НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
В. о. завідувача кафедри
_____ Артем ВОЛОКИТА
(підпис)

“__” _____2026 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою “Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

на тему: Інтерактивний довідник лікарських рослин на основі застосування штучного інтелекту

Виконала: студентка 4 курсу, групи ІМ-21
(шифр групи)

Фесун Анна Валеріївна
(прізвище, ім’я, по батькові) (підпис)

Керівник доцент, к.т.н. Павлов В.Г.
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) Нікольський С.С.
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент професор кафедри ІСТ, д.т.н. Корнієнко Б.Я.
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2026 р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

ЗАТВЕРДЖУЮ
В. о. завідувача кафедри
_____ Артем ВОЛОКИТА
(підпис)

“__” _____2026 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студентки

Фесун Анни Валеріївни

1. Тема проєкту *Інтерактивний довідник лікарських рослин на основі застосування штучного інтелекту*
керівник проєкту Павлов Валерій Георгійович, доцент, к.т.н.,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету 03 червня 2026 року №2055-с

2. Термін здачі студентом закінченого проєкту *8 червня 2026 р.*

3. Вихідні дані до проєкту *технічна документація, теоретичні дані.*

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

Розділ 1. Огляд предметної області та аналіз існуючих рішень

Розділ 2. Аналіз методів та технологій для розробки платформи

Розділ 3. Розробка та реалізація системи

Розділ 4. Тестування та аналіз результатів роботи системи

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) структурна схема системи, функціональна схема (діаграма класів), алгоритм дій програмного забезпечення.

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Нікольський С.С.		

7. Дата видачі завдання «05» січня 2026 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	*12.12.2025-05.01.2026*	
2.	*Вивчення та аналіз завдання*	*06.01.2026-22.02.2026*	
3.	*Розробка архітектури та загальної структури системи*	*23.02.2026-15.03.2026*	
4.	*Розробка структур окремих підсистем*	*16.03.2026-29.03.2026*	
5.	*Програмна реалізація системи*	*30.03.2026-25.04.2026*	
6.	*Оформлення пояснювальної записки*	*26.04.2026-06.06.2026*	
7.	*Захист програмного продукту*	*08.06.2026*	
8.	*Передзахист*	*09.06.2026*	
9.	*Захист*	*16.06.2026*	

Студент-дипломник _____ Анна ФЕСУН
(підпис)

Керівник проєкту _____ Валерій ПАВЛОВ
(підпис)

АНОТАЦІЯ

Дипломний проєкт присвячено проєктуванню та розробці вебзастосунку «Інтерактивний довідник лікарських рослин на основі застосування штучного інтелекту». У роботі досліджено методи автоматичної ідентифікації рослин засобами комп'ютерного зору, можливості великих мовних моделей для генерації структурованих медичних довідок, а також архітектурний патерн cache-aside як спосіб оптимізації звернень до зовнішніх API. Розроблений застосунок надає користувачам можливість визначати вид лікарської рослини за фотографією за допомогою Pl@ntNet API, отримувати згенеровану Gemini API медичну довідку з результатами, збереженими у базі даних для повторного використання, переглядати каталог рослин, позначати місця їх знаходження на інтерактивній карті та відстежувати особисту історію пошуків. Клієнтську частину реалізовано на React із Tailwind CSS та Leaflet.js, серверну – на Node.js з Express, Prisma ORM і PostgreSQL; автентифікацію побудовано на основі JWT.

Ключові слова: вебзастосунок, лікарські рослини, штучний інтелект, комп'ютерний зір, Pl@ntNet API, Gemini API, cache-aside, React, Node.js, PostgreSQL.

ANNOTATION

This bachelor's thesis is dedicated to the design and development of the web application "Interactive Medicinal Plant Guide Based on Artificial Intelligence". The work investigates methods of automatic plant identification using computer vision, the capabilities of large language models for generating structured medical information, and the cache-aside architectural pattern as a means of optimising external API calls. The developed application enables users to identify medicinal plant species from photographs via the Pl@ntNet API, receive Gemini API-generated medical information cached in the database for reuse, browse a plant catalogue, mark plant locations on an interactive map, and track their personal search history. The frontend is implemented using React with Tailwind CSS and Leaflet.js; the backend is built on Node.js with Express, Prisma ORM and PostgreSQL, with JWT-based authentication.

Keywords: web application, medicinal plants, artificial intelligence, computer vision, Pl@ntNet API, Gemini API, cache-aside, React, Node.js, PostgreSQL.

Справки	Формат	*Значення*	*Найменування*	Кіл. листів	№ екземпляра	*Додаток*
			Документація загальна			
			Знову розроблена			
	А4	***ІАЛЦ.467200.002 ТЗ***	Інтерактивний довідник	4		
			лікарських рослин на основі			
			застосування штучного			
			інтелекту			
			Технічне завдання			
	А4	***ІАЛЦ.467200.003 ПЗ***	Інтерактивний довідник	114		
			лікарських рослин на основі			
			застосування штучного			
			інтелекту			
			Пояснювальна записка			
	А4	***ІАЛЦ.467200.004 Д1***	Інтерактивний довідник	1		
			лікарських рослин на основі			
			застосування штучного			
			інтелекту			
			Структурна схема системи			
	А4	***ІАЛЦ.4672008.005 Д2***	Інтерактивний довідник	1		
			лікарських рослин на основі			
			застосування штучного			
			інтелекту			
			ER-діаграма бази даних			
			системи			
	А4	***ІАЛЦ.467200.006 Д3***	Інтерактивний довідник	1		
			лікарських рослин на основі			
			застосування штучного			
			інтелекту			
			Алгоритм дій користувача в			
			системі			
	А4	***ІАЛЦ.467200.007 Д4***	Інтерактивний довідник	47		
			лікарських рослин на основі			
			застосування штучного			
			інтелекту			
			Текст програмного коду			

Зм.	Арк.	№ докум.	Підпис	Дата	ІАЛЦ.467200.001 ОА		
Розробила		Фесун А.В.			Інтерактивний довідник лікарських рослин на основі застосування штучного інтелекту Опис альбому	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Павлов В.Г.					
Реценз.		Корнієнко Б.Я.				КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-21	
Н. Контр.		Нікольський С.С.					
Затвердив		Волокита А.М.					

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: *Інтерактивний довідник лікарських рослин на основі застосування штучного інтелекту*

Київ – 2026

ЗМІСТ

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Технічне завдання поширюється на розробку вебзастосунку «Інтерактивний довідник лікарських рослин на основі застосування штучного інтелекту» в межах дипломної роботи бакалавра.

Областю застосування є цифрова медицина та фітотерапія: застосунок орієнтований на широку аудиторію користувачів, які прагнуть ідентифікувати лікарські рослини, отримати структуровану медичну інформацію про їхні властивості та зафіксувати місця знаходження рослин на інтерактивній карті.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки є індивідуальне завдання на виконання кваліфікаційної роботи освітнього ступеня «бакалавр інженерії програмного забезпечення», затверджене кафедрою обчислювальної техніки факультету інформатики та обчислювальної техніки Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського».

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою роботи є проєктування та реалізація вебзастосунку, що поєднує автоматичну ідентифікацію лікарських рослин за фотографією засобами комп'ютерного зору, генерацію структурованих медичних довідок із застосуванням великих мовних моделей, інтерактивну карту знаходжень рослин та персональну історію пошуків.

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелами розробки даного дипломного проєкту є офіційна документація використовуваних технологій та сервісів, науково-технічна література з питань комп'ютерного зору, генеративного штучного інтелекту та

проєктування вебсистем, а також відповідні публікації та матеріали мережі Інтернет.

5 ТЕХНІЧНІ ВИМОГИ

5.1 Вимоги до розробленого продукту

Розроблена система має виконувати такі вимоги:

- Зрозумілий та адаптивний інтерфейс з підтримкою мобільних і десктоп-пристроїв.
- Реєстрація та автентифікація користувачів через email і пароль із забезпеченням захисту сесій.
- Визначення виду лікарської рослини за завантаженою фотографією з відображенням ступеня впевненості результату
- Автоматично згенерована медична довідка з лікарськими властивостями, протипоказаннями та способами приготування.
- Кешування результатів AI-аналізу у базі даних за латинською назвою виду для уникнення повторних звернень до зовнішніх сервісів.
- Каталог лікарських рослин із текстовим пошуком та фільтрацією за ознакою лікарська.
- Інтерактивна карта зі спільнотними мітками знаходжень рослин та можливістю додавати власні за GPS-координатами або вручну.
- Збереження та перегляд особистої історії пошукових запитів у профілі користувача.

5.2 Вимоги до програмного забезпечення

- Windows, macOS або Linux.
- Node.js версії 18 або вище.
- PostgreSQL версії 14 або вище.
- Сучасний браузер.

5.3 Вимоги до апаратної частини

- ЦП не менше ніж Intel® Core (TM) i3 або еквівалент.
- ROM не менше ніж 10 ГБ
- RAM не менше ніж 4 ГБ.
- Стабільне підключення до мережі Інтернет.

6 ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	12.12.2025-05.01.2026
Вивчення та аналіз завдання	06.01.2026-22.02.2026
Розробка архітектури та загальної структури системи	23.02.2026-15.03.2026
Розробка структур окремих частин системи	16.03.2026-29.03.2026
Програмна реалізація системи	29.03.2026-20.04.2026
Виправлення помилок	21.04.2026-25.04.2026
Оформлення пояснювальної записки	26.04.2026-06.06.2026

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: *Інтерактивний довідник лікарських рослин на основі застосування штучного інтелекту*

Київ – 2026

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

ACID	Atomicity, Consistency, Isolation, Durability (Властивості транзакцій: атомарність, узгодженість, ізольованість, тривалість)
AI	Artificial Intelligence (Штучний інтелект)
API	Application Programming Interface (Прикладний програмний інтерфейс)
БД	База даних
BSON	Binary JavaScript Object Notation (Бінарна нотація об'єктів JavaScript)
CNN	Convolutional Neural Network (Згортальна нейронна мережа)
CORS	Cross-Origin Resource Sharing (Спільне використання ресурсів між різними джерелами)
CSS	Cascading Style Sheets (Каскадні таблиці стилів)
DOM	Document Object Model (Об'єктна модель документа)
EAV	Entity-Attribute-Value (Патерн зберігання даних «сутність-атрибут-значення»)
GBIF	Global Biodiversity Information Facility (Глобальна інформаційна система з біорізноманіття)
ГІС	Геоінформаційна система
GIN	Generalized Inverted Index (Узагальнений інвертований індекс)
gRPC	Google Remote Procedure Call (Фреймворк віддалених викликів процедур від Google)
HMAC	Hash-based Message Authentication Code (Код автентифікації повідомлення на основі хешу)
HTTP	HyperText Transfer Protocol (Протокол передачі гіпертексту)

HTTPS	HyperText Transfer Protocol Secure (Захищений протокол передачі гіпертексту)
JSON	JavaScript Object Notation (Нотація об'єктів JavaScript)
JWT	JSON Web Token (Вебтокен JSON)
LLM	Large Language Model (Велика мовна модель)
MVP	Minimum Viable Product (Мінімально життєздатний продукт)
MVCC	Multi-Version Concurrency Control (Багатоверсійний контроль паралельного доступу)
OCR	Optical Character Recognition (Оптичне розпізнавання символів)
ORM	Object-Relational Mapping (Об'єктно-реляційне відображення)
PWA	Progressive Web App (Прогресивний вебзастосунок)
ПЗ	Програмне забезпечення
REST	Representational State Transfer (Архітектурний стиль взаємодії компонентів розподіленого застосунку)
RFC	Request for Comments (Запит на коментарі)
SDK	Software Development Kit (Комплект засобів розробки)
SPA	Single Page Application (Односторінковий застосунок)
SQL	Structured Query Language (Структурована мова запитів)
СУБД	Система управління базами даних
TTL	Time To Live (Час життя)
URI	Uniform Resource Identifier (Уніфікований ідентифікатор ресурсу)
URL	Uniform Resource Locator (Уніфікований локатор ресурсів)
WAL	Write-Ahead Logging (Журналювання з випередженням запису)

ВСТУП

Розвиток інформаційних технологій у сфері медицини, біології та фармацевтики зумовив появу нових підходів до автоматизованого аналізу біологічних об'єктів. Одним із перспективних напрямів є застосування методів комп'ютерного зору та машинного навчання для автоматизованого визначення лікарських рослин за фотографіями.

Лікарські рослини протягом багатьох століть залишаються важливою складовою системи охорони здоров’я та використовуються у традиційній і сучасній медицині. За даними Всесвітньої організації охорони здоров’я, значна частина населення світу використовує традиційну медицину та рослинні засоби у межах первинної медичної допомоги [1]. У сучасних дослідженнях зазначається, що понад 80% населення світу використовує рослинні або традиційні засоби лікування, а кількість рослин, які застосовуються у медичних практиках, перевищує 30 тисяч видів [2].

Флора України характеризується значним біорізноманіттям лікарських рослин. За даними «Енциклопедії сучасної України», до лікарських рослин України належать 2219 видів судинних рослин, серед яких близько 1975 видів є дикорослими [3]. Значна частина таких рослин використовується у фітотерапії, фармацевтичній промисловості та народній медицині. Водночас самостійне визначення лікарських рослин залишається складним завданням для нефахівців через морфологічну подібність багатьох видів рослин. Помилки під час ідентифікації можуть призводити до використання непридатних або потенційно небезпечних рослин, що створює ризики для здоров’я людини.

Традиційні способи визначення рослин, зокрема ботанічні визначники та гербарії, потребують спеціалізованих знань і не завжди є зручними під час практичного використання. У зв’язку з цим особливої актуальності набуває створення цифрових довідкових систем, здатних автоматизувати процес

визначення рослин та забезпечити користувача структурованою інформацією через вебінтерфейс.

Стрімкий розвиток технологій штучного інтелекту суттєво змінив підходи до автоматизованої класифікації зображень. У сучасних системах комп'ютерного зору для розпізнавання рослин активно використовуються згортальні нейронні мережі (CNN), які демонструють високі показники точності при аналізі ботанічних зображень [4]. Додаткового розвитку набули технології генеративного штучного інтелекту та великі мовні моделі. Їх використання дозволяє автоматизувати формування структурованої довідкової інформації на основі ботанічних даних, зокрема описів лікувальних властивостей, способів застосування та можливих протипоказань рослин [5].

