для оцінки якості функціонування веб-застосунку було проведено функціональне, інтеграційне та частково навантажувальне тестування з акцентом на критично важливі підсистеми: автентифікація користувача, обробка маркерів, взаємодія з картою, робота REST API та обробка подій через WebSocket. Тестування проводилося в ручному режимі з використанням тест-кейсів, сформованих за логікою очікуваної поведінки системи. Загальна кількість виконаних тестів становила:

$$N_{загальна} = 92$$

Кількість успішно пройдених тестів, тобто тих, що завершилися без помилок або збоїв, дорівнює:

$$N_{\text{успішних}} = 74$$

Для обчислення загального рівня точності тестування було використано класичну формулу ефективності:

$$P = \frac{N_{\text{успішних}}}{N_{\text{загальна}}} \cdot 100\backslash\%, \tag{3.1}$$

де

$N_{\text{успішних}}$ – кількість тестів, що завершились без помилок;

$N_{\text{загальна}}$ – загальна кількість тестів.

Підставивши значення, отримаємо:

$$P = \frac{74}{92} \cdot 100\backslash\% = 80{,}43\backslash\%$$

Таким чином, успішно виконано понад 80 % тестів, що свідчить про високий рівень стабільності системи на етапі розробки.

Окремі модулі показали різний рівень стабільності:

Для модуля автентифікації (реєстрація, вхід, авторизація через JWT) успішно виконано 18 із 20 тестів:

$$P_{\text{auth}} = \frac{18}{20} \cdot 100\backslash\% = 90\backslash\%$$

Для обробки маркерів через API (/markers, /markers/:id) — 28 із 34:

$$P_{\text{маркери}} = \frac{28}{34} \cdot 100\backslash\% \approx 82{,}35\backslash\%$$

Для обробки WebSocket-подій (створення та оновлення маркерів у реальному часі) — 11 з 15:

$$P_{\text{WebSocket}} = \frac{11}{15} \cdot 100\backslash\% \approx 73{,}33\backslash\%$$

Для перевірки користувацького інтерфейсу (форм реєстрації, додавання маркерів, pop-up’ів) — 17 з 23:

$$P_{\text{UI}} = \frac{17}{23} \cdot 100\backslash\% \approx 73{,}91\backslash\%$$

Було виявлено 18 помилок, які умовно класифіковано так:

- логіка фронтенду — 7 помилок (38.9 %)
- невідповідність клієнт-сервер — 5 помилок (27.8 %)
- відсутність валідації на сервері — 4 помилки (22.2 %)
- авторизація — 2 помилки (11.1 %)

Для визначення частоти помилок у системі використано таку формулу:

$$F = \frac{N_{\text{помилок}}}{N_{\text{загальна}}} \cdot 100\backslash\% \qquad (3.2)$$

де

$N_{\text{помилок}}$ – кількість тестів, які завершилися з помилками;
$N_{\text{загальна}}$ – загальна кількість виконаних тестів.

Після підставлення значень отримаємо:

$$F = \frac{18}{92} \cdot 100\backslash\% \approx 19{,}57\backslash\%$$

Для оцінки рівня стабільності API-інтерфейсу застосовано метрику, що враховує кількість стабільно працюючих маршрутів (що не дали збоїв у жодному з запусків) до загальної кількості:

$$S_{\text{API}} = \frac{N_{\text{стабільних}}}{N_{\text{REST}}} \cdot 100\backslash\%\,, \tag{3.3}$$

де

$N_{\text{стабільних}}$ - кількість стабільно виконаних REST-запитів без помилок;
N_{REST} - загальна кількість виконаних REST-запитів.

За результатами тестування, 21 із 24 ендпоінтів працюють стабільно:

$$S_{\text{API}} = \frac{21}{24} \cdot 100\backslash\% = 87{,}5\backslash\%$$

Результати тестування свідчать про достатню зрілість системи для використання в тестовому або пілотному середовищі. Найбільшу кількість збоїв зафіксовано в модулях взаємодії з WebSocket та формування маркерів у специфічних edge-case сценаріях (наприклад, видалення маркера одночасно з оновленням через інший клієнт). Пріоритет подальшої роботи спрямований на доповнення серверної валідації, покращення логування помилок і розширення автоматизованого тестового покриття.

3.2 Результати тестування і вразливості безпеки

У процесі верифікації безпеки web-застосунку було проведено окреме тестування, спрямоване на виявлення вразливостей, пов’язаних з автентифікацією, доступом до ресурсів, обробкою вхідних даних, а також взаємодією між клієнтом і сервером. Загалом було перевірено 60 ключових сценаріїв взаємодії, які потенційно можуть містити критичні або середні за рівнем ризику уразливості. Методика перевірки включала ручне тестування (з підміною токенів, некоректними заголовками), використання Postman, а також базову перевірку на типові загрози OWASP: ін’єкції, міжсайтове виконання скриптів (XSS), повторне використання токенів, неправильну обробку сесій тощо.

Загальна кількість зафіксованих уразливостей становила:

$$V_{\text{загальна}} = 11$$

З них:

- критичних — 2,
- середнього рівня — 5,
- низького рівня — 4.

Для обчислення коефіцієнта захищеності системи використовуємо співвідношення кількості перевірених безпечних сценаріїв до загальної кількості перевірених:

$$S_{\text{захист}} = \frac{N_{\text{безпечних}}}{N_{\text{перевірених}}} \cdot 100\backslash\%\,, \qquad (3.4)$$

де

$N_{\text{безпечних}}$=49,

$N_{\text{перевірених}}$=60.

Підставивши:

$$S_{\text{захист}} = \frac{49}{60} \cdot 100\backslash\% = 81{,}67\backslash\%$$

Цей показник демонструє, що понад 81 % перевірених взаємодій не містять загроз безпеці.

Також було розраховано частку виявлених уразливостей за рівнями ризику, відповідно до загального числа:

Критичні:

$$P_{\text{critical}} = \frac{2}{60} \cdot 100\backslash\% = 3{,}33\backslash\%$$

Середні:

$$P_{\text{medium}} = \frac{5}{60} \cdot 100\backslash\% = 8{,}33\backslash\%$$

Низькі:

$$P_{\text{low}} = \frac{4}{60} \cdot 100\backslash\% = 6{,}67\backslash\%$$

Найтиповішими уразливостями були:

1. Відсутність перевірки ownership ID під час оновлення маркера через PUT-запит — дозволяло редагувати чужу точку при знанні її ID (критична).
2. Відсутність обмеження кількості запитів на вхід — дозволяло підбирати паролі (середня).
3. Недостатня фільтрація текстових полів опису — можливість вставки HTML (XSS) у коментарях (середня).
4. Передача токена через небезпечні канали (http замість https) — виявлено у тестовому середовищі, усунуто (низька).
5. Недостатнє обмеження розміру зображень у формі AddForm, що могло призвести до навантаження серверної пам’яті (низька).

Після виявлення вразливостей були проведені відповідні коригування:

- реалізовано middleware-перевірку прав доступу (authorId === req.user.id);

- додано обмеження кількості спроб входу (rate-limiting);
- впроваджено HTML-санітизацію на рівні серверної обробки описів і текстових полів;
- забезпечено обов’язковий перехід на HTTPS у всіх середовищах розгортання;
- додано перевірку MIME-типів і обмеження розміру зображень (до 2 МБ).

Також було визначено коефіцієнт усунення вразливостей, як відношення кількості виправлених загроз до виявлених:

$$R_{\text{усунення}} = \frac{V_{\text{усунено}}}{V_{\text{загальна}}} \cdot 100\backslash\% = \frac{11}{11} \cdot 100\backslash\% = 100\backslash\% \qquad (3.5)$$

3.3 Документація для кінцевого користувача

Для забезпечення ефективної взаємодії користувача із системою було розроблено повноцінну документацію, що пояснює всі ключові етапи роботи з веб-застосунком — від реєстрації до взаємодії з картою та обробки особистих даних. Документація оформлена у вигляді покрокових інструкцій і відображає логіку використання інтерфейсу з урахуванням особливостей авторизації, додавання та перегляду маркерів, роботи з формами та підказками. Вона надається у вигляді окремого розділу на сторінці довідки всередині застосунку, а також у PDF-версії для офлайн-доступу.

Після входу до системи через email і пароль (попередньо зареєстрований обліковий запис) користувач автоматично перенаправляється на головну сторінку з інтерактивною картою. У верхній частині інтерфейсу розміщена панель навігації (Navbar), що містить кнопки: «Карта», «Мій профіль», «Вийти». На головному екрані відображається карта з маркерами. Для перегляду інформації про точку достатньо навести курсор на маркер або натиснути на нього — з’являється спливаюче вікно з назвою, описом, категорією, автором і датою створення. Якщо

користувач є автором цього маркера, з’являються кнопки «Редагувати» та «Видалити».

Для створення нової точки достатньо двічі клацнути на карту — автоматично відкриється форма AddForm, у якій користувач має ввести назву місця, опис, категорію з випадаючого списку, координати (автоматично визначаються, але можуть бути змінені вручну), а також завантажити фото. Після натискання кнопки «Створити» маркер додається на карту і стає доступним для перегляду іншим користувачам. Система автоматично виконує перевірку автентифікації, додає JWT-токен до заголовка запиту, а у разі відсутності авторизації перенаправляє користувача на сторінку входу.

Редагування маркера доступне лише автору. Кнопка «Редагувати» відкриває форму UpdateForm, де поля заповнюються автоматично. Після редагування оновлений маркер зберігається, а інші клієнти отримують зміни в реальному часі через WebSocket. Видалення підтверджується окремим діалоговим вікном.

Усі поля форм мають вбудовану валідацію: назва та опис є обов’язковими, обмежено допустиму довжину тексту (до 300 символів), завантаження зображення дозволено лише у форматі JPEG/PNG, не більше 2 МБ. Якщо заповнення форми некоректне — з’являється відповідне повідомлення.

Для користувачів, які забули пароль, реалізована форма ForgotPassword, де необхідно ввести email — на нього надсилається інструкція для відновлення. У профілі користувача доступні функції перегляду особистих даних, а також — у майбутньому — перегляд історії створених маркерів і змін.

Документація доповнена скриншотами інтерфейсу, прикладами заповнених форм і поясненнями до кожного кроку. Також додано інструкції щодо захисту даних: користувач інформується, що пароль зберігається у зашифрованому вигляді, а передача даних відбувається через HTTPS. Усі дані, пов’язані з авторизацією, не зберігаються на клієнті у відкритому вигляді.

Отже, розроблена документація, структура якої зображена на рисунку 3.1, забезпечує повне охоплення всіх користувацьких сценаріїв, спрощує знайомство з

платформою, мінімізує кількість помилок з боку користувачів і сприяє підвищенню довіри до системи.

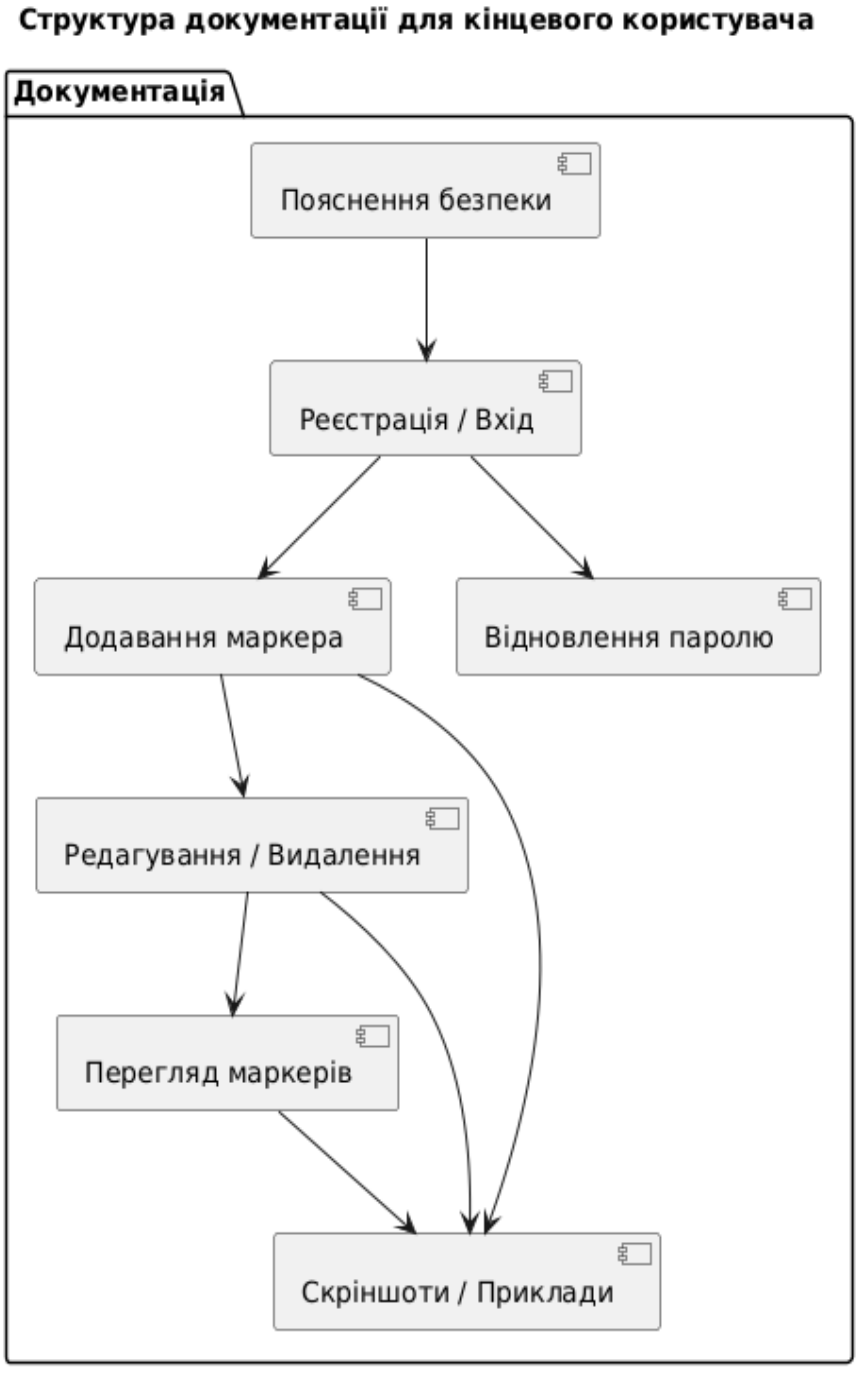


Рис. 3.1 Структура документації для кінцевого користувача

3.4 Перспективи вдосконалення та масштабування системи

З огляду на позитивні результати тестування та успішне впровадження основних функціональних модулів веб-застосунку, подальший розвиток системи передбачає впровадження низки удосконалень, спрямованих на підвищення її функціональної повноти, продуктивності, безпеки та масштабованості. На поточному етапі система забезпечує повноцінну взаємодію користувачів із геопросторовим контентом, дозволяє створювати, редагувати та переглядати маркери на мапі в режимі реального часу, однак структура платформи дозволяє її ефективне розширення без фундаментальних змін у архітектурі.

Одним із основних напрямів вдосконалення є розширення підтримуваних типів контенту. На відміну від поточної реалізації, яка обмежується маркерами з текстовими описами та зображеннями, можливо інтегрувати додаткові об’єкти: маршрути (треки з кількома точками), відеоогляди локацій, голосові коментарі. Це вимагатиме розширення схеми даних у MongoDB, оптимізації зберігання великих медіафайлів (наприклад, через підключення хмарних сервісів з CDN) і модифікації форми введення. Додатково можна реалізувати модуль геофільтрації, який дозволить користувачу отримувати персоналізовані маркери на основі поточного місцезнаходження або обраного регіону.

Іншим стратегічним напрямом розвитку є масштабування архітектури для підтримки великої кількості одночасних користувачів. Поточна реалізація базується на монолітному сервері Express, однак її структура дозволяє легко переходити до мікросервісного або розподіленого підходу. Зокрема, WebSocket-події можуть бути перенесені у окремий сервіс, із використанням Redis або RabbitMQ як брокера повідомлень. Це дозволить забезпечити горизонтальне масштабування (через балансувальники навантаження) без втрати синхронності між клієнтами. Також доцільно впровадити кешування даних за допомогою Redis або Memcached, особливо для запитів до популярних локацій, що дозволить зменшити навантаження на базу даних і пришвидшити рендеринг мапи.

У контексті аналітики користувацької активності перспективним є впровадження модуля статистики: система збиратиме дані про кількість створених маркерів, їх рейтинг, частоту редагувань, а також часові патерни використання. Зібрана інформація може бути агрегована й виведена в адмінпанелі у вигляді графіків, що забезпечить кращий моніторинг системи та прийняття рішень щодо подальших оновлень. Для зберігання метаданих аналітики можливо використати окрему колекцію в MongoDB або сторонню систему (наприклад, InfluxDB).

З точки зору інтерфейсу, перспективи полягають у впровадженні системи адаптивного дизайну з використанням Tailwind CSS або Material UI, що забезпечить зручну роботу з мобільних пристроїв без втрати функціональності. Додатково, можливо впровадити темну тему, підтримку декількох мов (через i18n) та розширену доступність (accessibility), відповідно до стандартів WCAG. Реалізація drag-and-drop для редагування маршруту або маркера, а також можливість залишати відгуки іншим користувачам додадуть платформі елементів соціальної взаємодії.

Важливим етапом розвитку є посилення безпеки та захисту персональних даних. Передбачається додавання двофакторної автентифікації (2FA) через email або мобільні застосунки, а також впровадження обмеження по IP-адресах та захисту від автоматизованих ботів за допомогою CAPTCHA. Крім того, можливим є логування всіх змін у системі (створення, редагування, видалення) з прив’язкою до часу та користувача, що дозволить здійснювати аудит активності та швидко реагувати на аномальні дії.

Загалом розроблена система має достатньо гнучку модульну структуру для подальшого вдосконалення. Її фронтенд легко масштабований завдяки використанню компонентного підходу в React, а бекенд побудовано з урахуванням REST-принципів і стандартів безпеки. Усі запропоновані напрями розвитку можуть бути реалізовані поступово, без зупинки роботи платформи, що дозволяє не лише підтримувати її життєвий цикл, але й перетворити на повноцінний соціальний простір для мандрівників і дослідників.