На сучасному етапі вже існують цифрові сервіси для визначення рослин, серед яких найбільш відомими є PlantSnap, PictureThis та iNaturalist. Однак більшість існуючих рішень орієнтовані переважно на ботанічну класифікацію та не забезпечують комплексного поєднання автоматизованого AI-аналізу, структурованої інформації про лікувальні властивості рослин, інтерактивної карти поширення рослин та україномовного довідкового контенту.

У межах даного дипломного проєкту розроблено вебзастосунок «Інтерактивний довідник лікарських рослин на основі застосування штучного інтелекту». Система забезпечує автоматизоване визначення рослин за фотографією із використанням технологій штучного інтелекту, перегляд каталогу лікарських рослин, отримання структурованої довідкової інформації та роботу з інтерактивною картою знаходжень рослин.

Актуальність теми дипломного проєкту обумовлена активним розвитком технологій штучного інтелекту та комп'ютерного зору, зростанням популярності фітотерапії та рослинних засобів лікування, потребою у доступних цифрових довідкових системах, необхідністю автоматизації процесу визначення рослин, а також відсутністю україномовного спеціалізованого сервісу з повноцінним медичним контентом.

Метою дипломного проєкту є проєктування та реалізація вебзастосунку, що поєднує автоматичну ідентифікацію лікарських рослин за фотографією засобами комп'ютерного зору, генерацію структурованих медичних довідок із застосуванням великих мовних моделей, інтерактивну карту знаходжень рослин та персональну історію пошуків, забезпечуючи доступний та достовірний інструмент для визначення лікарських рослин і отримання медичної інформації про них.

Реалізація поставленої мети передбачає розв'язання таких завдань: проведення аналізу предметної області та порівняльного огляду існуючих рішень; обґрунтування вибору архітектурних підходів і технологічного стеку; проєктування та реалізації повнофункціонального вебзастосунку з інтеграцією зовнішніх AI-сервісів для ідентифікації рослин і генерації медичних довідок; проведення тестування та верифікації коректності роботи системи.

Об’єктом дослідження є процес автоматизованого визначення лікарських рослин та надання довідкової інформації із використанням технологій штучного інтелекту.

Предметом дослідження є методи, алгоритми та програмні засоби реалізації веборієнтованих систем розпізнавання рослин на основі технологій комп’ютерного зору та AI-аналізу.

Практична цінність розробленого застосунку визначається можливістю його безпосереднього використання як публічного україномовного сервісу для ідентифікації лікарських рослин і отримання структурованої медичної інформації. Застосунок може слугувати навчальним інструментом для студентів біологічних, фармацевтичних і медичних спеціальностей під час польових практик, довідковим ресурсом для практиків збору лікарських рослин, а також основою для подальшого розвитку у напрямі інтеграції з державними реєстрами лікарської флори України.

РОЗДІЛ 1. ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ ТА АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ

1.1 Основні поняття та визначення

Для коректного проєктування застосунку необхідно чітко окреслити терміни предметної області. Нижче наведено визначення ключових понять, що охоплюють ботанічну, медичну та технічну складові проєкту.

1.1.1 Лікарська рослина та фітотерапія

Лікарська рослина – рослина, один або декілька органів якої (листя, квіти, плоди, коріння, кора) містять біологічно активні речовини з доведеним або традиційно визнаним терапевтичним ефектом. Відповідно до класифікації Державної фармакопеї України, лікарські рослини поділяються на офіцинальні та народні [6].

Фітотерапія – метод лікування і профілактики захворювань за допомогою лікарських рослин або препаратів на їх основі. За даними Всесвітньої організації охорони здоров’я, близько 80% населення світу тією чи іншою мірою використовує традиційні рослинні засоби у межах первинної медичної допомоги [1]. Важливим поняттям є протипоказання – стани або захворювання, за яких застосування конкретної рослини є небажаним або небезпечним.

1.1.2 Комп'ютерний зір та розпізнавання рослин

Комп'ютерний зір (Computer Vision) – галузь штучного інтелекту, що займається автоматичним аналізом та інтерпретацією цифрових зображень. Розпізнавання рослин за фотографією є задачею класифікації зображень, у якій

за морфологічними ознаками – формою листя, кольором квіток, структурою стебла – визначається таксономічний вид рослини.

Сучасні системи розпізнавання рослин базуються на методах глибокого навчання, зокрема на згортальних нейронних мережах (CNN). На відміну від традиційних підходів, де ознаки зображення визначалися вручну, CNN здатні автоматично виокремлювати ієрархічні візуальні патерни безпосередньо з піксельних даних у процесі навчання [4]. Дослідження LeCun, Bengio та Hinton підтверджують, що глибоке навчання кардинально змінило підходи до задач розпізнавання образів, забезпечуючи якісно вищий рівень точності порівняно з класичними методами машинного навчання [7].

Архітектура CNN складається з трьох функціональних блоків, що наведені на рисунку 1.1. Перший блок – виокремлення ознак (Feature Extraction) – включає чергування згортальних шарів (Convolution) і шарів підвибірки (Pooling). Згортальний шар застосовує набір фільтрів до вхідного зображення, виявляючи локальні патерни: на початкових шарах – краї та текстури, на глибших – складніші структури, характерні для конкретних видів, наприклад форму листкової пластини або розташування жилкування. Шари підвибірки зменшують просторовий розмір карт ознак (Feature Maps), зберігаючи найбільш значущу інформацію та знижуючи обчислювальне навантаження [5]. Другий блок – класифікація (Classification) – перетворює отримані карти ознак у вектор за допомогою шару вирівнювання (Flatten Layer) і передає його через повністю з'єднані шари (Fully Connected Layer). Третій блок – імовірнісний розподіл (Probabilistic Distribution) – застосовує функцію активації SoftMax, яка нормалізує вихідні значення мережі до імовірностей приналежності до кожного класу, де клас з найвищою імовірністю відповідає передбаченому виду рослини.

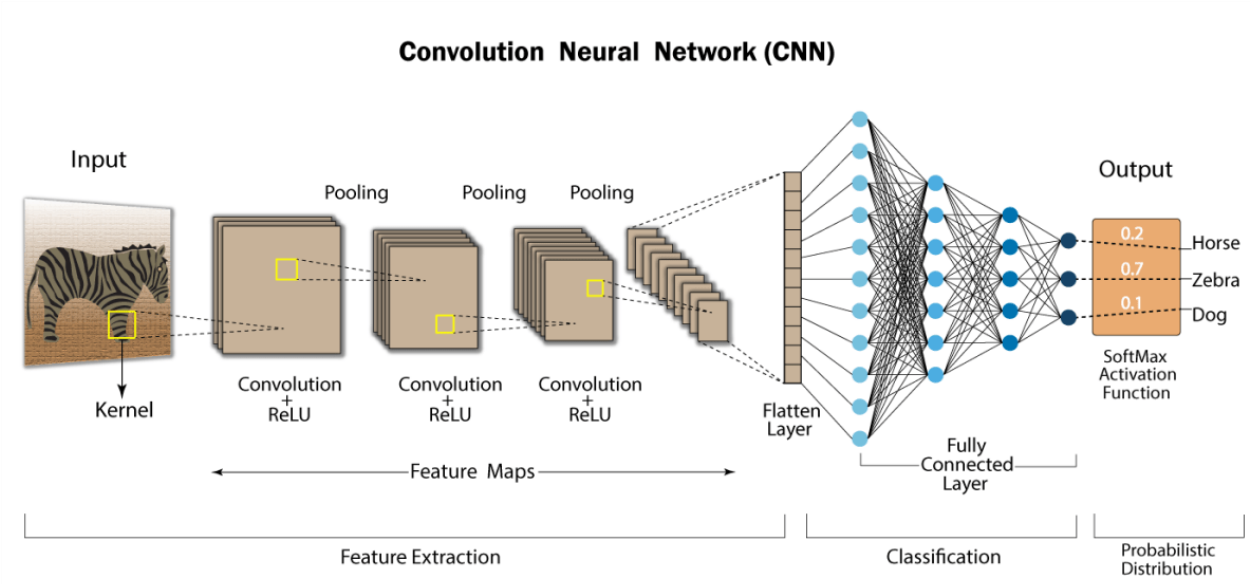


Рисунок 1.1 – Архітектура згортальної нейронної мережі (CNN)

Дослідження у сфері автоматичної ідентифікації рослин засобами глибокого навчання демонструють точність класифікації, порівнянну з результатами кваліфікованих ботаніків, а мультимодальні підходи, що поєднують зображення кількох органів рослини, досягають точності понад 82% на великих таксономічних вибірках [8]. У даному проєкті для розпізнавання рослин використовується Pl@ntNet API (Application Programming Interface) – спеціалізована платформа ботанічної ідентифікації, розроблена консорціумом французьких наукових установ, що повертає результат у вигляді назви виду та показника впевненості класифікатора [9].

1.1.3 Генеративний штучний інтелект та великі мовні моделі

Генеративний штучний інтелект (Generative AI) – клас моделей машинного навчання, здатних генерувати новий контент – текст, зображення, програмний код – на основі патернів, засвоєних під час навчання на великих масивах даних [5]. Великі мовні моделі (Large Language Models, LLM) є підтипом генеративного ШІ, орієнтованим на розуміння та генерацію природної мови. Навчання LLM відбувається на корпусах текстів значного обсягу, що дозволяє

моделям засвоювати семантичні зв'язки між поняттями і генерувати контекстуально релевантні відповіді [10].

Технічною основою сучасних LLM є трансформерна архітектура, запропонована у роботі Vaswani et al. у 2017 році [11]. Ключовим елементом цієї архітектури є механізм уваги (self-attention), який дозволяє моделі враховувати взаємозалежності між усіма елементами вхідної послідовності одночасно, незалежно від відстані між ними у тексті. На відміну від рекурентних мереж, що обробляють текст послідовно, трансформери опрацьовують увесь контекст паралельно, що суттєво підвищує їхню ефективність при роботі з довгими текстами та складними семантичними структурами. Саме ця властивість забезпечує здатність LLM генерувати розгорнуті, структуровані та контекстуально точні відповіді на складні запити.

Практичне застосування LLM у медицині та суміжних галузях активно досліджується науковою спільнотою. Дослідження Thirunavukarasu et al. демонструють, що великі мовні моделі здатні успішно вирішувати задачі медичної діагностики, узагальнення клінічної інформації та генерації структурованих медичних звітів [12]. У фармацевтиці LLM використовуються для систематизації даних про лікарські засоби, аналізу взаємодій між препаратами та автоматизованого наповнення довідкових баз даних. Ці можливості безпосередньо застосовні у сфері фітотерапії: LLM здатні генерувати структуровані медичні довідки про лікарські властивості рослин, протипоказання та способи приготування на основі їхньої наукової назви, що відкриває можливість автоматизованого наповнення ботанічних баз даних достовірною медичною інформацією навіть для рідкісних або регіонально специфічних видів [12].

Важливим архітектурним рішенням при інтеграції LLM у вебзастосунки є організація кешування відповідей. Патерн cache-aside передбачає, що застосунок спочатку перевіряє наявність необхідних даних у власному сховищі, і лише за їх відсутності звертається до зовнішнього API. Отримані дані

зберігаються у сховищі для обслуговування наступних ідентичних запитів [13]. Застосування цього підходу суттєво знижує кількість звернень до зовнішніх сервісів, зменшує час відповіді системи та забезпечує стабільність роботи застосунку в умовах обмежень на кількість API-викликів.

1.1.4 Вебзастосунок та REST API

Вебзастосунок – програмний продукт, що виконується у браузері та взаємодіє з сервером через протокол HTTP/HTTPS. На відміну від настільних застосунків, вебзастосунки не потребують локального встановлення та доступні з будь-якого пристрою з браузером і мережевим підключенням. Архітектура вебзастосунків передбачає розподіл на клієнтську (frontend) та серверну (backend) частини, між якими здійснюється обмін даними через визначений програмний інтерфейс [14].

REST (Representational State Transfer) – архітектурний стиль для побудови розподілених систем, запропонований Роєм Філдінгом у 2000 році [15]. Стиль ґрунтується на низці принципів: відсутність збереженого стану на сервері між запитами (statelessness), уніфікований інтерфейс ресурсів через URI, можливість кешування відповідей та багаторівнева архітектура системи. Взаємодія між клієнтом і сервером здійснюється за допомогою стандартних HTTP-методів – GET, POST, PUT, DELETE – кожен з яких відповідає певній операції над ресурсом. Завдяки відсутності стану кожен запит містить усю необхідну інформацію для його обробки, що суттєво спрощує масштабування серверної частини.

JSON (JavaScript Object Notation) – легковагий текстовий формат обміну даними, що є стандартом де-факто для REST API [16]. Його лаконічний синтаксис і нативна підтримка у JavaScript роблять його оптимальним

форматом для передачі структурованих даних між клієнтом і сервером, зокрема результатів ідентифікації рослин та згенерованих медичних довідок.

JWT – відкритий стандарт RFC (Request for Comments) 7519 для передачі інформації між сторонами у вигляді підписаного JSON-об'єкта [17]. Токен складається з трьох частин: заголовка (header), корисного навантаження (payload) та підпису (signature), що забезпечує цілісність і автентичність даних без необхідності зберігати стан сесії на сервері. Така модель автентифікації є природним доповненням до stateless-архітектури REST.

1.1.5 Геоінформаційні системи та інтерактивні карти

Геоінформаційна система (ГІС) – інтегрована система збирання, зберігання, аналізу та відображення просторово прив'язаних даних. У біологічних науках ГІС широко застосовуються для картографування поширення видів, моніторингу популяцій, аналізу біорізноманіття та прогнозування змін ареалів під впливом кліматичних факторів [18]. Просторові дані у ГІС представлені у вигляді координат географічної системи відліку – широти (latitude) та довготи (longitude), – що дозволяє однозначно локалізувати будь-який об'єкт на поверхні Землі.

Вебкартографія є підгалуззю ГІС, що забезпечує відображення та взаємодію з просторовими даними безпосередньо у браузері без необхідності встановлення спеціалізованого програмного забезпечення. Сучасні вебкартографічні рішення базуються на тайловій моделі: карта поділяється на фрагменти фіксованого розміру (тайли), які завантажуються динамічно залежно від поточного масштабу та положення вікна перегляду, що суттєво знижує обсяг переданих даних [19].