Висновки до розділу 3

У цьому розділі було проведено комплексне тестування веб-застосунку, розглянуто результати виявлення помилок, оцінено рівень безпеки системи, охарактеризовано надану документацію для кінцевого користувача, а також сформульовано перспективи розвитку та масштабування платформи.

Результати функціонального та інтеграційного тестування засвідчили достатній рівень стабільності системи — понад 80 % тестів пройдено успішно, що є позитивним показником для продукту на етапі пілотного впровадження. Найбільшу кількість помилок виявлено в компонентах реального часу (WebSocket), де були зафіксовані збої у випадках конкурентної взаємодії. Такі результати дозволили визначити пріоритетні напрямки вдосконалення логіки взаємодії клієнтів і серверної валідації.

Перевірка безпеки показала наявність низки уразливостей, зокрема критичних — таких як відсутність перевірки прав доступу до ресурсу. Усі виявлені загрози були оперативно усунені, що дозволило досягти повного (100 %) рівня виправлення. Рівень загального захисту взаємодій системи оцінено на рівні 81,67 %, що свідчить про прийнятну стійкість проти типових загроз OWASP.

Документація користувача представлена у доступному вигляді, охоплює всі основні функціональні сценарії та сприяє швидкому входженню в систему, зменшуючи ймовірність помилок з боку кінцевих користувачів.

Окреслені перспективи вдосконалення демонструють високу адаптивність системи до зростання функціонального навантаження. Запропоновані напрями розвитку охоплюють масштабування, покращення UX/UI, інтеграцію нових типів контенту, підвищення безпеки та розширення соціальної взаємодії. Це формує підґрунтя для подальшої еволюції проєкту в напрямку повноцінного сервісу для обміну геолокаційним досвідом.

ВИСНОВКИ

У підсумку проведеного дослідження було реалізовано повнофункціональний веб-застосунок, що забезпечує можливість обміну туристичним досвідом між користувачами з інтегрованою базою геолокаційних маркерів. На всіх етапах проєктування та розробки було дотримано принципів модульності, масштабованості, безпеки та інтерактивності, що дозволило створити сучасну систему, орієнтовану на реальні потреби цільової аудиторії. Комплексне опрацювання архітектури, вибору технологічного стеку, інтерфейсного дизайну, а також засобів захисту даних забезпечило високий рівень відповідності програмного продукту вимогам сучасних веб-платформ, що працюють із геопросторовою інформацією.

На теоретичному етапі було здійснено порівняльний аналіз існуючих застосунків туристичного спрямування, таких як TripAdvisor, Google Maps, Maps.me, що дозволило ідентифікувати функціональні обмеження наявних сервісів у контексті індивідуального досвіду користувача. Зокрема, більшість рішень орієнтовані на централізовану обробку контенту та мають обмежений рівень персоналізації та інтерактивності. Відсутність можливості швидкого додавання неформальних маркерів, відсутність відкритої структури комунікації в реальному часі та слабкий механізм контролю над вмістом з боку самих користувачів окреслили потребу в створенні системи з акцентом на свободу внесення даних, гнучке керування власними маркерами та оновлення в реальному часі без втрати стабільності.

У результаті було сформульовано функціональні та нефункціональні вимоги до розроблюваної системи, зокрема: підтримка автентифікації та контролю доступу, можливість створення, редагування та видалення маркерів, обробка координат та інтеграція з картою, адаптивність інтерфейсу, захищене зберігання даних, реальна синхронізація змін між клієнтами. Крім того, було враховано вимоги

до масштабованості системи, розширення API, підтримки нових типів контенту в майбутньому, що визначило подальші технічні рішення на етапі проєктування.

Для реалізації обрано стек технологій, який повністю відповідає цілям проєкту: клієнтська частина побудована з використанням **TypeScript** і **React**, що забезпечило статичну типізацію, підвищення читабельності та надійності коду, а також ефективну організацію компонентів.

Серверна частина реалізована на базі **Node.js** з використанням **Express**, що дозволило побудувати легкий та розширюваний REST API. Для зберігання даних використано **MongoDB** із обгорткою **Mongoose**, що дало змогу зберігати напівструктуровані документи з автоматичною валідацією, індексацією та геолокаційною обробкою. Двосторонню синхронізацію клієнтів реалізовано за допомогою **WebSocket** через бібліотеку **socket.io**, що дало змогу досягти високої чутливості до змін в інтерфейсі без повторного завантаження сторінки.

На рівні клієнтської частини були створені інтуїтивно зрозумілі компоненти: панель навігації, карта, форма додавання маркера, спливаючі вікна з інформацією про об’єкти. Усі компоненти взаємодіють із глобальним контекстом автентифікації, зберігають стан користувача та дозволяють керувати доступом до функцій згідно з роллю. Для створення точки на карті використовується подвійний клік, що ініціює відкриття форми з координатами, заповненими автоматично. Кожен маркер містить категорію, опис, дату створення, ідентифікатор автора, а також може бути редагований або видалений тільки його автором.

Серверна логіка реалізує валідацію вхідних даних, перевірку авторизації через JWT, обробку запитів на створення, оновлення та видалення маркерів, а також обробку підключення WebSocket. Захищені маршрути відокремлені middleware-функцією, яка перевіряє наявність токена, його дійсність, а також роль користувача. Для захисту паролів реалізовано хешування через bcrypt, а доступ до бази даних конфігуровано з використанням змінних середовища. Для запобігання SQL-ін’єкціям та XSS-атакам усі вхідні дані проходять фільтрацію.

Результати тестування показали високу стабільність системи. Загальний рівень успішності тестів становив понад 80 %, зокрема, найбільш стабільними виявилися модулі автентифікації та API взаємодії з маркерами.

Було виявлено низку вразливостей, які було повністю усунуто: відсутність обмеження прав доступу до ресурсів, XSS через опис місця, незахищені зображення, а також відсутність обмеження кількості запитів на вхід. Після внесення виправлень рівень усунення склав 100 %, що засвідчує ефективність заходів безпеки. Аналіз продуктивності показав стабільну роботу WebSocket при паралельній взаємодії кількох клієнтів, затримки в оновленні даних не перевищували 250 мс.

Окрема увага була приділена підготовці інтерфейсної документації для кінцевих користувачів. Вона охоплює всі ключові сценарії — від реєстрації до перегляду та редагування маркерів — і надає поетапне пояснення дій, супроводжене прикладами, інструкціями, індикаторами помилок та роз'ясненнями щодо безпеки облікових записів. Це підвищує доступність системи для користувачів без технічного бекґраунду й дозволяє їм ефективно використовувати всі функції застосунку з мінімальними ризиками помилок.

У рамках дослідження було сформульовано також перспективи розвитку та масштабування системи. Їх реалізація включає можливість переходу до мікросервісної архітектури з використанням брокерів повідомлень, розширення функціоналу до підтримки маршрутів і мультимедійного контенту, впровадження системи аналітики активності, додавання drag-and-drop, адаптивного дизайну, багатомовної підтримки та інтеграції з мобільними платформами. Важливим є також подальше зміцнення системи безпеки: впровадження двофакторної автентифікації, захисту від ботів, сесійного контролю.

Узагальнюючи, розроблений застосунок відповідає актуальним вимогам до інтерфейсних та серверних рішень у сфері взаємодії з просторовими даними, забезпечує безпечне й продуктивне середовище для збереження та обміну туристичною інформацією, має потенціал для подальшої інтеграції з іншими

сервісами та масштабування на рівні повноцінної цифрової екосистеми. Проведене дослідження підтвердило ефективність обраного стеку технологій, обґрунтованість архітектурних рішень і доцільність системного підходу до побудови багатофункціонального веб-застосунку.

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

1. Власенко Б.Є., Задонцев Ю.В. Визначення вимог до web-застосунку для обміну туристичним досвідом. Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 24.04.2025, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2025. С. 84-86.
2. Власенко Б.Є., Задонцев Ю.В. Захист інформації у web-застосунку для обміну туристичним досвідом на основі бази відгуків про місця інтересу. XII Всеукраїнська науково-практичній конференція молодих учених «ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ 2025», 15.05.2025, Київ, КСУ імені Бориса Грінченка. Збірник тез. К.: КСУ імені Грінченка, 2025. С. 269-270.

ПЕРЕЛІК ПОСИЛАНЬ

1. Aboalganam K. M., AlFraihat S. F., Tarabieh S. The Impact of User-Generated Content on Tourist Visit Intentions: The Mediating Role of Destination Imagery // *Administrative Sciences*. 2025. 15(4): 117. (дата звернення: 12.05.2025).

2. Ahmad K. MERN TypeScript Setup Guide. DEV Community. URL: https://dev.to/kafeel_ahmad/mern-typescript-setup-guide-4lbf (дата звернення: 12.05.2025).

3. Alwis L. W. Why MERN Developers Love TypeScript. Medium. URL: https://masterlwa.medium.com/why-mern-developers-love-typescript-e75def6e8dc9 (дата звернення: 12.05.2025).

4. Asmare E. Part One: Setting up Express, TypeScript, MongoDB. Medium. URL: https://medium.com/@it.ermias.asmare/part-one-setting-up-express-typescript-mongodb-b77ed847b094 (дата звернення: 12.05.2025).

5. Balaji D. JWT Authentication in Node.js: A Practical Guide. Medium. URL: https://dvmhn07.medium.com/jwt-authentication-in-node-js-a-practical-guide-c8ab1b432a49 (дата звернення: 12.05.2025).

6. Béjar R., Umer M., Martínez-Fernandez J., та ін. A Mobile Application to Share Georeferenced Tourist Experiences on a Discrete Global Grid // *Proceedings of GEOProcessing 2020: The 12th Int. Conf. on Advanced Geographic Information Systems, Applications, and Services*. 2020. – Режим доступу: https://thinkmind.org/articles/geoprocessing_2020_1_140_30091.pdf (дата звернення: 12.05.2025).

7. Crafting a stunning CRUD Application with MERN stack. GeeksforGeeks. URL: https://www.geeksforgeeks.org/crafting-a-stunning-crud-application-with-mern-stack/ (дата звернення: 12.05.2025).

8. Cross-Origin Resource Sharing (CORS) – HTTP // MDN Web Docs. URL: https://developer.mozilla.org/docs/Web/HTTP/CORS (дата звернення: 12.05.2025).

9. Don’t Block the Event Loop (or the Worker Pool). Node.js Official Docs. URL: https://nodejs.org/en/learn/asynchronous-work/dont-block-the-event-loop (дата звернення: 12.05.2025).

10. Error Handling in Express.js. GeeksforGeeks. URL: https://www.geeksforgeeks.org/error-handling-in-express-js/ (дата звернення: 12.05.2025).

11. Express Documentation. OpenJS Foundation. URL: https://expressjs.com/en/starter/installing.html (дата звернення: 12.05.2025).

12. Express + Node.js Handbook. freeCodeCamp. URL: https://www.freecodecamp.org/news/express-js-and-node-js-handbook (дата звернення: 12.05.2025).

13. Express.js and MongoDB REST API Tutorial. MongoDB. URL: https://www.mongodb.com/languages/express-mongodb-rest-api-tutorial (дата звернення: 12.05.2025).

14. Express.js Security Tips: How to secure your Express.js app. freeCodeCamp. URL: https://www.freecodecamp.org/news/express-js-security-tips-how-to-secure-your-app (дата звернення: 12.05.2025).

15. Fielding R. Architectural Styles and the Design of Network-based Software Architectures : дис. … Ph.D. – University of California, Irvine, 2000. – 162 с. (дата звернення: 12.05.2025).

16. Getting Started with MERN Stack. GeeksforGeeks. URL: https://www.geeksforgeeks.org/getting-started-with-mern-stack (дата звернення: 12.05.2025).

17. Hahn D. TripShare – MERN Stack App. GitHub. URL: https://github.com/DavidKHahn/TripShare (дата звернення: 12.05.2025).

18. Health Tracker using MERN Stack. GeeksforGeeks. URL: https://www.geeksforgeeks.org/health-tracker-using-mern-stack (дата звернення: 12.05.2025).

19. Helmet.js – Express security middleware. HelmetJS. URL: https://helmetjs.github.io (дата звернення: 12.05.2025).

20. Henrique J. How to create a React frontend and a Node/Express backend and connect them. freeCodeCamp. URL: https://medium.com/free-code-camp/create-a-react-frontend-a-node-express-backend-and-connect-them-together-c5798926047c (дата звернення: 12.05.2025).

21. How to Build a Todo App with React, TypeScript, Node.js, and MongoDB. freeCodeCamp. URL: https://www.freecodecamp.org/news/how-to-build-a-todo-app-with-react-typescript-nodejs-mongodb (дата звернення: 12.05.2025).

22. Ibrahim M. How to use MERN Stack: A Complete Guide. MongoDB. URL: https://www.mongodb.com/developer/languages/javascript/mern-stack-tutorial (дата звернення: 12.05.2025).

23. Importance of User-Generated Content in Tourism Events for Creating Engagement // *Journal of Business*. 2021. – URL: https://openaccessojs.com/JBReview/article/view/4546 (дата звернення: 12.05.2025).

24. JSON Web Token (JWT) – Introduction. JWT.io (Auth0). URL: https://jwt.io/introduction (дата звернення: 12.05.2025).

25. Manoj H. MERN Stack with TypeScript. SkillVertex. URL: https://www.skillvertex.com/blog/mern-stack-with-typescript (дата звернення: 12.05.2025).

26. Megida D. How to Deploy a MERN Application to Heroku Using MongoDB Atlas. freeCodeCamp. URL: https://www.freecodecamp.org/news/deploying-a-mern-application-using-mongodb-atlas-to-heroku (дата звернення: 12.05.2025).

27. Megida D. Multer: Easily upload files with Node.js and Express. LogRocket Blog. URL: https://blog.logrocket.com/multer-nodejs-express-upload-file (дата звернення: 12.05.2025).

28. Microsoft. TypeScript Documentation. Microsoft. URL: https://www.typescriptlang.org/docs (дата звернення: 12.05.2025).

29. Microsoft. TypeScript-Node-Starter: A reference example for Node.js and TypeScript. GitHub. URL: https://github.com/microsoft/TypeScript-Node-Starter (дата звернення: 12.05.2025).

30. MongoDB Documentation. MongoDB, Inc. URL: https://www.mongodb.com/docs (дата звернення: 12.05.2025).

31. Mongoose Documentation. Mongoose ODM. URL: https://mongoosejs.com/docs (дата звернення: 12.05.2025).

32. Morgan A. The Modern Application Stack – MEAN, MERN, and More. MongoDB (White Paper). 2017. URL: https://www.mongodb.com/mern-stack (дата звернення: 12.05.2025).

33. Music Discovery App with MERN Stack. GeeksforGeeks. URL: https://www.geeksforgeeks.org/music-discovery-app-with-mern-stack (дата звернення: 12.05.2025).

34. Nick. How To Deploy a Full-Stack MERN App with Heroku/Netlify. DEV Community. URL: https://dev.to/stlnick/how-to-deploy-a-full-stack-mern-app-with-heroku-netlify-ncb (дата звернення: 12.05.2025).

35. Node.js Documentation. OpenJS Foundation. URL: https://nodejs.org/en/docs (дата звернення: 12.05.2025).

36. Note-Taking App with Status Tracker using MERN Stack. GeeksforGeeks. URL: https://www.geeksforgeeks.org/note-taking-app-with-status-tracker-using-mern-stack (дата звернення: 12.05.2025).

37. Osiroski. Backend: Nodejs, MongoDB, Express TypeScript. DEV Community. URL: https://dev.to/osiroski/backend-nodejs-mongodb-express-typescript-33m (дата звернення: 12.05.2025).

38. Osiroski. Guide to develop MERN application, step by step. Part 1. DEV Community. URL: https://dev.to/osiroski/guide-to-develop-mern-application-step-by-step-part-1-3o8m (дата звернення: 12.05.2025).

39. Pros and Cons of using MERN Stack in your project. Laxaar. URL: https://laxaar.com/blog/pros-and-cons-of-using-mern-stack-in-your-project-1677657062536 (дата звернення: 12.05.2025).

40. React Documentation. Meta. URL: https://react.dev/learn (дата звернення: 12.05.2025).

41. React Router Documentation. Remix Software. URL: https://reactrouter.com/en/main (дата звернення: 12.05.2025).

42. SQL vs NoSQL: 5 Critical Differences. Integrate.io Blog. URL: https://www.integrate.io/blog/sql-vs-nosql-differences(дата звернення: 12.05.2025).

43. Task Manager App using MERN Stack. GeeksforGeeks. URL: https://www.geeksforgeeks.org/task-manager-app-using-mern-stack (дата звернення: 12.05.2025).

44. Travel Journal App with MERN Stack with API. GeeksforGeeks. URL: https://www.geeksforgeeks.org/travel-journal-app-with-mern-stack-with-api (дата звернення: 12.05.2025).

45. User-Generated Content: A Destination Marketer’s Secret Weapon. Advance Travel & Tourism. URL: https://www.advancetravelandtourism.com/blog/user-generated-content-destination-marketers-secret-weapon (дата звернення: 12.05.2025).

46. Using promises – JavaScript. MDN Web Docs. URL: https://developer.mozilla.org/docs/Learn/JavaScript/Asynchronous/Promises (дата звернення: 12.05.2025).

47. VimukthiMK. Map-Explore – MERN Stack Travel App. GitHub. URL: https://github.com/VimukthiMK/Map-Explore(дата звернення: 12.05.2025).

48. When to Use NoSQL and MongoDB. Fivetran Blog. URL: https://www.fivetran.com/blog/when-to-use-nosql-and-mongodb (дата звернення: 12.05.2025).

49. Yadav A. Travel Nest – Hotel Booking MERN Website. GitHub. URL: https://github.com/webdevankit07/hotel-booking-app (дата звернення: 12.05.2025).