Leaflet.js – відкрита JavaScript-бібліотека для побудови інтерактивних вебкарт, що відзначається легковагістю та широкою екосистемою розширень [20]. На відміну від комерційних альтернатив, Leaflet.js є повністю

безкоштовною і не потребує платіжного облікового запису. Бібліотека підтримує розміщення маркерів за координатами, спливаючі підказки, кластеризацію точок та підключення різних джерел тайлів, зокрема OpenStreetMap. У розроблюваному застосунку Leaflet.js використовується для відображення спільнотної карти знаходжень лікарських рослин, де кожен зареєстрований користувач може додавати власні мітки двома способами: автоматично – через Geolocation API браузера, що визначає поточне місцезнаходження за GPS, або вручну – шляхом вибору точки безпосередньо на карті.

1.2 Огляд предметної області

1.2.1 Актуальність задачі ідентифікації лікарських рослин

Фітотерапія є однією з найдавніших форм медичної практики та залишається широко застосовуваною в сучасному світі. Глобальна стратегія ВООЗ щодо традиційної медицини на 2025-2034 роки закріплює курс на інтеграцію рослинних засобів у національні системи охорони здоров'я та забезпечення загального доступу до перевірених методів фітотерапії [1]. Паралельно світовий ринок лікарських рослин демонструє стійке зростання: за оцінками Grand View Research, його обсяг прогнозується на рівні понад 328 млрд доларів до 2030 року зі щорічним темпом зростання понад 20% [21]. Ці показники відображають реальний суспільний запит на доступні та достовірні інструменти роботи з рослинними ресурсами.

Практична небезпека помилкової ідентифікації є конкретною і задокументованою. Болиголов плямистий (Conium maculatum) зовні схожий на кріп городній (Anethum graveolens), а вороняче око (Paris quadrifolia) – на конвалію (Convallaria majalis): обидві пари видів мають схожу морфологію

листя, проте перші є сильнотоксичними. Саме такі випадки плутанини є причиною значної частини отруєнь рослинами в побутових умовах. Традиційні засоби ідентифікації – польові визначники та гербарії – потребують спеціалізованої підготовки і є фізично недоступними в польових умовах, що робить цифровий інструмент ідентифікації не зручністю, а реальною потребою безпеки.

Окремою проблемою є якість і структурованість медичної інформації, доступної пересічному користувачеві. Навіть правильно ідентифікована рослина не є достатньою умовою безпечного застосування: необхідно мати конкретні відомості про лікарські властивості, протипоказання та допустимі способи приготування. Розрізнені відомості з народних джерел нерідко є неповними або суперечливими, що формує запит на структуровану та верифіковану медичну довідку.

Окремою невирішеною проблемою залишається відсутність централізованого інструменту для фіксації місць знаходження лікарських рослин. Геодані про поширення видів мають подвійну цінність: наукову – для ботанічних досліджень і моніторингу популяцій, та практичну – для людей, що цілеспрямовано шукають певний вид у своєму регіоні. Попри це, спільнотний збір таких даних досі не інтегрований з інструментами ідентифікації та медичним контентом в єдиній системі, доступній широкій аудиторії без фахової підготовки.

1.2.2 Роль штучного інтелекту в обробці ботанічних даних

Розвиток методів глибокого навчання суттєво змінив підходи до автоматичної класифікації біологічних об'єктів. Дослідження Wäldchen та Mäder засвідчують, що сучасні системи на основі CNN здатні ідентифікувати види рослин із точністю, порівнянною з результатами кваліфікованих ботаніків [4]. Мультимодальні підходи, що поєднують зображення кількох органів

рослини – листя, квіток, плодів, кори – демонструють точність понад 82% на великих таксономічних вибірках [8]. Це робить автоматичну ідентифікацію практично застосовною, однак потребує відображення показника впевненості класифікатора для коректного інформування користувача про надійність результату.

Поряд із комп'ютерним зором, великі мовні моделі відкривають принципово нову можливість у роботі з ботанічними даними – автоматичне збагачення баз даних структурованою медичною інформацією. Статичні довідники охоплюють переважно найпоширеніші види і потребують значних ресурсів для ручного наповнення. LLM, навчені на масивах наукової та медичної літератури, здатні генерувати деталізовані довідки про лікарські властивості, протипоказання та способи приготування для будь-якого виду рослини на основі її латинської назви [12]. Це особливо цінно для рідкісних або регіонально специфічних видів, відомості про які у готових довідниках можуть бути відсутні або неповні.

Критично важливим архітектурним аспектом при інтеграції LLM є організація кешування відповідей. Без відповідного механізму один і той самий вид рослини може аналізуватися повторно при кожному зверненні різних користувачів, що призводить до надлишкового використання API-ліміту та збільшення часу відповіді. Патерн cache-aside вирішує цю проблему: після першого аналізу виду результат зберігається у власній базі даних застосунку і повторно використовується для всіх наступних запитів щодо того ж виду [13].

1.3 Цільова аудиторія застосунку

Чітке розуміння цільової аудиторії є вихідною точкою для обґрунтованого формулювання функціональних вимог: саме потреби конкретних груп користувачів визначають пріоритет функцій, структуру каталогу та механізми

взаємодії із застосунком. На основі аналізу предметної області виокремлено три основні групи користувачів.

1.3.1 Аматори фітотерапії та народної медицини

Найбільша за чисельністю група – люди без спеціальної ботанічної освіти, які цікавляться лікарськими рослинами для власного оздоровлення або з пізнавальною метою. Типовий сценарій використання: знайдена в лісі, на городі або на ринку рослина невідомого виду – потреба швидко ідентифікувати та дізнатися, чи можна її використати в лікарських цілях і яким чином. Для цієї групи критично важливі швидкість ідентифікації та зрозумілість медичної інформації: результат має бути представлений у вигляді чітко структурованих полів, а не суцільного тексту. Наявність попередження при низькій впевненості класифікатора та виразне відображення протипоказань є обов'язковими умовами безпечного використання застосунку цією групою.

1.3.2 Студенти та дослідники біологічних і медичних спеціальностей

Студенти фармацевтичних, біологічних і медичних спеціальностей можуть використовувати застосунок як навчальний довідник і інструмент польових спостережень під час практики. Для цієї групи принципово важлива точність латинських назв, що забезпечується через параметр lang=uk Pl@ntNet API з поверненням офіційних ботанічних назв, а не народних варіантів. Можливість фіксувати знаходження рослин на інтерактивній карті під час польових досліджень перетворює застосунок на інструмент накопичення первинних даних про поширення видів. Функція пошуку в каталозі та фільтрація за ознакою лікарськості підтримують систематичне вивчення матеріалу.

1.3.3 Практики збору лікарських рослин

Практики збору лікарських рослин – люди, що цілеспрямовано збирають лікарські рослини в природі та формують групу з найбільш специфічними потребами. Для них ключовою функцією є інтерактивна карта знаходжень: можливість переглядати мітки інших користувачів дозволяє планувати маршрути збору до появи у конкретній місцевості. Додавання власних міток – як автоматично через геолокацію, так і вручну шляхом вибору точки на карті – дає змогу робити внесок у спільнотну базу знаходжень. Функція ідентифікації за фото слугує інструментом верифікації виду безпосередньо в полі перед збором.

1.3.4 Узагальнення потреб цільової аудиторії

Зіставлення потреб трьох груп виявляє не лише спільні пріоритети, а й протиріччя, що потребують врахування при проєктуванні системи.

Спільним для всіх груп є запит на точну ідентифікацію та структуровану медичну інформацію – це незаперечний пріоритет. Однак рівень деталізації, який є прийнятним для студента-фармацевта, може бути надмірним або незрозумілим для аматора фітотерапії. Це протиріччя вирішується через відображення одночасно латинської та загальновживаної назви рослини, а також через структуровані поля довідки, де кожен знаходить потрібний рівень деталізації без необхідності читати весь текст повністю.

Інтерактивна карта знаходжень є ключовою для практиків збору та корисною для студентів під час польових досліджень, тоді як для аматорів вона є другорядною функцією. Це означає, що карта не повинна домінувати в інтерфейсі, але має бути легко доступною як окремий розділ.

Важливим узагальненням є те, що всі три групи об'єднує одна фундаментальна умова – відсутність фахової ботанічної підготовки або її

недостатність у польових умовах. Саме тому система має бути орієнтована на мінімальний поріг входу: завантажив фото – отримав результат з чіткою структурою та попередженням при низькій впевненості класифікатора. Будь-яка додаткова функціональність має залишатися доступною, але не обов'язковою для базового сценарію використання.

1.4 Аналіз існуючих рішень

Для визначення функціональних прогалин на ринку проаналізовано три найбільш поширених застосунки суміжної тематики: iNaturalist, PlantSnap та PictureThis. Критерії аналізу сформовані на основі потреб цільової аудиторії, визначених у підрозділі 1.3.

1.4.1 iNaturalist

iNaturalist – глобальна платформа для спостережень за живою природою, розроблена спільно Каліфорнійською академією наук та Національним географічним товариством і запущена у 2008 році [22]. Платформа поєднує інструменти комп'ютерного зору з механізмом верифікації спільноти експертів: завантажена фотографія організму автоматично аналізується AI-системою, після чого пропонований вид може бути підтверджений або уточнений іншими користувачами. Платформа охоплює не лише рослини, а й тварин, гриби та інші організми, що робить її універсальним інструментом для натуралістів. Станом на 2024 рік платформа налічує понад 200 млн спостережень та 8 млн зареєстрованих користувачів із понад 190 країн світу. Застосунок доступний на платформах iOS та Android, а також через вебінтерфейс. Головна сторінка застосунку iNaturalist наведена на рисунку 1.2.

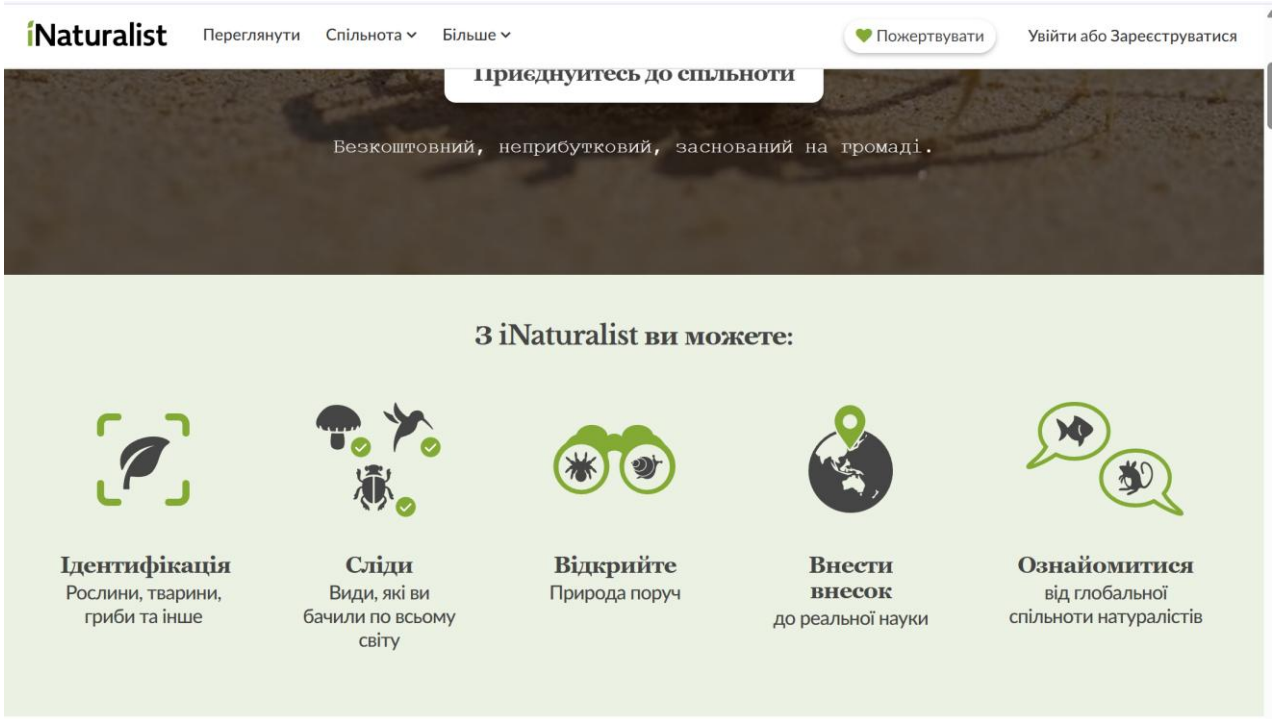


Рисунок 1.2 – Головна сторінка застосунку iNaturalist

Серед переваг iNaturalist – розвинена система інтерактивних карт для відображення географічного поширення видів, відкритий безкоштовний доступ до повного функціоналу, активна міжнародна наукова спільнота та відкритий API, що дозволяє дослідникам використовувати накопичені дані у наукових роботах з біорізноманіття.

Разом з тим платформа орієнтована на загальне спостереження за природою, а не на медичне застосування рослин. Відомості про лікарські властивості, протипоказання та способи приготування у картках видів відсутні. AI-генерація структурованих медичних довідок не реалізована, а інтерфейс орієнтований переважно на досвідчених натуралістів і може бути складним для пересічного користувача, що цікавиться виключно фітотерапевтичним застосуванням рослин.

1.4.2 PlantSnap

PlantSnap – мобільний застосунок для ідентифікації рослин за фотографією, розроблений компанією EasyBit Technologies і вперше представлений у 2016 році [23]. Застосунок використовує власну нейронну мережу для класифікації і позиціонує себе як інструмент для широкої аудиторії завдяки простому інтерфейсу. База даних охоплює понад 1 млн. видів рослин, квітів, грибів та дерев із понад 200 країн. Крім ідентифікації за фото, застосунок пропонує функцію визначення хвороб рослин та базову інформацію про виявлений вид. Доступний на платформах iOS та Android більш ніж 37 мовами інтерфейсу. Зовнішній вигляд головної сторінки застосунку PlantSnap наведено на рисунку 1.3.