50. Yongsiriwit K. Creating a simple REST API using Express.js + MongoDB. Medium. URL: https://javascript.plainenglish.io/creating-a-simple-rest-api-using-express-js-mongodb-c7f53d145e4d (дата звернення: 12.05.2025).

ДОДАТОК А. ДЕМОНСТАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка Full-stack застосунку для обміну туристичним досвідом на основі бази відгуків про місця інтересу

Виконав студент 4 курсу
групи ПД-41
Власенко Богдан Євгенійович
Керівник роботи
к.т.н., доц., доцент кафедри ІПЗ Задонцев Юрій Вікторович

Київ – 2025

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: допомога та спрощення обміну туристичним досвідом на базі web-застосунку.
Об'єкт дослідження: процес обміну інформацією між користувачами туристичних платформ.
Предмет дослідження: веб-застосунки з реалізацією обміну туристичним досвідом

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. *Провести аналіз сучасних web-застосунків, що реалізують функціонал обміну туристичними враженнями, і виявити їх ключові особливості та недоліки.*
2. *Визначити потреби кінцевих користувачів і сформулювати функціональні та нефункціональні вимоги до системи.*
3. *Обґрунтувати вибір інструментів і технологій для реалізації клієнтської та серверної частин застосунку.*
4. *Розробити архітектуру програмного забезпечення, що поєднує REST API та WebSocket для підтримки синхронізації даних.*
5. *Реалізувати механізми автентифікації та захисту даних користувачів з використанням JWT та алгоритмів хешування.*
6. *Побудувати інтерфейс користувача з інтерактивною картою на основі Mapbox, забезпечивши можливість додавання, редагування та перегляду маркерів.*
7. *Провести тестування застосунку, виявити потенційні помилки та оцінити стійкість до типових вразливостей.*
8. *Розробити пропозиції щодо вдосконалення та масштабування системи, враховуючи перспективи інтеграції додаткового функціоналу.*

3

АНАЛІЗ АНАЛОГІВ

Платформа	Основне призначення	Функціональні можливості	UX / Інтерфейсні особливості	Архітектура / Технічна база	Основні переваги	Основні недоліки
TripAdvisor	Відгуки, рейтинги та бронювання туристичних об'єктів	Рецензії, фото, оцінки, пошук, фільтрація, персональні списки, пряме бронювання	Каталожний інтерфейс, орієнтований на текст; мобільна версія менш оптимізована	Централізована архітектура з REST API; пошукові бази типу Elasticsearch	Велика база користувацького контенту, багаторівнева оцінка, підтримка бронювання	Обмежена інтерактивність; наявність рекламних пріоритетів; перевантаженість UI
Google Maps	Геонавігація та рекомендації місць інтересу	Прокладання маршрутів, відгуки, фото, Street View, популярність, інтеграція з іншими сервісами Google	Інтерактивний багатошаровий інтерфейс, адаптивність, персоналізація	Мікросервісна архітектура; GCP; масштабованість; кешування; AI-аналітика	Потужна навігація, зручна інтеграція, офлайн-режим, персоналізовані поради	Надмірна складність; реклама; платна модель API при великому навантаженні
Maps.me	Офлайн-навігація для туристів	Завантаження карт, пошук POI, прокладання маршрутів, закладки	Мінімалістичний дизайн, простота, оптимізовано під офлайн	Монолітна офлайн-архітектура на OpenStreetMap; локальне збереження даних	Повноцінна робота без інтернету, простота, висока швидкість	Відсутність коментарів, рейтингів, слабка персоналізація, відсутність API
Travel map	Обмін туристичним досвідом через відгуки на мапі	Додавання, видалення, редагування, перегляд маркерів, оновлення в real-time	Мінімалістичний інтерфейс, адаптивність	Монолітна архітектура, REST API, WEBSOCKET, збереження даних в MongoDB	Висока швидкість, простота використання за рахунок відсутності перевантаження функціями	Відсутність великої бази точок інтересу, за рахунок наразі невеликої популярності застосунку

4

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги:

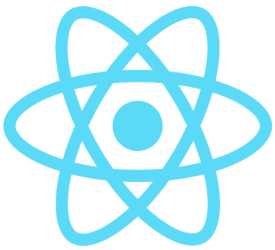
- Реєстрація та авторизація користувача з використанням токенів.
- Інтерактивна карта
- Додавання, перегляд та фільтрація відгуків на карті.
- Збереження контенту (оцінки, текст, геолокація) в базі MongoDB.
- Автоматичне оновлення мапи у режимі реального часу.

Нефункціональні вимоги:

- Захист даних через HTTPS, bcrypt, JWT.
- Стабільна робота при збільшенні кількості користувачів до 1000
- Адаптивний дизайн, коректне відображення на мобільних пристроях та комп'ютерах
- Коректна робота без візуальних дефектів для різних браузерів: Chrome, Firefox, Safari, Edge

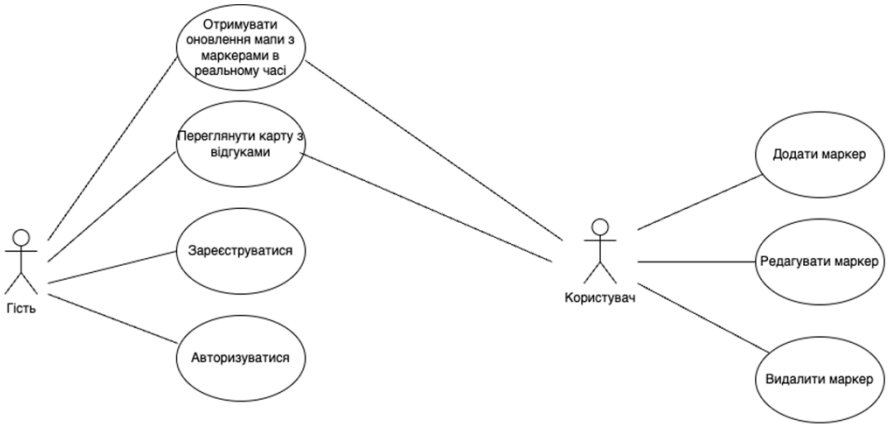
5

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ



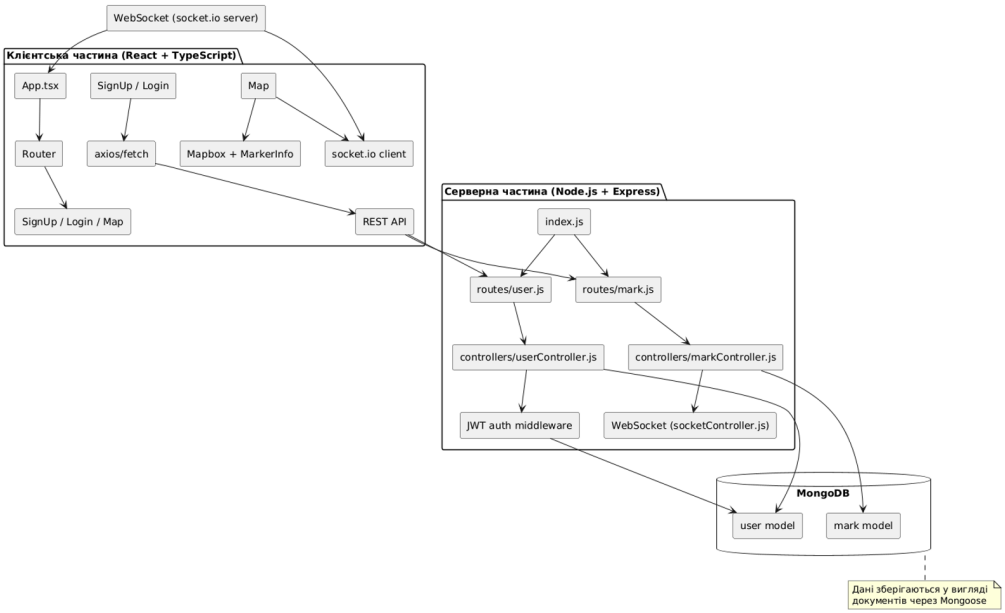
6

Діаграма варіантів використання

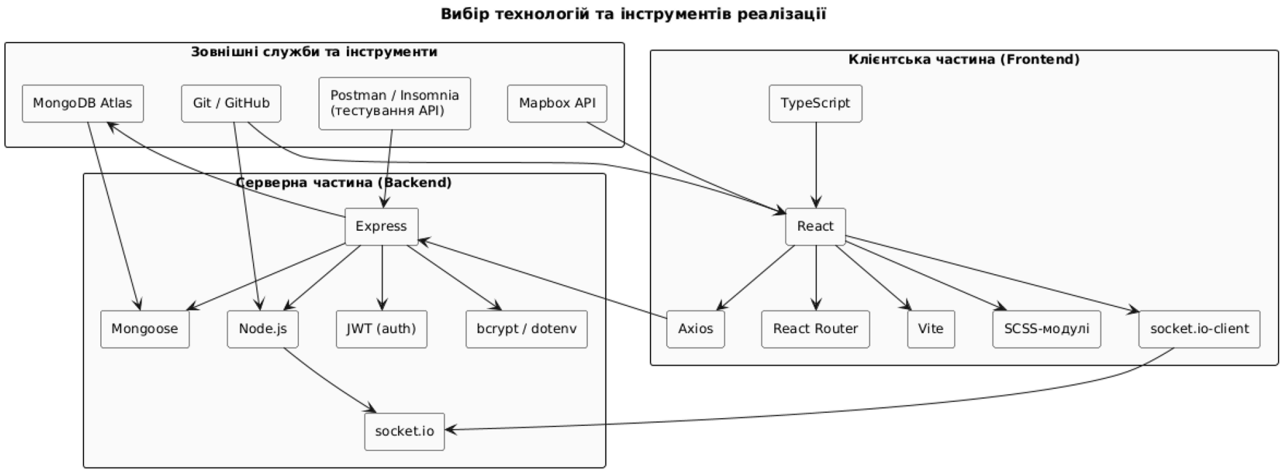


7

Архітектура програмної системи



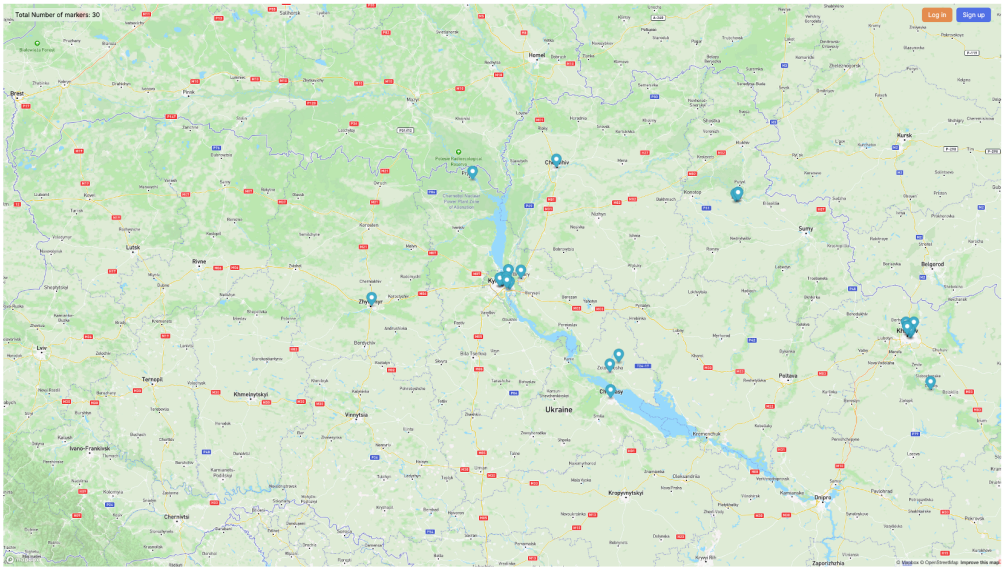
8



9

ЕКРАННІ ФОРМИ

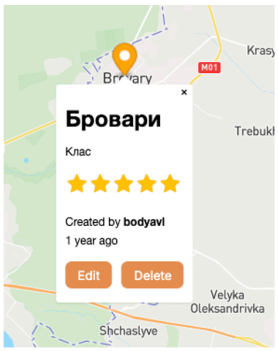
Головний екран



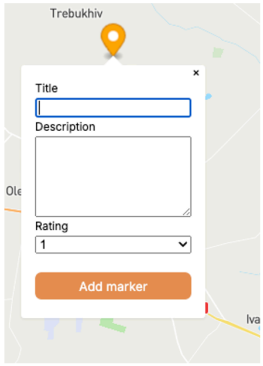
10

ЕКРАННІ ФОРМИ

Перегляд інформації про маркер



Створення нового маркеру



11

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в ІКТ» ВИЗНАЧЕННЯ ВИМОГ ДО WEB-ЗАСТОСУНКУ ДЛЯ ОБМІНУ ТУРИСТИЧНИМ ДОВСВІДОМ , 24 квітня 2025 року с. 84-86.
2. XII Всеукраїнській науково-практичній конференції молодих учених «ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ 2025» ТЕМА: ЗАХИСТ ІНФОРМАЦІЇ У WEB-ЗАСТОСУНКУ ДЛЯ ОБМІНУ ТУРИСТИЧНИМ ДОСВІДОМ НА ОСНОВІ БАЗИ ВІДГУКІВ ПРО МІСЦЯ ІНТЕРЕСУ , 15 травня 2025 року с.269-270.

12

ВИСНОВКИ

1. Проведено аналіз сучасних web-застосунків, що реалізують функціонал обміну туристичними враженнями, і виявлено їх ключові особливості та недоліки.
2. Визначено потреби кінцевих користувачів і сформульовано функціональні та нефункціональні вимоги до системи.
3. Розроблено архітектуру програмного забезпечення, що поєднує REST API та WebSocket для підтримки синхронізації даних.
4. Реалізовані механізми автентифікації та захисту даних користувачів з використанням JWT та алгоритмів хешування.
5. Побудовано інтерфейс користувача з інтерактивною картою на основі Mapbox, забезпечена можливість додавання, редагування та перегляду маркерів.
6. Проведено тестування застосунку, виявлено та виправлено помилки.
7. Розгорнуто застосунок на платформі Render, який працює відповідно до вимог

ДОДАТОК Б. ЛІСТИНГ ПРОГРАМНОГО КОДУ

SignUp.tsx

```
import s from './SignUp.module.scss'
import {useState, FormEvent} from 'react'
import { useNavigate } from 'react-router-dom'
import { signup } from '../../services';

const SignUp = () => {
   const navigate = useNavigate();
   const [email, setEmail] = useState('');
   const [username, setUsername] = useState('');
   const [password, setPassword] = useState('');
   const [isLoading, setIsLoading] = useState(false);


   async function handleSubmit(e: FormEvent<HTMLFormElement>) {
      setIsLoading(true);
      e.preventDefault();
      let res = await signup(email, password, username);
      if(res.status === 400) {
         alert("User with this username already exists!")
         setIsLoading(false);
         return;
      }
      let result = await res.json();
      localStorage.setItem('accessToken', result.accessToken)
      localStorage.setItem('username', result.username);
      localStorage.setItem('refreshToken', result.refreshToken)
      setIsLoading(false);
      navigate('/')
   }


   return (
      <>
         <form onSubmit={handleSubmit} className={s.formContainer}>
            <label htmlFor="email" className={s.formLabel}>Email</label>
            <input type="email" id='email' className={s.formTextInput} name='email' onChange={e => setEmail(e.target.value)} disabled={isLoading} required/>
            <label htmlFor="username" className={s.formLabel}>Username</label>
            <input type="text" id='username' className={s.formTextInput} name='username' onChange={e => setUsername(e.target.value)} disabled={isLoading} required/>
            <label htmlFor="password" className={s.formLabel}>Password</label>
            <input type="password" id='password' className={s.formTextInput} name='password' onChange={e => setPassword(e.target.value)} disabled={isLoading} required/>
            <button className={s.formButton} disabled={isLoading}>Sign up</button>
         </form>
      </>
   )
}

export default SignUp
```

Login.tsx

```
import s from './Login.module.scss'
import { FormEvent, useState } from 'react'
import { Link, useNavigate } from 'react-router-dom';
import { login } from '../../services';

const Login = () => {
   const navigate = useNavigate();
   const [email, setEmail] = useState('');
   const [password, setPassword] = useState('');
   const [isLoading, setIsLoading] = useState(false);

   async function handleSubmit(e: FormEvent<HTMLFormElement>) {
      setIsLoading(true);
      e.preventDefault();
      let res = await login(email, password)
      if(res.status === 500) {
         alert("User with this email doesn't exist!")
         setIsLoading(false);
         return;
      }
      else if(res.status === 403) {
         alert("Wrong password!")
         setIsLoading(false);
         return;
      }
      let result = await res.json();
      localStorage.setItem('accessToken', result.accessToken)
      localStorage.setItem('username', result.username);
      localStorage.setItem('refreshToken', result.refreshToken)
      setIsLoading(false);
      navigate('/');
   }


   return (
      <>
         <form onSubmit={handleSubmit} className={s.formContainer}>
            <label htmlFor="email" className={s.formLabel}>Email</label>
            <input type="email" id='email' className={s.formTextInput} name='email' onChange={e => setEmail(e.target.value)} disabled={isLoading} required/>
            <label htmlFor="password" className={s.formLabel}>Password</label>
            <input type="password" id='password' className={s.formTextInput} name='password' onChange={e => setPassword(e.target.value)} disabled={isLoading} required/>
            <button className={s.formButton} disabled={isLoading}>Login</button>
         </form>
         <Link to={'/forgotPassword'}></Link>
      </>
   )
}
```

```
export default Login;
```

ForgotPassword.tsx

```
import s from './ForgotPassword.module.scss'
import React, { FormEvent, useState } from 'react'

const ForgotPassword = () => {

  const [isLoading, setIsLoading] = useState(false);
  const [email, setEmail] = useState('');


  async function handleSubmit(e: FormEvent<HTMLFormElement>) {
    setIsLoading(true);

    setIsLoading(false);
  }

  return (
    <>
    <form onSubmit={handleSubmit} className={s.formContainer}>
      <label htmlFor="email" className={s.formLabel}>Email</label>
      <input type="email" id='email' className={s.formTextInput} name='email' onChange={e => setEmail(e.target.value)} disabled={isLoading} required/>
      <button className={s.formButton} disabled={isLoading}>Change Password</button>
    </form>
</>
  )
}

export default ForgotPassword
```