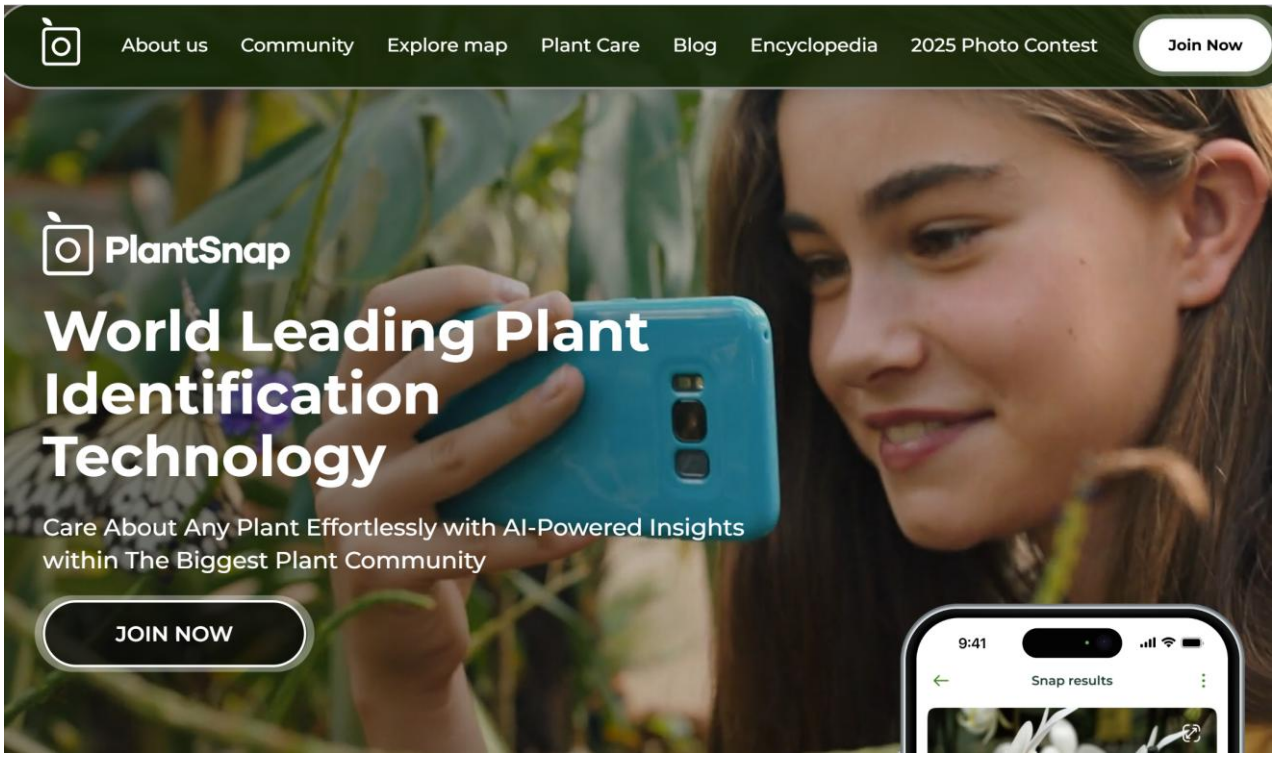


Рисунок 1.3 – Головна сторінка застосунку PlantSnap

До переваг PlantSnap належать широке географічне охоплення бази видів, інтуїтивно зрозумілий процес ідентифікації та підтримка великої кількості мов інтерфейсу, що робить його доступним для міжнародної аудиторії.

Проте базовий функціонал застосунку є платним після короткого безкоштовного пробного періоду. Медична інформація є поверхневою і не структурована у вигляді окремих полів для лікарських властивостей, протипоказань та способів приготування – акцент зроблено на декоративних і садових видах. Інтерактивна карта знаходжень рослин і особиста історія пошуків відсутні. AI-збагачення медичних даних не реалізовано.

1.4.3 PictureThis

PictureThis – один з найбільш завантажуваних мобільних застосунків для ідентифікації рослин, розроблений компанією Glority Global Group Ltd і запущений у 2017 році [24]. Застосунок розпізнає понад 1 млн. видів і орієнтований переважно на аудиторію садівників та любителів декоративних рослин, пропонуючи детальні рекомендації щодо догляду, діагностику захворювань та поради з вирощування. Інтерфейс підтримує понад 30 мов, застосунок доступний на платформах iOS та Android. Щомісячна аудиторія застосунку перевищує 50 млн користувачів у понад 180 країнах, що робить його одним із найпопулярніших рішень у своєму сегменті. Головна сторінка застосунку PictureThis наведена на рисунку 1.4.

Серед переваг – висока точність розпізнавання декоративних і культурних видів, детальні картки рослин з рекомендаціями щодо догляду, функція діагностики хвороб рослин та широка мовна підтримка інтерфейсу.

Водночас застосунок практично не містить спеціалізованої медичної інформації про лікарські властивості та протипоказання, оскільки його предметна область – садівництво, а не фітотерапія. Геолокаційні функції, ведення персональної бази пошуків та розширені відомості про види доступні лише в платній підписці. AI-генерація структурованих медичних довідок не реалізована.

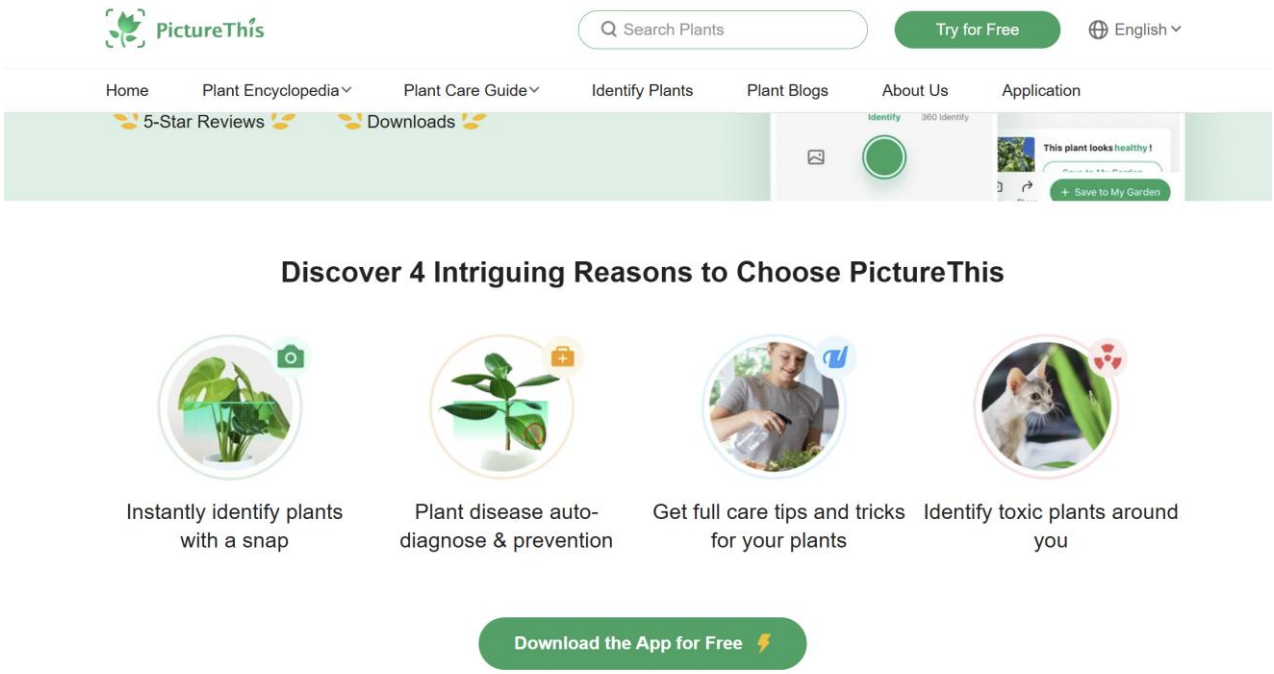


Рисунок 1.4 – Головна сторінка застосунку PictureThis

1.4.4 Порівняльний аналіз рішень

На основі проведеного аналізу складено порівняльну таблицю (Таблиця 1.1), що охоплює ключові функціональні критерії для оцінки розглянутих рішень у порівнянні з розроблюваним застосунком.

Таблиця 1.1 – Порівняльний аналіз існуючих рішень

Критерій	**iNaturalist**	**PlantSnap**	**PictureThis**	**Розроблюваний застосунок**
Розпізнавання рослин за фото	Так	Так	Так	Так (Pl@ntNet API)
AI-аналіз лікарських властивостей	Ні	Частково	Частково	Так (Gemini API)
Інтерактивна карта знаходжень	Так	Ні	Ні	Так (Leaflet.js)

Кінець таблиці 1.1

Критерій	**iNaturalist**	**PlantSnap**	**PictureThis**	**Розроблюваний застосунок**
Структуровані поля: опис / протипоказання / приготування	Ні	Ні	Ні	Так
Спільнотні мітки від користувачів	Так	Ні	Ні	Так
Каталог з пошуком і фільтрацією	Так	Так	Так	Так
Особистий профіль та історія	Так	Так	Платно	Безкоштовно
Кешування AI-відповідей (cache-aside)	Ні	Ні	Ні	Так
Українські ботанічні назви (lang=uk)	Ні	Ні	Ні	Так (PlantNet lang=uk)
Безкоштовний повний доступ	Так	Частково	Ні	Так

Дані таблиці підтверджують, що жодне з розглянутих рішень не забезпечує повного комплексу необхідних функцій: структурованої медичної інформації з окремими полями для протипоказань та способів приготування, AI-генерації довідок із кешуванням результатів, спільнотної інтерактивної карти знаходжень з можливістю додавання міток як автоматично через геолокацію, так і вручну, а також безкоштовного повного доступу. Це обґрунтовує доцільність розробки спеціалізованого застосунку, орієнтованого на потреби користувачів у сфері лікарських рослин.

1.4.5 Недоліки та обмеження наявних рішень

Узагальнення результатів порівняльного аналізу дозволяє систематизувати ключові недоліки існуючих рішень, що визначають вектор розробки застосунку:

- відсутність структурованої медичної інформації – жоден із розглянутих застосунків не надає окремих полів для лікарських властивостей, протипоказань та способів приготування лікарських форм; інформація або подається у вигляді загального опису, або відсутня взагалі, що є недостатнім для безпечного фітотерапевтичного застосування;
- відсутність AI-збагачення ботанічних даних – розглянуті рішення використовують статичні бази даних, що охоплюють переважно найпоширеніші види; автоматична генерація медичних довідок засобами великих мовних моделей у жодному з аналізованих застосунків не реалізована;
- відсутність або обмеженість геолокаційного функціоналу – лише iNaturalist надає повноцінну карту знаходжень, однак без прив'язки до медичного контексту і без можливості додавати мітки вручну; PlantSnap та PictureThis геолокаційний функціонал у безкоштовній версії не надають;
- платність ключового функціоналу – PlantSnap та PictureThis обмежують доступ до розширених можливостей платною підпискою, що знижує доступність інструменту для широкої аудиторії;
- відсутність кешування результатів AI-аналізу – архітектура існуючих рішень не передбачає збереження результатів аналізу для повторного використання, що при масштабуванні призводить до надлишкових витрат на API-виклики та збільшення часу відповіді.

Виявлені недоліки є безпосереднім обґрунтуванням функціональних вимог до розроблюваного застосунку, сформульованих у підрозділі 1.5.

1.5 Формування вимог до застосунку

1.5.1 Функціональні вимоги

Функціональні вимоги сформульовано на основі виявлених недоліків існуючих рішень (підрозділ 1.4.5) та потреб цільової аудиторії (підрозділ 1.3). Кожній вимозі присвоєно унікальний ідентифікатор у форматі «ФВ-ХХ», де «ФВ» означає «функціональна вимога», а «ХХ» – порядковий номер, що використовується для посилань у подальших розділах роботи. Вимоги систематизовано у таблиці 1.2.

Таблиця 1.2 – Функціональні вимоги до застосунку

№	Вимога	Пріоритет
ФВ-01	Визначення виду лікарської рослини за завантаженою фотографією з відображенням латинської назви, українського найменування та показника впевненості класифікатора	Високий
ФВ-02	Автоматична генерація структурованої медичної довідки із лікарськими властивостями, протипоказаннями та способами приготування для ідентифікованого виду	Високий
ФВ-03	Кешування результатів AI-аналізу у базі даних за латинською назвою виду для унеможливлення повторних звернень до зовнішніх сервісів	Високий
ФВ-04	Каталог лікарських рослин із можливістю текстового пошуку та фільтрацією за ознакою лікарськості	Високий

Кінець таблиці 1.2

№	Вимога	Пріоритет
ФВ-05	Інтерактивна карта зі спільнотними мітками знаходжень рослин та можливістю додавання власних міток – автоматично через геолокацію або вручну шляхом вибору точки на карті	Високий
ФВ-06	Реєстрація та автентифікація користувачів із захистом персональних даних на основі JWT	Високий
ФВ-07	Збереження та перегляд особистої історії пошукових запитів у профілі користувача	Середній

1.5.2 Нефункціональні вимоги

Нефункціональні вимоги визначають якісні характеристики системи, що безпосередньо впливають на надійність, безпеку та користувацький досвід. На відміну від функціональних вимог, що описують що система має робити, нефункціональні визначають як саме вона має це робити. Вимоги сформульовано з урахуванням специфіки застосунку, що взаємодіє із зовнішніми API та зберігає персональні дані користувачів, і систематизовано у таблиці 1.3.

Таблиця 1.3 – Нефункціональні вимоги до застосунку

Категорія	Вимога	Метрика
Продуктивність	Відповідь на запити, що обслуговуються з кешу бази даних, не повинна перевищувати 3 секунди	Час відповіді API < 3 с

Кінець таблиці 1.3

Категорія	Вимога	Метрика
Безпека	Паролі користувачів зберігаються виключно у вигляді bcrypt-хешу; API-ключі зовнішніх сервісів передаються лише на серверній стороні; захищені маршрути перевіряють JWT-токен	Відсутність відкритих секретів у клієнтському коді
Надійність	Система коректно обробляє низький показник впевненості класифікатора та тимчасову недоступність зовнішніх API без аварійного завершення роботи	Відсутність необроблених виключень у логах
Масштабованість	Архітектура застосунку допускає розширення переліку властивостей рослин без зміни схеми основних таблиць бази даних	Можливість додавання нових типів медичних властивостей без внесення структурних змін до бази даних
Зручність	Інтерфейс є адаптивним і коректно відображається на мобільних та десктоп-пристроях; при низькому confidence score відображається відповідне попередження	Коректне відображення на екранах від 375px

ВИСНОВКИ ДО РОЗДІЛУ

У першому розділі проведено комплексний аналіз предметної області, що охоплює ботанічну, медичну та технічну складові розроблюваного застосунку. Визначено та систематизовано ключові поняття: лікарська рослина, фітотерапія, протипоказання, комп'ютерний зір, згортальна нейронна мережа, генеративний штучний інтелект, велика мовна модель, трансформерна архітектура, патерн cache-aside, REST API, JWT, геоінформаційна система та інтерактивна карта – кожне з яких безпосередньо пов'язане з конкретним архітектурним або технічним рішенням застосунку.