AddForm.tsx

```
import { FormEvent, useState } from "react";
import s from "./AddForm.module.scss";
import { Socket } from "socket.io-client";
import { addMarker, getNewTokens } from "../../../services";

interface IMapMarker {
  latitude: number;
  longitude: number;
  rating: number;
  title: string;
  description: string;
  username: string;
  createdAt: Date;
  updatedAt: Date;
  _id: string;
}
interface Position {
  lng: number;
  lat: number;
}
interface IAddFormProps {
  socket: Socket
  addMarkerToArray: (value: IMapMarker) => void;
  newPosition: Position;
  updateNewPosition: (value: Position | null) => void;
}

const AddForm = ({
  socket,
  addMarkerToArray,
  newPosition,
  updateNewPosition,
}: IAddFormProps) => {
  const [title, setTitle] = useState("");
  const [description, setDescription] = useState("");
  const [rating, setRating] = useState(1);

  async function handleAddMarkerSubmit(e: FormEvent<HTMLFormElement>) {
    e.preventDefault();
    const data = {
      latitude: newPosition?.lat,
      longitude: newPosition?.lng,
      title,
      description,
      rating,
      username: localStorage.username,
    };
    let res = await addMarker(data);
    if (res.status === 403) {
      if (await getNewTokens()) {
        res = await addMarker(data);
      }
      else return alert("Error occured");
    }
    let result: IMapMarker = await res.json();
    addMarkerToArray(result);
    socket.emit('new marker', result._id)
    updateNewPosition(null);
    clearValues();
  }
  async function clearValues() {
    setTitle("");
    setDescription("");
    setRating(1);
  }

  return (
    <form onSubmit={handleAddMarkerSubmit}>
      <label htmlFor="title" className={s.formLabel}>
        Title
      </label>
      <input
        type="text"
        id="title"
        className={s.formTextInput}
        name="title"
        onChange={(e) => setTitle(e.target.value)}
        required
      />
      <label htmlFor="descr" className={s.formLabel}>
        Description
      </label>
      <textarea
        name="description"
        id="descr"
        className={s.formTextarea}
        cols={20}
        rows={5}
        onChange={(e) => setDescription(e.target.value)}
        required
      ></textarea>
      <label htmlFor="rating" className={s.formLabel}>
        Rating
      </label>
      <select
```

```
            name="rating"
            id="rating"
            className={s.formSelect}
            onChange={(e)                                    =>
        setRating(parseInt(e.target.value))}
          >
            <option value={1}>1</option>
            <option value={2}>2</option>
            <option value={3}>3</option>
            <option value={4}>4</option>
            <option value={5}>5</option>
          </select>{" "}
          <br />
          <button                  className={s.formButton}>Add
        marker</button>
        </form>
      );
    };

    export default AddForm;
```

ButtonPopup.tsx

```
import { ReactNode } from 'react'
import s from './ButtonPopup.module.scss'

interface IButtonPopupProps {
   handleClick: () => void | Promise<void> ,
   children: ReactNode
}
const      ButtonPopup      =      ({handleClick,      children}:
IButtonPopupProps) => {
 return (
   <button                                  className={s.button}
onClick={handleClick}>{children}</button>
 )
}

export default ButtonPopup
```

MapMarker.tsx

```
import { MapboxEvent, Marker, Popup } from "react-map-gl";
import UpdateForm from "../UpdateForm/UpdateForm";
import MarkerInfo from "../MarkerInfo/MarkerInfo";
import { Socket } from "socket.io-client";

interface IMapMarker {
 latitude: number;
 longitude: number;
 rating: number;
 title: string;
 description: string;
 username: string;
 createdAt: Date;
 updatedAt: Date;
 _id: string;
}
interface IMapMarkerProps {
 socket: Socket,
 marker: IMapMarker;
 isUpdating: boolean;
 updateMarkerInArray: (value: IMapMarker) => void;
 deleteMarkerInArray: (id: string) => void;
 updateIsUpdating: (value: boolean) => void;
 handleClick: (
   e: MapboxEvent<MouseEvent>,
   id: string,
   longitude: number,
   latitude: number
 ) => void;
 currentPositionId: string | null;
 updateCurrentPositionId: (value: string | null) => void;
}

const MapMarker = ({
 socket,
 marker,
 updateMarkerInArray,
 deleteMarkerInArray,
 isUpdating,
 updateIsUpdating,
 handleClick,
 currentPositionId,
 updateCurrentPositionId,
}: IMapMarkerProps) => {
 return (
   <>
     <Marker
       longitude={marker.longitude}
       style={{ cursor: "pointer" }}
       color={
         localStorage.username      &&      marker.username      ===
localStorage.username
           ? "orange"
           : ""
       }
       onClick={(e) =>
         handleClick(e,          marker._id,          marker.longitude,
marker.latitude)
       }
       latitude={marker.latitude}
     ></Marker>
     {marker._id === currentPositionId && (
       <Popup
         latitude={marker.latitude}
         onClose={() => [
           updateCurrentPositionId(null),
           updateIsUpdating(false),
         ]}
         longitude={marker.longitude}
         anchor="top"
       >
         {isUpdating ? (
           <UpdateForm
             socket={socket}
             updateIsUpdating={updateIsUpdating}
             updateMarkerInArray={updateMarkerInArray}
             id={marker._id}
             initialTitle={marker.title}
             initialDescription={marker.description}
             initialRating={marker.rating}
           />
         ) : (
           <MarkerInfo
             socket={socket}
             deleteMarkerInArray={deleteMarkerInArray}
             id={marker._id}
             title={marker.title}
             description={marker.description}
             rating={marker.rating}
             username={marker.username}
             date={
               marker.updatedAt === marker.createdAt
                 ? marker.createdAt
                 : marker.updatedAt
             }
```

```
            isUpdated={Boolean(marker.updatedAt !==
marker.createdAt)}
            updateIsUpdating={updateIsUpdating}
           />
         )}
       </Popup>
     )}
   </>
  );
};

export default MapMarker;
```

Mapbox.tsx

```
import { MapMouseEvent } from "mapbox-gl";
import Map, { MapRef, MapboxEvent } from "react-map-gl";
import "mapbox-gl/dist/mapbox-gl.css";
import { useState, useEffect, useRef } from "react";
import s from "./Mapbox.module.scss";
import Navbar from "../Navbar/Navbar";
import MapMarker from "../MapMarker/MapMarker";
import NewMapMarker from
"../NewMapMarker/NewMapMarker";
import { io } from "socket.io-client";
import { getMarkers } from "../../../services";
const socket = io(import.meta.env.VITE_API_URL);

interface IMapMarker {
  latitude: number;
  longitude: number;
  rating: number;
  title: string;
  description: string;
  username: string;
  createdAt: Date;
  updatedAt: Date;
  _id: string;
}
interface Position {
  lng: number;
  lat: number;
}

const Mapbox = () => {
  const [viewState, setViewState] = useState({
    longitude: 30.523333,
    latitude: 50.45,
    zoom: 7,
  });
  const mapRef = useRef<MapRef>(null);
  const [currentPositionId, setCurrentPositionId] =
useState<string | null>(
    null
  );
  const [newPosition, setNewPosition] = useState<Position |
null>(null);
  const [markers, setMarkers] = useState<IMapMarker[]>([]);

  const [isLoading, setIsLoading] = useState(false);
  const [isUpdating, setIsUpdating] = useState(false);

  const fetchMarkers = async () => {
    setIsLoading(true);
    if(markers) setMarkers([]);
    let result = await (await getMarkers()).json();
    if (result) setMarkers(result);
    setIsLoading(false);
  }

  function addMarkerToArray(value: IMapMarker) {
    setMarkers([...markers, value]);
  }
  function updateMarkerInArray(value: IMapMarker) {
    setMarkers([
      ...markers?.filter((marker: IMapMarker) => marker._id !==
value._id),
      value,
    ]);
  }
  function deleteMarkerInArray(id: string) {
    setMarkers([...markers?.filter((marker: IMapMarker) =>
marker._id !== id)]);
  }

  useEffect(() => {
    fetchMarkers();
  }, []);

  useEffect(() => {
    if(markers) {
      socket.on("fetch new", addMarkerToArray);
      socket.on("fetch update", updateMarkerInArray);
      socket.on("fetch delete", deleteMarkerInArray);
    }
  }, [markers]);

  function handleMarkerClick(
    e: MapboxEvent<MouseEvent>,
    id: string,
    longitude: number,
    latitude: number
  ) {
    e.originalEvent.stopPropagation();
    setCurrentPositionId(id);
    mapRef.current?.flyTo({ center: [longitude, latitude],
duration: 1000 });
  }

  function handleMapDbClick(e: MapMouseEvent) {
    if (!localStorage.username) {
      alert("Log in to add new markers!");
      return;
    }
    const position = { ...e.lngLat };
    setNewPosition(position);
    mapRef.current?.flyTo({
      center: [position.lng, position.lat],
      duration: 1000,
    });
  }
  return (
    <div className={s.map_container}>
      <>
        <Map
          ref={mapRef}

mapboxAccessToken={import.meta.env.VITE_MAPBOXTOK
EN}
          {...viewState}
          onMove={(evt) => setViewState(evt.viewState)}
          onDblClick={handleMapDbClick}
          onClick={() => setNewPosition(null)}
          doubleClickZoom={false}
          mapStyle="mapbox://styles/mapbox/streets-v12"
        >
          {markers?.map((marker: IMapMarker, index: number) =>
(
```

```
          <MapMarker
            socket={socket}
            updateMarkerInArray={updateMarkerInArray}
            deleteMarkerInArray={deleteMarkerInArray}
            key={index}
            marker={marker}
            isUpdating={isUpdating}
            updateIsUpdating={setIsUpdating}
            currentPositionId={currentPositionId}
            updateCurrentPositionId={setCurrentPositionId}
            handleClick={handleMarkerClick}
          />
        ))}
        {localStorage.username && newPosition && (
          <NewMapMarker
            socket={socket}
            addMarkerToArray={addMarkerToArray}
            newPosition={newPosition}
            updateNewPosition={setNewPosition}
          />
        )}
      </Map>
      <Navbar markersCount={markers.length}
fetchMarkers={fetchMarkers} isLoading={isLoading}
updateIsLoading={setIsLoading} />
    </>
  </div>
 );
};

export default Mapbox;
```