Обґрунтовано актуальність обраної теми. Світовий ринок лікарських рослин демонструє щорічне зростання понад 20%, глобальна стратегія ВООЗ на 2025-2034 роки закріплює курс на інтеграцію фітотерапії у національні системи охорони здоров'я, а флора України налічує понад 2 000 лікарських видів судинних рослин, значна частина яких має морфологічно схожих токсичних двійників. Встановлено, що проблема доступності достовірної медичної інформації про лікарські рослини набуває особливої гостроти в умовах воєнного стану, коли потреба у цифрових інструментах первинної медичної допомоги суттєво зростає.

Виокремлено три групи цільової аудиторії – аматори фітотерапії, студенти біологічних і медичних спеціальностей та практики збору лікарських рослин. Узагальнено їхні спільні й специфічні потреби, що дозволило сформулювати вимоги до функціоналу застосунку, які відображають реальні сценарії використання.

Проведено порівняльний аналіз існуючих рішень – iNaturalist, PlantSnap та PictureThis. Встановлено відсутність на ринку інструменту, який би поєднував структуровану медичну інформацію з окремими полями для протипоказань та способів приготування, AI-генерацію довідок із кешуванням результатів,

спільнотну карту знаходжень з можливістю додавання міток як автоматично, так і вручну, та безкоштовний повний доступ. Виявлені прогалини безпосередньо визначили перелік вимог до розроблюваного застосунку.

Сформульовано 7 функціональних та 5 нефункціональних вимог, що слугують основою для проєктування архітектури системи та обґрунтування вибору технологічного стеку у наступних розділах.

РОЗДІЛ 2. АНАЛІЗ МЕТОДІВ ТА ТЕХНОЛОГІЙ ДЛЯ РОЗРОБКИ ПЛАТФОРМИ

2.1 Аналіз типів платформ

Перед початком проєктування системи необхідно визначити форм-фактор розроблюваного продукту, оскільки цей вибір безпосередньо обумовлює архітектурні рішення, технологічний стек та сценарії взаємодії користувача з системою. Розглянуто три основних варіанти: вебплатформу, мобільну платформу та десктопну платформу.

2.1.1 Вебплатформа

Вебплатформа функціонує у браузері та не потребує встановлення програмного забезпечення на пристрій користувача. Доступ здійснюється через URL-адресу з будь-якого пристрою, що має сучасний браузер і підключення до мережі Інтернет [14]. Завдяки прогресивним вебтехнологіям (Progressive Web App, PWA) сучасні вебзастосунки здатні частково реалізовувати функціональність мобільних додатків: геолокацію через Geolocation API, завантаження фотографій через елемент input з атрибутом capture, адаптивний інтерфейс через CSS media queries та кешування ресурсів через Service Worker.

Для розроблюваного застосунку вебплатформа є природним вибором: функція ідентифікації рослин за фотографією реалізується через завантаження файлу, геолокація для карти – через Geolocation API браузера, а інтерактивна карта – через бібліотеку Leaflet.js. Усі ці можливості доступні у браузері без встановлення додаткового програмного забезпечення. Крім того, взаємодія з зовнішніми API (Pl@ntNet, Gemini) виконується на серверній стороні, що не залежить від типу клієнтської платформи.

2.1.2 Мобільна платформа

Мобільний застосунок розробляється для iOS та Android і розповсюджується через App Store або Google Play. Нативні мобільні застосунки мають прямий доступ до апаратних можливостей пристрою – камери, GPS, push-сповіщень – без проміжного шару браузера. Проте розробка потребує підтримки двох окремих кодових баз або використання кросплатформних фреймворків (React Native, Flutter), що суттєво збільшує обсяг роботи, час розробки та вартість підтримки [25].

Додатковим обмеженням є процедура публікації у магазинах застосунків: проходження модерації App Store та Google Play займає від кількох днів до тижнів і може бути відхилено. Для академічного проєкту з обмеженим часом розробки ці витрати є непропорційними. Геолокація та завантаження фото, що є ключовими перевагами мобільної платформи, доступні через відповідні браузерні API, що нівелює цю перевагу для цілей проєкту.

2.1.3 Десктопна платформа

Десктопний застосунок встановлюється локально і виконується нативно на операційній системі користувача. Він надає прямий доступ до апаратних ресурсів та може функціонувати без постійного підключення до мережі Інтернет. Проте для застосунку, що принципово залежить від зовнішніх API (Pl@ntNet, Gemini), офлайн-режим позбавлений практичного сенсу. Десктопна платформа потребує окремих дистрибутивів для Windows, macOS та Linux, що ускладнює підтримку та поширення [26]. Крім того, завантаження та встановлення застосунку є додатковим бар'єром для кінцевого користувача.

2.1.4 Вибір типу платформи

На основі проведеного аналізу складено порівняльну таблицю (Таблиця 2.1).

Таблиця 2.1 – Порівняльний аналіз типів платформ

Критерій	**Вебплатформа**	**Мобільна платформа**	**Десктопна платформа**
Доступність без встановлення	Так	Ні	Ні
Кросплатформеність	Так	Часткова (iOS/Android)	Ні
Взаємодія з зовнішніми API	Так	Так	Так
Геолокація через браузер	Так (Geolocation API)	Так (нативно)	Обмежено
Інтерактивна карта	Так (Leaflet.js)	Так	Обмежено
Завантаження фото	Так (input file)	Так (камера)	Так
Адаптивний інтерфейс	Так	Так	Ні
Вартість підтримки	Низька	Висока	Середня
Офлайн-режим	Обмежено (PWA)	Так	Так

Вебплатформа є оптимальним вибором для розроблюваного застосунку. Вона забезпечує кросплатформеність без додаткових витрат на розробку окремих збірок, просту інтеграцію з зовнішніми API через серверну частину, підтримку інтерактивних карт засобами Leaflet.js та адаптивного інтерфейсу через Tailwind CSS. Геолокація та завантаження фотографій доступні через

стандартні браузерні API без необхідності нативного доступу до апаратури пристрою.

2.2 Аналіз архітектурних підходів

Архітектура програмного забезпечення визначає спосіб організації компонентів системи, їхню взаємодію та розподіл відповідальності. Вибір архітектурного підходу впливає на масштабованість, підтримуваність та складність розробки. Розглянуто два основних підходи: монолітну та мікросервісну архітектуру.

2.2.1 Монолітна архітектура

За монолітного підходу всі компоненти застосунку – клієнтська частина, серверна логіка та рівень доступу до даних – розгортаються як єдиний артефакт. Внутрішньо моноліт може бути структурований у вигляді шарів: презентаційного (обробка HTTP-запитів та формування відповідей), бізнес-логіки (обробка запитів до зовнішніх API, кешування, валідація) та рівня даних (взаємодія з PostgreSQL через Prisma ORM(Object-Relational Mapping)). Така внутрішня структуризація забезпечує розподіл відповідальності без операційної складності мікросервісів [27].

Монолітна архітектура є природною відправною точкою для більшості проєктів: вона проста у розробці, тестуванні та розгортанні, не потребує складної інфраструктури, дозволяє легко рефакторити код і виключає потребу у міжсервісній комунікації. Для застосунку з одним розробником і чітко визначеним функціональним периметром ця архітектура є оптимальною.

2.2.2 Мікросервісна архітектура

Мікросервісний підхід передбачає декомпозицію застосунку на набір незалежних сервісів, кожен з яких реалізує окрему бізнес-функцію і може розгортатися та масштабуватися незалежно. Сервіси взаємодіють через мережеві протоколи (REST, gRPC (Google Remote Procedure Call), черги повідомлень) [28]. Ця архітектура добре підходить для великих систем з високим навантаженням і командами з десятків розробників.

Проте мікросервісна архітектура вносить значну операційну складність: потребує налаштування оркестрації контейнерів (Kubernetes), міжсервісної комунікації, розподіленого логування та трасування запитів, окремих конвеєрів CI/CD для кожного сервісу. Для проєкту одного розробника ці витрати є непропорційними до отриманих переваг, а потреба у горизонтальному масштабуванні окремих компонентів відсутня на поточному етапі.

2.2.3 Вибір архітектурного підходу

На основі проведеного аналізу складено порівняльну таблицю (Таблиця 2.2).

Таблиця 2.2 – Порівняльний аналіз архітектурних підходів

Критерій	**Монолітна архітектура**	**Мікросервісна архітектура**
Простота початкової розробки	Висока	Низька
Складність розгортання	Низька	Висока
Масштабованість	Обмежена (вертикальна)	Висока (горизонтальна)
Налаштування інфраструктури	Мінімальне	Складне (оркестрація)

Кінець таблиці 2.2

Критерій	**Монолітна архітектура**	**Мікросервісна архітектура**
Спільна кодова база	Так	Ні
Час запуску MVP (Minimum Viable Product)	Короткий	Тривалий
Відповідність масштабу проєкту	Висока	Надлишкова
Операційні витрати	Низькі	Високі

Для розроблюваного застосунку обрано монолітну клієнт-серверну архітектуру з чітким внутрішнім шаруванням. Система складається з двох розгорнутих артефактів: клієнтської частини (React Single Page Application, SPA) та серверної частини (Node.js + Express REST API). Сервер внутрішньо організовано у три шари: маршрутизатори (routing) – контролери (controllers) – сервіси (services). Такий підхід забезпечує чіткий розподіл відповідальності при збереженні простоти розгортання та підтримки.

2.3 Аналіз технологій реалізації

2.3.1 Мова програмування

Вибір мови програмування є фундаментальним рішенням, що визначає технологічний стек, доступну екосистему бібліотек та можливість повторного використання коду між клієнтською і серверною частинами. Розглянуто три кандидати: JavaScript, Python та Java.

Python – мова з лаконічним синтаксисом і розвиненою екосистемою для задач машинного навчання (TensorFlow, PyTorch) та обробки даних. У контексті

веброзробки Python застосовується на серверній стороні через фреймворки Django та Flask. Проте Python не виконується у браузері, що унеможливлює єдиний мовний стек для клієнтської та серверної частин і потребує перемикання між двома мовами [29].

Java – строго типізована компільована мова для JVM, широко застосовувана у корпоративному секторі через фреймворк Spring Boot. Java пропонує потужну систему типів та зрілу екосистему, однак потребує більшого обсягу шаблонного коду, має тривалий час запуску JVM та не є природним вибором для легковагих вебзастосунків з інтеграцією зовнішніх API [30].

JavaScript – базова мова вебплатформи з подієво-орієнтованою асинхронною моделлю виконання. Завдяки середовищу Node.js JavaScript поширився на серверну розробку, забезпечуючи можливість використання єдиної мови для всього стеку. Це дозволяє повторно використовувати логіку валідації даних, спільні типи та утиліти між клієнтом і сервером, знижує когнітивне навантаження на розробника та спрощує підтримку кодової бази [31].

Узагальнення результатів порівняльного аналізу наведено у таблиці 2.3.

Таблиця 2.3 – Порівняльний аналіз мов програмування

Критерій	**JavaScript**	**Python**	**Java**
Використання на фронтенді	Так (нативно)	Ні	Ні
Використання на бекенді	Так (Node.js)	Так (Django/Flask)	Так (Spring)
Єдина мова для всього стеку	Так	Ні	Ні
Екосистема пакетів (npm)	Найбільша	Велика	Велика
Асинхронна модель	Подієво-орієнтована	Async/await	Threads
Поріг входу	Низький	Низький	Середній

Для розроблюваного застосунку обрано JavaScript як єдину мову програмування для клієнтської та серверної частин. Ключовою перевагою є уніфікований стек: спільна мова дозволяє повторно використовувати логіку валідації та утиліти між фронтендом і бекендом, знижує когнітивне навантаження на розробника та спрощує підтримку кодової бази. Додатковим чинником є найширша серед розглянутих варіантів екосистема npm-пакетів, що забезпечує готові рішення для кожної задачі проєкту – від HTTP-клієнта до інтеграції з картами.

2.3.2 Frontend

Для побудови клієнтської частини розглянуто три провідних рішення: React, Vue.js та Angular.

Angular – повноцінний фреймворк від Google із вбудованою системою впровадження залежностей (Dependency Injection), двостороннім зв'язуванням даних та власним механізмом виявлення змін (Change Detection). Angular орієнтований на великі корпоративні застосунки зі строгою TypeScript-типізацією та жорсткою структурою проєкту. Це забезпечує прогнозованість, проте значно підвищує поріг входу та вносить надлишкову складність для проєкту одного розробника [32].

Vue.js – прогресивний фреймворк із реактивною системою даних та інтуїтивним синтаксисом одно-файлових компонентів (Single File Components). Vue пропонує плавну криву навчання та добре підходить для малих і середніх проєктів. Проте за обсягом екосистеми та кількістю готових інтеграцій Vue поступається React, що може обмежити вибір бібліотек для специфічних задач [33].

React – відкрита JavaScript-бібліотека від Meta для побудови компонентних інтерфейсів. Ключовою концепцією React є Virtual DOM –

легковагове представлення реального DOM у пам'яті, яке дозволяє мінімізувати кількість реальних DOM-операцій шляхом порівняння (diffing) попереднього та нового стану дерева компонентів. Хуки (useState, useEffect, useCallback) забезпечують управління станом і побічними ефектами у функціональних компонентах без використання класів [34].

Для стилізації обрано Tailwind CSS – утилітарну CSS-бібліотеку, що реалізує підхід utility-first: замість написання власних CSS-класів розробник комбінує готові утилітарні класи безпосередньо у JSX. Це забезпечує швидке прототипування, усуває проблему конфліктів CSS-специфічності та гарантує мінімальний розмір фінального бандлу завдяки PurgeCSS [35]. Для інтерактивної карти використовується Leaflet.js [20].

Порівняльні характеристики розглянутих рішень зведено у таблиці 2.4.

Таблиця 2.4 – Порівняльний аналіз frontend-рішень

Критерій	**React**	**Vue.js**	**Angular**
Тип рішення	Бібліотека UI	Прогресивний фреймворк	Повний фреймворк
Поріг входу	Середній	Низький	Високий
Virtual DOM	Так	Так	Так (Change Detection)
Екосистема npm	Найбільша	Велика	Велика
Гнучкість архітектури	Висока	Висока	Обмежена
Підтримка TypeScript	Так	Так	Так (за замовчуванням)
Спільнота та документація	Найбільша	Велика	Велика

Для розроблюваного застосунку обрано React у поєднанні з Tailwind CSS та Leaflet.js. React забезпечує компонентну архітектуру та ефективне оновлення

інтерфейсу, Tailwind CSS – швидку адаптивну стилізацію, Leaflet.js – повнофункціональну безкоштовну інтерактивну карту.