MarkerInfo.tsx

```
import s from './MarkerInfo.module.scss'
import { AiFillStar } from "react-icons/ai";
import { format } from "timeago.js";
import ButtonPopup from '../ButtonPopup/ButtonPopup';

import { Socket } from 'socket.io-client';
import { deleteMarker, getNewTokens } from '../../../services';

interface IMapMarker {
 latitude: number;
 longitude: number;
 rating: number;
 title: string;
 description: string;
 username: string;
 createdAt: Date;
 updatedAt: Date;
 _id: string;
}

interface IMarkerInfoProps {
   socket: Socket,
   deleteMarkerInArray: (id: string) => void
   id: string,
   title: string,
   description: string,
   rating: number,
   username: string,
   date: Date,
   isUpdated: boolean
   updateIsUpdating: (value: boolean) => void;
}

const MarkerInfo = ({ socket, id, title, description, rating,
username, date, isUpdated, updateIsUpdating,
deleteMarkerInArray}: IMarkerInfoProps) => {
 async function handleDeleteClick() {
  let res = await deleteMarker(id);
  if (res.status === 403) {
   if (await getNewTokens())
    res = await deleteMarker(id);
   else return alert("Error occured");
  }
  let result: IMapMarker = await res.json();
  deleteMarkerInArray(result._id);
  socket.emit('delete marker', result._id);
 }
 return (
  <>
   <h1>{title}</h1>
   <p>{description}</p>
   {Array(5)
    .fill("")
    .map((_, i) => {
     const ratingValue = i + 1;
     return (
      <AiFillStar
       key={i}
       color={ratingValue <= rating ? "#ffc107" : "#e4e5e9"}
       size={25}
      />
     );
    })}
   <p>
    Created by <b>{username}</b>
    <br />
    {isUpdated ? (
     <>Updated {format(date)}</>
    ) : (
     <>{format(date)}</>
    )}
   </p>
   {username === localStorage.username && (
    <div className={s.buttonsContainer}>
     <ButtonPopup
      handleClick={() => updateIsUpdating(true)}
     >
      Edit
     </ButtonPopup>
     <ButtonPopup
      handleClick={() => handleDeleteClick()}
     >
      Delete
     </ButtonPopup>
    </div>
   )}
  </>
 );
};

export default MarkerInfo;
```

Navbar.tsx

```
import { Link } from "react-router-dom";
import s from "./Navbar.module.scss";
import { logout } from "../../../services";
import { RotatingLines } from "react-loader-spinner";

interface INavbarProps {
 markersCount: number;
 updateIsLoading: (value: boolean) => void;
```

```
  fetchMarkers: () => Promise<void>;
  isLoading: boolean;
}

const Navbar = ({ markersCount, fetchMarkers, updateIsLoading, isLoading }: INavbarProps) => {
  async function handleLogout() {
    updateIsLoading(true);
    await logout();
    localStorage.removeItem("username");
    localStorage.removeItem("accessToken");
    localStorage.removeItem("refreshToken");
    await fetchMarkers();
    updateIsLoading(false);
  }
  return (
    <div className={s.navbar}>
      <div className={s.left}>Total Number of markers: {markersCount || 0}</div>
      <div className={s.right}>
        {isLoading && (
          <div className={s.loaderContainer}>
            <RotatingLines strokeColor="black" width="20" />
          </div>
        )}
        {localStorage.username ? (
          <button className={s.loginButton} onClick={handleLogout}>
            Log out
          </button>
        ) : (
          <>
            <Link className={s.loginButton} to={"/login"}>
              Log in
            </Link>
            <Link className={s.signupButton} to={"/signup"}>
              Sign up
            </Link>
          </>
        )}
      </div>
    </div>
  );
};

export default Navbar;
```

NewMapMarker.tsx

```
import { Marker, Popup } from "react-map-gl";
import AddForm from "../AddForm/AddForm";
import { Socket } from "socket.io-client";

interface Position {
  lng: number;
  lat: number;
}
interface IMapMarker {
  latitude: number;
  longitude: number;
  rating: number;
  title: string;
  description: string;
  username: string;
  createdAt: Date;
  updatedAt: Date;
  _id: string;
}
interface INewMapMarkerProps {
  socket: Socket,
  addMarkerToArray: (value: IMapMarker) => void
  newPosition: Position;
  updateNewPosition: (value: Position | null) => void;
}

const NewMapMarker = ({
  socket,
  addMarkerToArray,
  newPosition,
  updateNewPosition,
}: INewMapMarkerProps) => {
  return (
    <>
      <Marker
        longitude={newPosition.lng}
        latitude={newPosition.lat}
        color="orange"
      ></Marker>
      <Popup
        longitude={newPosition.lng}
        latitude={newPosition.lat}
        onClose={() => updateNewPosition(null)}
        anchor="top"
      >
        <AddForm
          socket={socket}
          addMarkerToArray={addMarkerToArray}
          newPosition={newPosition}
          updateNewPosition={updateNewPosition}
        />
      </Popup>
    </>
  );
};

export default NewMapMarker;
```

UpdateForm.tsx

```
import { FormEvent, useState } from "react";
import s from "./UpdateForm.module.scss";
import { Socket } from "socket.io-client";
import { getNewTokens, updateMarker } from "../../../services";

interface IMapMarker {
  latitude: number;
  longitude: number;
  rating: number;
  title: string;
  description: string;
  username: string;
  createdAt: Date;
  updatedAt: Date;
  _id: string;
}
interface IUpdateFormProps {
  socket: Socket,
  updateMarkerInArray: (value: IMapMarker) => void,
  updateIsUpdating: (value: boolean) => void,
  id: string;
  initialTitle: string;
  initialDescription: string;
  initialRating: number;
}

const UpdateForm = ({ socket, updateIsUpdating, updateMarkerInArray, id, initialTitle, initialDescription, initialRating }: IUpdateFormProps) => {
```

```
  const [title, setTitle] = useState(initialTitle);
  const [description, setDescription] = useState(initialDescription);
  const [rating, setRating] = useState(initialRating);

  async function handleUpdateMarkerSubmit(
    e: FormEvent<HTMLFormElement>,
    id: string
  ) {
    e.preventDefault();
    const data = {
      title,
      description,
      rating,
    };
    let res = await updateMarker(data, id);
    if (res.status === 403) {
      if (await getNewTokens()) res = await updateMarker(data, id);
      else return alert("Error occured");
    }
    let result: IMapMarker = await res.json();
    updateMarkerInArray(result);
    socket.emit('update marker', result._id);
    updateIsUpdating(false);
  }
  return (
    <form onSubmit={(e) => handleUpdateMarkerSubmit(e, id)}>
      <label htmlFor="title" className={s.formLabel}>
        Title
      </label>
      <input
        type="text"
        id="title"
        className={s.formTextInput}
        name="title"
        value={title}
        onChange={(e) => setTitle(e.target.value)}
        required
      />
      <label htmlFor="descr" className={s.formLabel}>
        Description
      </label>
      <textarea
        name="description"
        id="descr"
        className={s.formTextarea}
        cols={20}
        rows={5}
        value={description}
        onChange={(e) => setDescription(e.target.value)}
        required
      ></textarea>
      <label htmlFor="rating" className={s.formLabel}>
        Rating
      </label>
      <select
        name="rating"
        id="rating"
        value={rating}
        className={s.formSelect}
        onChange={(e) => setRating(parseInt(e.target.value))}
      >
        <option value={1}>1</option>
        <option value={2}>2</option>
        <option value={3}>3</option>
        <option value={4}>4</option>
        <option value={5}>5</option>
      </select>{" "}
      <br />
      <button className={s.formButton}>Update marker</button>
    </form>
  );
};

export default UpdateForm;
```