Додатковим чинником вибору React є його нативна інтеграція з бібліотеками, що критично важливі для даного проєкту. Бібліотека react-leaflet надає React-компоненти-обгортки над Leaflet.js, що дозволяє декларативно керувати станом карти, маркерами та шарами через стандартні React-хуки, не порушуючи компонентної моделі застосунку [34]. Бібліотека react-router-dom забезпечує клієнтську маршрутизацію між сторінками без перезавантаження браузера, реалізуючи патерн Single Page Application (SPA). Хук useEffect у поєднанні з бібліотекою Axios (HTTP-клієнт) забезпечує декларативне управління асинхронними запитами до REST API сервера з автоматичним очищенням підписок.

Екосистема React (npm) є найбільшою серед розглянутих рішень: для React існують готові, активно підтримувані пакети для кожної задачі проєкту – маршрутизація (react-router-dom), іконки (lucide-react), HTTP-запити (axios), карти (react-leaflet). Vue.js, попри зрілу екосистему, має менший вибір готових компонентів для нішевих задач, зокрема для інтеграції з Leaflet.js [33].

2.3.3 Backend

Серверна частина застосунку реалізується на платформі Node.js. Node.js побудований на подієво-орієнтованій неблокуючій моделі вводу-виводу (Event Loop + Input/Output): замість виділення окремого потоку на кожне з'єднання (як у традиційних серверах), Node.js обробляє запити асинхронно в одному потоці. Це робить платформу ефективною для Input/Output-інтенсивних задач, зокрема проксування запитів до зовнішніх API та роботи з базою даних [31]. Розглянуто три серверних рішення.

Fastify – сучасний Node.js-фреймворк, оптимізований для максимальної продуктивності. Бенчмарки показують, що Fastify обробляє до 76 000 запитів

на секунду порівняно з ~15 000 у Express.js [36]. Fastify використовує JSON Schema для автоматичної валідації запитів і серіалізації відповідей, що прискорює обробку. Проте екосистема плагінів Fastify є значно меншою за Express: зокрема, підтримка Multer (завантаження файлів) та деяких JWT-бібліотек реалізована через сторонні адаптери, а не нативно. Для застосунку, де завантаження файлів (фото рослин) є ключовою функцією, зрілість підтримки Multer є критичним фактором [36].

Django (Python) – повноцінний вебфреймворк із вбудованою ORM, адміністративною панеллю та системою аутентифікації. Попри зрілість та багатий функціонал, Django вимагає зміни мови програмування відносно фронтенду, що порушує уніфікованість стеку та потребує подвійного знання мов [29].

Spring Boot (Java) – корпоративний фреймворк із потужною системою впровадження залежностей та широкою екосистемою. Він добре підходить для великих систем з мікросервісною архітектурою, проте має значний час запуску, велику кількість шаблонного коду та не відповідає вимозі єдиного JavaScript-стеку [30].

Express.js – мінімалістичний та найбільш поширений Node.js-фреймворк, що забезпечує маршрутизацію HTTP-запитів та middleware-ланцюжки обробки. Middleware – це функції, що виконуються послідовно у ланцюжку обробки запиту і дозволяють реалізувати автентифікацію, логування, обробку помилок та CORS (Cross-Origin Resource Sharing) без зайвої зв'язності. Express легко інтегрується з Prisma ORM та має найбільшу кількість навчальних матеріалів і прикладів у відкритому доступі [37].

На основі проведеного аналізу складено порівняльну таблицю (Таблиця 2.5).

Таблиця 2.5 – Порівняльний аналіз серверних рішень

Критерій	**Node.js + Express.js**	**Fastify**	**Django (Python)**	**Spring Boot (Java)**
Мова реалізації	JavaScript	JavaScript	Python	Java
Єдина мова з фронтендом	Так	Так	Ні	Ні
Модель обробки запитів	Подієво-орієнтована	Подієво-орієнтована	Синхронна/Async	Thread-based
Гнучкість конфігурації	Висока	Висока	Середня	Обмежена
Екосистема пакетів	Найбільша	Середня	Велика	Велика

Для розроблюваного застосунку обрано Node.js у поєднанні з Express.js. Незважаючи на нижчу теоретичну продуктивність порівняно з Fastify, вибір Express обґрунтований трьома факторами. По-перше, Express має найбільшу екосистему middleware і є де-факто стандартом для Node.js REST API, що забезпечує широку документованість і передбачуваність поведінки при інтеграції зі сторонніми бібліотеками. По-друге, інтеграція з Prisma ORM, Multer для завантаження файлів та JWT реалізована через перевірені, активно підтримувані пакети з мільйонами завантажень на тиждень. По-третє, для застосунку з помірним навантаженням різниця у продуктивності між Express та Fastify не є практично значущою – вузьким місцем системи є мережеві запити до зовнішніх API, а не обробка HTTP на рівні фреймворку.

2.3.4 API-протокол взаємодії клієнта з сервером

Для обміну даними між клієнтською та серверною частинами розглянуто два підходи: REST та GraphQL.

GraphQL – мова запитів для API, розроблена Facebook у 2012 році і відкрита у 2015. GraphQL дозволяє клієнту точно визначати структуру необхідних даних у запиті, що усуває проблеми надмірної (over-fetching) та недостатньої (under-fetching) вибірки. GraphQL ефективний для складних систем з великою кількістю взаємопов'язаних сутностей та різноманітними клієнтами, однак вносить додаткову складність: потребує визначення схеми, резолверів та спеціалізованих інструментів кешування [38].

REST (Representational State Transfer) – архітектурний стиль на основі стандартних HTTP-методів (GET, POST, PUT, DELETE) та ресурсних URI. REST є індустріальним стандартом для веб-API з чіткою семантикою операцій, простим HTTP-кешуванням та мінімальними вимогами до клієнта [15]. Для застосунку з чітко визначеним набором ресурсів (Plant, MapMarker, SearchHistory, User) REST забезпечує достатній рівень гнучкості при значно нижчій складності реалізації та підтримки. Для розроблюваного застосунку обрано REST API.

2.4 Аналіз систем управління базами даних та ORM

2.4.1 Аналіз СУБД

Вибір системи управління базами даних визначається структурою даних застосунку, вимогами до цілісності та специфікою типів даних, що зберігаються. Розглянуто три варіанти: PostgreSQL, MySQL та MongoDB.

MongoDB – документоорієнтована NoSQL система управління базами даних (СУБД), що зберігає дані у форматі BSON (бінарний JSON). Вона добре підходить для неструктурованих або схемно нестабільних даних та горизонтального масштабування. Проте підтримка складних реляційних зв'язків є обмеженою: операції JOIN виконуються через aggregation pipeline, що

є менш ефективним і інтуїтивним порівняно з SQL. Повна підтримка ACID-транзакцій (Atomicity, Consistency, Isolation, Durability) з'явилась лише у версії 4.0 і залишається обмеженою для багатодокументних операцій [39].

MySQL – реляційна СУБД з тривалою історією та широким застосуванням у LAMP-стеку. MySQL є надійним і добре документованим рішенням, однак поступається PostgreSQL у кількох ключових аспектах: нативний тип JSONB (бінарний JSON з GIN-індексами (Generalized Inverted Index)) у MySQL відсутній, підтримка деяких стандартів SQL є неповною, а GIN реалізовано лише через рушій InnoDB [40].

PostgreSQL – об'єктно-реляційна СУБД з відкритим кодом, що розвивається з 1996 року і вирізняється розширеними можливостями порівняно з класичними реляційними системами. PostgreSQL реалізує MVCC (Multi-Version Concurrency Control) на рівні ядра СУБД, що дозволяє читачам не блокувати записувачів і забезпечує високу паралельність без деградації продуктивності [39]. WAL-логування (Write-Ahead Logging) гарантує відновлення після збоїв без втрати даних.

Критично важливою для даного проєкту є підтримка нативного типу JSONB. На відміну від текстового JSON, JSONB зберігає документ у бінарному декомпонованому форматі та підтримує GIN-індекси для ефективного пошуку всередині JSON-документів. Це безпосередньо використовується для зберігання структурованих AI-відповідей Gemini API у полі medicinalInfo таблиці Plant: вся медична довідка (лікарські властивості, протипоказання, способи приготування) зберігається як єдиний JSONB-об'єкт без потреби у нормалізації у додаткові таблиці.

Таблиця 2.6 – Порівняльний аналіз СУБД

Критерій	**PostgreSQL**	**MySQL**	**MongoDB**
Тип	Об'єктно-реляційна	Реляційна	Документоорієнтована

Кінець таблиці 2.6

Критерій	PostgreSQL	MySQL	MongoDB
ACID-транзакції	Повна підтримка	Повна підтримка	Часткова
Нативний тип JSONB	Так	Ні	Так (BSON)
GIN-індекси для JSON	Так	Ні	Ні
Складні JOIN-запити	Висока підтримка	Висока підтримка	Обмежена
MVCC (без блокувань)	Так	Частково (InnoDB)	Так
Відкритий код	Так	Так (GPL)	Частково

Для розроблюваного застосунку обрано PostgreSQL. Вирішальними факторами є: повна ACID-транзакційність для гарантії цілісності даних при одночасному збереженні результатів ідентифікації та міток карти, нативний тип JSONB з GIN-індексами для ефективного зберігання AI-відповідей та зріла реалізація MVCC для підтримки паралельних запитів без блокувань.

2.4.2 Аналіз ORM-інструментів

Для взаємодії з PostgreSQL з JavaScript-середовища розглянуто два ORM-інструменти: Prisma та Sequelize.

Sequelize – зрілий ORM для Node.js з підтримкою кількох СУБД (PostgreSQL, MySQL, SQLite, MSSQL). Він використовує підхід на основі JavaScript-класів для визначення моделей та підтримує асоціації, хуки та міграції. Проте типізація в Sequelize є менш строгою: типи моделей необхідно визначати вручну, що збільшує ризик розходження між схемою БД та типами у коді [41].

Prisma – сучасний ORM нового покоління з декларативною схемою, що описує структуру бази даних у єдиному файлі schema.prisma. На основі цієї схеми Prisma автоматично генерує типізований клієнт, що забезпечує повну відповідність між схемою БД та типами у TypeScript/JavaScript без ручного визначення. Prisma Studio – вбудований графічний інтерфейс для перегляду та редагування даних у базі – суттєво прискорює процес налагодження на етапі розробки. Підтримка автоматичної генерації та застосування міграцій через команду prisma migrate dev забезпечує контрольований розвиток схеми [42].

Результати аналізу систематизовано у таблиці 2.7.

Таблиця 2.7 – Порівняльний аналіз ORM-інструментів

Критерій	**Prisma**	**Sequelize**
Декларативна схема	Так (schema.prisma)	Ні (JS-класи)
Автогенерація міграцій	Так	Так
Типізація клієнта	Автоматична (TypeScript)	Ручна
Graphical UI (Studio)	Так (Prisma Studio)	Ні
Підтримка PostgreSQL	Повна	Повна
Читабельність схеми	Висока	Середня

Для розроблюваного застосунку обрано Prisma ORM. Декларативна схема, автогенерація типізованого клієнта та Prisma Studio визначили цей вибір як оптимальний для проєкту з PostgreSQL.

2.5 Аналіз механізмів автентифікації

Автентифікація є критичною підсистемою застосунку, оскільки персональна історія пошуків та мітки карти прив'язані до конкретного облікового запису користувача. Розглянуто два підходи: сесійну автентифікацію та автентифікацію на основі JWT.

2.5.1 Сесійна автентифікація

Традиційна сесійна автентифікація передбачає зберігання стану сесії на сервері. Після успішного входу сервер створює запис сесії у базі даних або Redis-сховищі і повертає клієнту HTTP-cookie з унікальним ідентифікатором сесії (session ID). При кожному наступному запиті сервер зчитує цей ідентифікатор, звертається до сховища сесій та перевіряє актуальність і права доступу. Такий підхід є stateful: сервер зберігає стан між запитами. При горизонтальному масштабуванні всі сервери мають розділяти спільне сховище сесій, що потребує додаткової інфраструктури [43].

2.5.2 Автентифікація на основі JWT

JWT (JSON Web Token) – відкритий стандарт RFC 7519 для передачі верифікованої інформації між сторонами у вигляді підписаного JSON-об'єкта [17]. Токен складається з трьох частин, закодованих у Base64Url та розділених крапками: заголовка (header) з вказанням алгоритму підпису, корисного навантаження (payload) з клеймами (claims) – ідентифікатором користувача, терміном дії та іншими даними – та підпису (signature), що гарантує цілісність токена. Сервер верифікує підпис без звернення до бази даних, що відповідає stateless-підходу REST API.

Паролі користувачів зберігаються у базі даних виключно у вигляді хешів, отриманих за допомогою алгоритму bcrypt. Bcrypt є адаптивним алгоритмом хешування паролів, що включає вбудований «сіль»-фактор (salt) – випадковий рядок, що додається до пароля перед хешуванням для захисту від атак за попередньо обчисленими таблицями (rainbow tables). Параметр cost factor визначає обчислювальну складність хешування і може збільшуватися з часом відповідно до зростання обчислювальних потужностей, зберігаючи актуальність захисту [44].

Таблиця 2.8 – Порівняльний аналіз механізмів автентифікації

Критерій	**JWT (Stateless)**	**Сесійна автентифікація**
Збереження стану на сервері	Ні	Так (БД або Redis)
Масштабованість	Висока	Потребує спільного сховища
Відповідність REST API	Висока (stateless)	Обмежена
Складність реалізації	Низька	Середня
Термін дії токена	Налаштовується (exp)	Налаштовується (session TTL)
Захист від підробки	HMAC-підпис (Hash-based Message Authentication Code)	Випадковий ідентифікатор

Для розроблюваного застосунку обрано JWT-автентифікацію у поєднанні з bcrypt для хешування паролів. JWT природно поєднується зі stateless REST API, не потребує додаткового сховища сесій та спрощує горизонтальне масштабування у майбутньому.

2.6 Аналіз API для ботанічної ідентифікації рослин

Ботанічна ідентифікація рослин за фотографією є центральною функцією застосунку. Від точності та зручності API цього класу безпосередньо залежить якість результатів, що отримує користувач. Розглянуто три сервіси: Pl@ntNet API, iNaturalist API та Google Vision API.

2.6.1 Google Vision API

Google Vision API – універсальний сервіс комп'ютерного зору від Google Cloud, що підтримує широкий спектр задач: розпізнавання об'єктів, тексту (OCR), облич, орієнтирів та вмісту зображень. Попри загальну точність моделей Google, сервіс не є спеціалізованим на ботанічній таксономії. Він може визначити, що на зображенні є рослина або квітка, однак ідентифікація до рівня виду (species-level) з наукової точністю не є основним призначенням сервісу [45]. Крім того, Google Vision API є повністю платним: безкоштовний рівень надається лише у межах обмеженого тріалу, що є суттєвим обмеженням для прототипу.

2.6.2 iNaturalist API

iNaturalist API надає програмний доступ до бази спостережень платформи та її механізму ідентифікації організмів. API підтримує пошук за видами та отримання агрегованих даних про спостереження. Проте основне призначення iNaturalist API – робота з накопиченими даними спільноти (завантаження спостережень, пошук за координатами, отримання статистики), а не ідентифікація довільної фотографії у режимі реального часу. Відсутність параметра мови для отримання загальних назв рослин українською є додатковим обмеженням [22].

2.6.3 Pl@ntNet API

Pl@ntNet API – спеціалізована платформа ботанічної ідентифікації, розроблена консорціумом французьких наукових установ (INRIA, CIRAD, INRAE, IRD) у рамках проєкту Pl@ntNet, що стартував у 2013 році [9]. База

даних платформи охоплює понад 78 000 видів рослин, а модель ідентифікації регулярно оновлюється завдяки верифікованим внескам спільноти.

З технічного боку запит до Pl@ntNet API формується як HTTP POST із типом вмісту multipart/form-data: зображення передається як бінарний файл разом з параметрами organs (тип органу рослини: leaf, flower, fruit, bark, auto) та lang (код мови для загальних назв). Сервіс повертає JSON-відповідь із відсортованим списком найімовірніших видів, де кожен містить: scientificName (латинська назва), commonNames (загальні назви обраною мовою) та score – показник впевненості у діапазоні від 0 до 1. Поріг впевненості 0.3 обрано як мінімально допустимий: результати нижче цього значення відображаються з відповідним попередженням для користувача [9].

Критичною перевагою є підтримка параметра lang=uk, що забезпечує отримання офіційних українських ботанічних назв з наукової бази Pl@ntNet без необхідності додаткового перекладу через LLM. Безкоштовний рівень складає 500 запитів на добу, що є достатнім для розробки та захисту дипломного проєкту.

Порівняльний аналіз розглянутих API для ботанічної ідентифікації рослин наведено у таблиці 2.9.

Таблиця 2.9 – Порівняльний аналіз API для ботанічної ідентифікації

Критерій	**Pl@ntNet API**	**iNaturalist API**	**Google Vision API**
Спеціалізація на рослинах	Так (78 000+ видів)	Часткова (всі організми)	Ні (загальний Computer Vision)
Безкоштовний рівень	500 запитів/добу	Необмежено	Лише тріал
Параметр мови (lang=uk)	Так (60 мов)	Ні	Ні
Confidence score	Так (0–1)	Так	Так (але не ботанічний)

Кінець таблиці 2.9

Критерій	Pl@ntNet API	iNaturalist API	Google Vision API
Формат відповіді	JSON (REST)	JSON (REST)	JSON (REST)
Наукова точність таксономії	Висока	Висока	Низька
Підтримка multipart/form-data	Так	Ні	Так (base64)

На основі проведеного аналізу для реалізації функції ботанічної ідентифікації обрано Pl@ntNet API як єдиний серед розглянутих сервісів, що поєднує вузьку ботанічну спеціалізацію, безкоштовний доступ, підтримку офіційних україномовних назв через параметр lang та структурований показник впевненості результату.

2.7 Аналіз AI-моделей для генерації медичних довідок

Автоматична генерація структурованих медичних довідок є ключовою відмінністю розроблюваного застосунку від існуючих аналогів. Для реалізації цієї функції розглянуто три провідних постачальника великих мовних моделей: Google Gemini API, OpenAI GPT API та Anthropic Claude API.

2.7.1 OpenAI GPT API

OpenAI GPT API надає доступ до серії моделей GPT-4 та GPT-4o, що демонструють високу якість генерації тексту, підтримку структурованого JSON-виводу через режим JSON mode та широкі можливості промпт-інжинірингу [46]. Проте безкоштовний рівень API є обмеженим і надається лише у вигляді тимчасового тріалу з невеликою кількістю кредитів. Контекстне

вікно моделей GPT-4 обмежене 128K токенів. Для прототипу, що генерує медичні довідки при кожній новій ідентифікації виду рослини, відсутність стабільного безкоштовного рівня є суттєвим практичним обмеженням.

2.7.2 Anthropic Claude API

Anthropic Claude API пропонує моделі серії Claude 3 та Claude 4 з акцентом на безпеку та корисність генерованого контенту. Claude демонструє високу якість при роботі зі структурованими запитами, медичними текстами та дотриманні форматних обмежень у відповідях [47]. Контекстне вікно до 200K токенів є значним. Проте, як і OpenAI, Anthropic не надає стабільного безкоштовного рівня для комерційного API-доступу, що є визначальним обмеженням для прототипу дипломного проєкту.

2.7.3 Google Gemini API

Google Gemini API надає доступ до серії мультимодальних моделей від Google DeepMind. Модель gemini-2.5-flash-lite пропонує стабільний безкоштовний рівень з лімітом 1000 запитів на добу – достатнім для розробки та захисту дипломного проєкту [48]. Gemini підтримує нативну генерацію структурованого JSON-виводу, що дозволяє безпосередньо зберігати згенеровану довідку у базі даних без додаткової обробки або парсингу.

Промпт для генерації медичної довідки формується англійською мовою і містить: латинську назву виду рослини, отриману від Pl@ntNet API, та інструкцію повернути відповідь у форматі JSON українською мовою з конкретними полями: nameUa (українська назва), description (загальний опис), medicinalUses (лікарські властивості), contraindications (протипоказання), preparation (способи приготування), isMedicinal (булева ознака лікарськості). Gemini повертає JSON-рядок, який після очищення від можливих markdown-

обгорток (```json```) безпосередньо зберігається у полі JSONB таблиці Plant бази даних.

Взаємодія Gemini API з патерном cache-aside є архітектурно ключовою: без кешування один і той самий вид рослини (наприклад, Melissa officinalis) аналізувався б повторно при кожному запиті будь-якого з користувачів, що призводило б до вичерпання ліміту 1000 запитів на добу. Завдяки cache-aside кожен вид аналізується рівно один раз, а результат зберігається назавжди у таблиці Plant для повторного використання всіма користувачами.

На основі проведеного аналізу складено порівняльну таблицю (Таблиця 2.10).

Таблиця 2.10 – Порівняльний аналіз AI-моделей для генерації медичних довідок

Критерій	**Gemini API**	**OpenAI GPT API**	**Anthropic Claude API**
Безкоштовний рівень	1000 запитів/добу	Лише пробний період	Лише пробний період
Нативна генерація JSON	Так	Так	Так
Контекстне вікно	До 1М токенів	128К токенів	200К токенів
Якість україномовного тексту	Висока	Висока	Висока
Швидкість відповіді	Висока	Середня	Середня
Підтримка промптів системою	Так	Так	Так
Простота інтеграції (SDK)	Так (@google/generative-ai)	Так (openai)	Так (@anthropic-ai/sdk)

На основі проведеного аналізу для генерації медичних довідок обрано Gemini API з моделлю gemini-2.5-flash-lite. Визначальними факторами стали стабільний безкоштовний рівень доступу обсягом 1000 запитів на добу, що є достатнім для розробки та захисту прототипу, нативна підтримка структурованого JSON-виводу, яка дозволяє безпосередньо зберігати згенеровану довідку у базі даних без додаткової обробки, а також висока якість україномовної генерації при формуванні промпту англійською мовою. Поєднання цих характеристик робить Gemini API оптимальним вибором для автоматизованого збагачення ботанічних даних структурованою медичною інформацією в умовах прототипної розробки.

2.8 Схема взаємодії компонентів системи

На основі проведеного аналізу сформовано технологічний стек та визначено схему взаємодії компонентів розроблюваного застосунку. Система побудована за трирівневою клієнт-серверною архітектурою. Схема взаємодії компонентів системи наведена на рисунку 2.1.

Клієнтський рівень реалізовано на React із Tailwind CSS для стилізації та Leaflet.js для інтерактивної карти. Користувач взаємодіє з інтерфейсом через браузер: завантажує фотографію рослини, переглядає каталог, додає мітки на карту вручну або через геолокацію та відстежує історію пошуків. Усі запити до сервера формуються у форматі JSON та надсилаються через HTTPS з JWT-токеном у заголовку Authorization Bearer.

Серверний рівень побудовано на Node.js з Express.js і організовано у три внутрішніх шари: маршрутизатори (routing) приймають HTTP-запити та делегують їх відповідним контролерам; контролери (controllers) верифікують JWT-токен для захищених маршрутів та викликають методи сервісного шару;

сервіси (services) реалізують бізнес-логіку та координують взаємодію між базою даних і зовнішніми API.

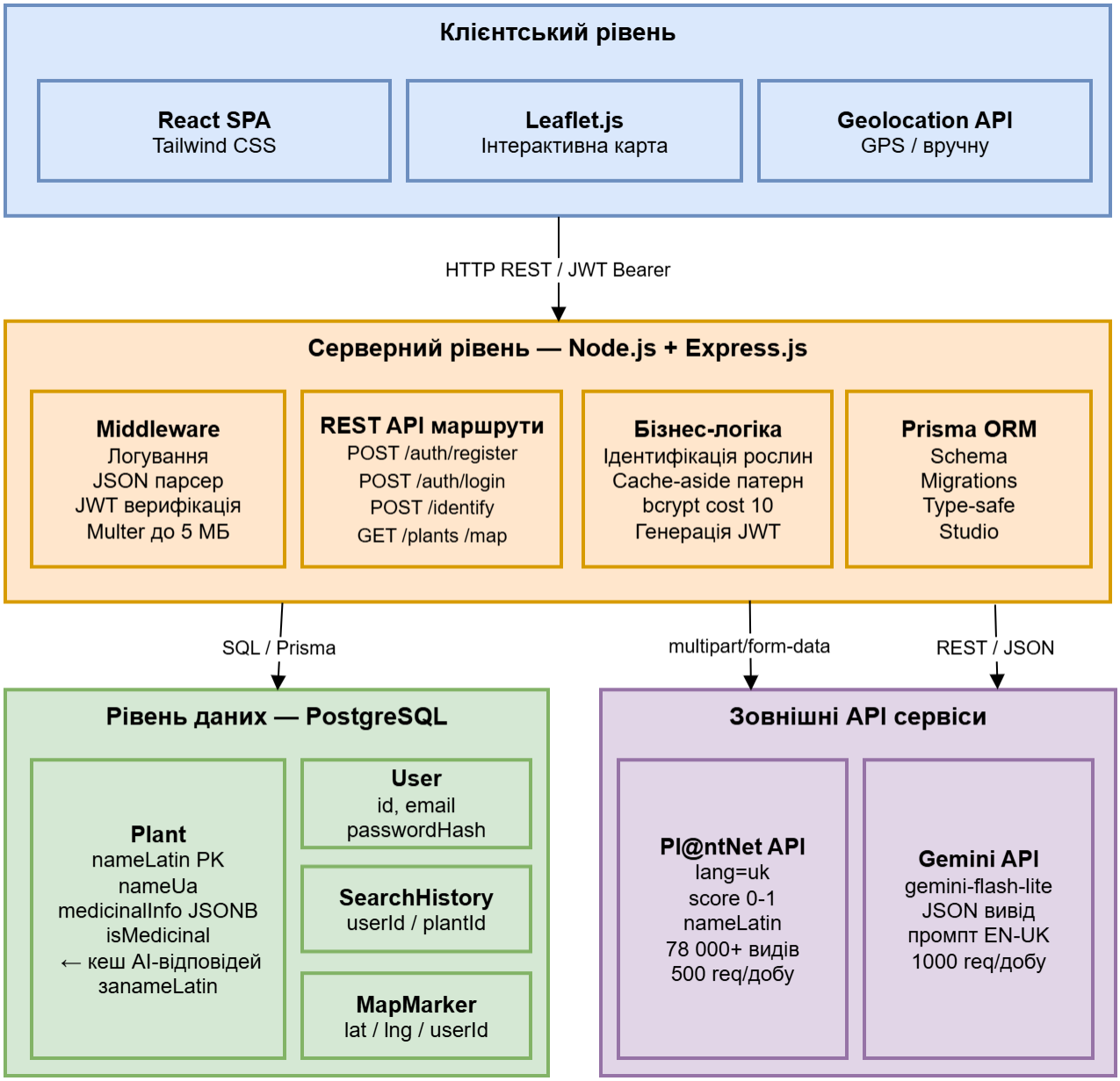


Рисунок 2.1 – Схема взаємодії компонентів системи

При надходженні запиту на ідентифікацію рослини сервер передає завантажене зображення до Pl@ntNet API у форматі multipart/form-data та отримує JSON-відповідь з латинською назвою виду і confidence score. Далі реалізується патерн cache-aside: сервіс перевіряє наявність запису у таблиці Plant за полем nameLatin через Prisma ORM. Якщо запис існує – медична довідка повертається з кешу без звернення до Gemini API; якщо ні – формується

структурований промпт, надсилається запит до Gemini API, отримана JSONB-довідка зберігається у таблиці Plant та повертається клієнту. Паралельно запис про пошук зберігається у таблиці SearchHistory незалежно від результату кешування.

Рівень даних і зовнішніх сервісів включає базу даних PostgreSQL (доступ через Prisma ORM), Pl@ntNet API та Gemini API. Зовнішні сервіси взаємодіють виключно з серверною частиною, що унеможливлює передачу API-ключів на клієнтську сторону та забезпечує безпеку конфіденційних даних.

ВИСНОВКИ ДО РОЗДІЛУ

У другому розділі проведено комплексний аналіз методів та технологій для реалізації застосунку, результатом якого є цілісний технологічний стек, де кожне рішення обумовлене як функціональними вимогами розділу 1, так і взаємною сумісністю компонентів між собою.

Архітектурну основу системи формує монолітний клієнт-серверний підхід з внутрішнім шаруванням. Це свідомий компроміс: мікросервісна архітектура забезпечила б вищу масштабованість окремих компонентів, однак внесла б операційну складність, непропорційну масштабу проєкту з одним розробником і чітко визначеним функціональним периметром. Внутрішнє шарування при цьому зберігає розподіл відповідальності та залишає можливість для майбутнього рефакторингу без зміни зовнішніх інтерфейсів.

JavaScript як єдина мова для клієнтської та серверної частин не просто спрощує стек – він дозволяє повторно використовувати логіку валідації між рівнями, усуває необхідність перетворення типів даних на межі клієнт-сервер та забезпечує природну інтеграцію між React, Node.js і Prisma ORM. Вибір React з Tailwind CSS і Leaflet.js на фронтенді та Node.js з Express.js на бекенді відповідає функціональним вимогам ФВ-01, ФВ-04 і ФВ-05: компонентна модель React забезпечує декларативне управління інтерфейсом ідентифікації, Tailwind CSS – адаптивність на екранах від 375px відповідно до нефункціональної вимоги зручності, а Leaflet.js – повнофункціональну безкоштовну інтерактивну карту знаходжень.

Вибір PostgreSQL з нативним типом JSONB є прямою відповіддю на архітектурне рішення зберігати AI-відповіді Gemini API як структурований документ без нормалізації у додаткові таблиці. Це стає можливим завдяки патерну cache-aside: оскільки медична довідка генерується рівно один раз на вид і зберігається назавжди, складна реляційна структура для цих даних є

надлишковою – JSONB з GIN-індексами вирішує задачу ефективніше і з меншою кількістю таблиць. Prisma ORM доповнює цю схему декларативною схемою та автогенерацією типізованого клієнта, що знижує ризик розходження між схемою бази даних і типами у коді.

Pl@ntNet API і Gemini API закривають дві ключові функціональні вимоги – ФВ-01 і ФВ-02 – які неможливо реалізувати без зовнішніх спеціалізованих сервісів: перший забезпечує ботанічну ідентифікацію з confidence score і підтримкою lang=uk, другий – генерацію структурованих медичних довідок у форматі JSON безпосередньо придатному для збереження у базі даних. JWT-автентифікація у поєднанні з bcrypt об'єднана з цими сервісами спільним принципом: усі зовнішні взаємодії та конфіденційні операції відбуваються виключно на серверній стороні, що унеможливлює витік API-ключів і відповідає нефункціональній вимозі безпеки з підрозділу 1.5.2.

Таким чином, обраний технологічний стек є не набором незалежних рішень, а взаємоузгодженою архітектурою, де кожен компонент підсилює інший. Єдина мова знижує складність інтеграції, JSONB і cache-aside разом мінімізують звернення до зовнішніх API, а серверна ізоляція зовнішніх викликів забезпечує безпеку системи. Разом ці рішення формують архітектуру, що відповідає як функціональним, так і нефункціональним вимогам, сформульованим за результатами аналізу предметної області у розділі 1.

РОЗДІЛ 3. РОЗРОБКА ТА РЕАЛІЗАЦІЯ СИСТЕМИ

Третій розділ присвячено практичній реалізації вебзастосунку «Інтерактивний довідник лікарських рослин на основі застосування штучного інтелекту». Описано архітектуру системи, схему бази даних, серверну та клієнтську частини, а також підсистему ідентифікації рослин.

3.1 Архітектура та структура застосунку

3.1.1 Загальна архітектура системи

Застосунок реалізований за трирівневою клієнт-серверною архітектурою. Кожен рівень має чітко визначену відповідальність і взаємодіє із сусіднім виключно через задокументований інтерфейс.

Клієнтський рівень являє собою React SPA (Single Page Application), що виконується у браузері користувача. Він відповідає за відображення інтерфейсу, маршрутизацію між сторінками та надсилання HTTP-запитів до серверного API. Клієнт ніколи не звертається до зовнішніх сервісів безпосередньо – усі запити до Pl@ntNet та Gemini API виконуються виключно через сервер.

Серверний рівень побудовано на Node.js з Express.js і організовано у три внутрішніх шари. Перший шар – маршрутизатори (routes/): auth.js, plants.js, search.js, map.js і profile.js – приймають вхідні HTTP-запити та визначають їх обробники. Другий шар – middleware/auth.js – реалізує наскрізну JWT-верифікацію для захищених маршрутів. Третій шар містить безпосередню бізнес-логіку: звернення до бази даних через Prisma ORM, виклики зовнішніх API та формування відповіді. Критично важливим архітектурним рішенням є те, що API-ключі Pl@ntNet і Gemini зберігаються виключно у файлі server/.env

і зчитуються через process.env – клієнтський код не має до них жодного доступу.

Рівень даних реалізований на PostgreSQL з доступом через Prisma ORM. Prisma виконує подвійну роль: типізований клієнт для виконання запитів і система міграцій для керованого розвитку схеми. Зовнішні API-сервіси – Pl@ntNet для ботанічної ідентифікації, Gemini для генерації медичних довідок і Cloudinary для збереження фотографій – є незалежними компонентами, з якими сервер взаємодіє через HTTPS.

3.1.2 Структура директорій проєкту

Проєкт організовано як монорепозиторій з двома кореневими директоріями: client/ і server/. Такий підхід зберігає єдину кодову базу в одному репозиторії, спрощує контроль версій і дозволяє незалежно розгортати кожну частину.

Серверна директорія server/ має таку структуру: routes/ містить п'ять файлів маршрутизаторів, кожен з яких відповідає за окремий ресурс; middleware/ містить auth.js – єдину точку JWT-верифікації, яка підключається до кожного захищеного маршруту; config/ зберігає ініціалізацію Cloudinary, виокремлену з маршрутизаторів для уникнення дублювання; prisma/ містить schema.prisma і директорію migrations/ з усіма міграціями.

Клієнтська директорія client/src/ поділена на pages/ – сім повноекранних сторінок (Home, Catalogue, PlantDetail, MapPage, Profile, Login, Register) – та components/ – основні UI-компоненти застосунку: Navbar, Footer, Layout, PlantCard, SearchBar, UploadPhoto, Spinner та ProtectedRoute. Файл services/api.js є єдиною точкою всіх HTTP-звернень до серверного API: він ініціалізує axios з базовим URL, налаштовує interceptor для автоматичного додавання JWT-заголовка і експортує іменовані функції для кожного ендпоінту. Завдяки цьому URL-адреси та логіка авторизації не дублюються у компонентах.

Структуру директорій клієнтської та серверної частин проєкту наведено на рисунках 3.1 та 3.2 відповідно.

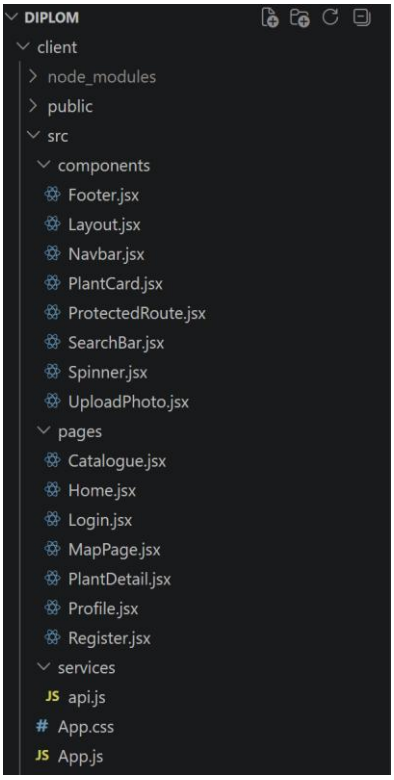


Рисунок 3.1 – Структура директорій клієнтської частини (client/src/)

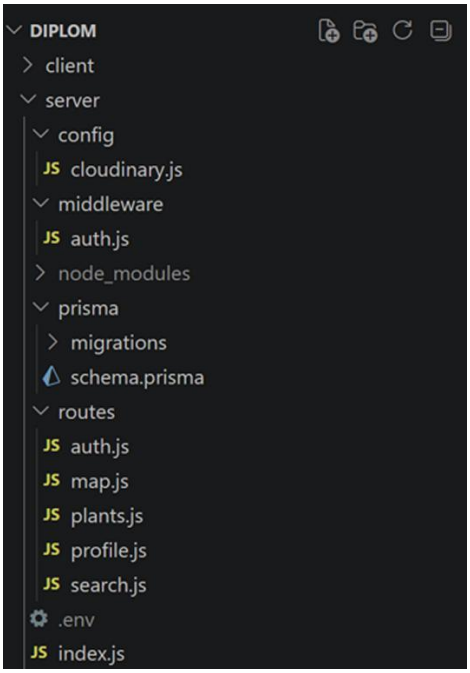


Рисунок 3.2 – Структура директорій серверної частини (server/)

3.2 Проєктування та реалізація бази даних

3.2.1 Схема бази даних та зв'язки між таблицями

База даних PostgreSQL складається з п'яти таблиць: User, Plant, SearchHistory, MapMarker та PlantProperty.

Таблиця User зберігає облікові дані користувачів: унікальний ідентифікатор у форматі UUID, ім'я користувача, email та хеш пароля. Таблиця Plant є центральною сутністю схеми і виконує подвійну роль: слугує каталогом лікарських рослин і одночасно кешем результатів AI-аналізу. Таблиця SearchHistory фіксує кожне звернення користувача до функції ідентифікації. Таблиця MapMarker зберігає геопросторові мітки з GPS-координатами та прив'язкою до конкретної рослини й автора. Таблиця PlantProperty реалізує розширювані характеристики рослин за патерном EAV (Entity-Attribute-Value), де кожен запис містить три поля: ідентифікатор рослини, тип властивості та її значення. Такий підхід дозволяє додавати нові характеристики без зміни схеми основних таблиць.

Ключовим архітектурним рішенням є розмежування між таблицями Plant і SearchHistory. Таблиця Plant кешує результати AI-аналізу на рівні біологічного виду: після першої ідентифікації певного виду його медична довідка зберігається одноразово і повторно використовується для всіх наступних запитів від будь-яких користувачів. Таблиця SearchHistory фіксує індивідуальну дію конкретного користувача: момент звернення, завантажене фото та показник впевненості класифікатора. Отже, Plant є спільним кешем на рівні системи, тоді як SearchHistory – персональним журналом дій користувача.

Зв'язки між таблицями побудовано за принципом «один до багатьох»: один User може мати багато записів SearchHistory та MapMarker; одна Plant може

бути пов'язана з багатьма SearchHistory, MapMarker і PlantProperty. ER-діаграму наведено у додатку Б.

3.2.2 Реалізація моделей Prisma

Схему бази даних описано у файлі server/prisma/schema.prisma мовою Prisma Schema Language. Цей файл є основним дескриптором структури бази даних: на його основі Prisma автоматично генерує типізований клієнт та файли міграцій [42].

Модель User описує сутність зареєстрованого користувача системи. Первинний ключ реалізовано як поле id типу String з атрибутом @default(uuid()), що забезпечує автоматичну генерацію унікального ідентифікатора без використання послідовностей бази даних. Поле passwordHash призначене для зберігання bcrypt-хешу пароля, що унеможливлює збереження облікових даних користувача у відкритому вигляді. Атрибут @unique, застосований до полів username та email, гарантує унікальність облікових даних на рівні бази даних незалежно від перевірок прикладного рівня. Реалізацію моделі User наведено на рисунку 3.3.

```
model User {
  id           String          @id @default(uuid())
  username     String          @unique
  email        String          @unique
  passwordHash String
  createdAt    DateTime        @default(now())

  searchHistory SearchHistory[]
  mapMarkers    MapMarker[]
}
```

Рисунок 3.3 – Модель User у schema.prisma

Модель Plant є центральною сутністю схеми бази даних і виконує подвійну роль: слугує каталогом лікарських рослин та забезпечує реалізацію патерну cache-aside для результатів AI-аналізу. Поле nameLatin оголошено з атрибутом

@unique і виконує роль природного ключа кешу – саме за значенням цього поля система перевіряє наявність запису у базі даних перед зверненням до Gemini API, що унеможливлює повторний аналіз одного й того самого виду рослини. Поля description, medicinalUses, contraindications та preparation мають опціональний тип String? і залишаються порожніми до першого звернення до зовнішнього AI-сервісу для конкретного виду: після генерації медичної довідки їх значення зберігаються у базі даних та використовуються повторно для всіх наступних запитів. Поле isMedicinal булевого типу визначає, чи є рослина лікарською, а поле photoUrl зберігає URL офіційного фото рослини. Реалізацію моделі Plant наведено на рисунку 3.4.

```
model Plant {
  id             String          @id @default(uuid())
  nameUa         String
  nameLatin      String          @unique
  description    String?
  medicinalUses  String?
  contraindications String?
  preparation    String?
  isMedicinal    Boolean         @default(false)
  photoUrl       String?
  externalApiId  String?
  updatedAt      DateTime        @updatedAt

  searchHistory  SearchHistory[]
  mapMarkers     MapMarker[]
  properties     PlantProperty[]
}
```

Рисунок 3.4 – Модель Plant у schema.prisma

Модель SearchHistory фіксує персональний журнал ідентифікацій кожного користувача. Поле confidenceScore типу Float зберігає показник впевненості класифікатора Pl@ntNet у діапазоні від 0.0 до 1.0. Поле photoUrl у даній моделі містить URL фотографії, завантаженої користувачем до Cloudinary під час конкретного сеансу ідентифікації, що принципово відрізняється від поля Plant.photoUrl, яке зберігає офіційне ботанічне зображення виду. Поле plantId є nullable: у разі якщо Pl@ntNet не розпізнав рослину на зображенні, запис у

таблиці Plant не створюється, однак факт спроби ідентифікації фіксується у SearchHistory. Реалізацію моделі SearchHistory наведено на рисунку 3.5.

```
model SearchHistory {
  id              String   @id @default(uuid())
  userId          String
  plantId         String?
  photoUrl        String?
  confidenceScore Float?
  aiAnalysis      String?
  searchedAt      DateTime @default(now())

  user  User   @relation(fields: [userId], references: [id])
  plant Plant? @relation(fields: [plantId], references: [id])
}
```

Рисунок 3.5 – Модель SearchHistory у schema.prisma

Модель MapMarker описує геопросторову мітку, розміщену користувачем на інтерактивній карті. Координати місцезнаходження зберігаються у полях latitude та longitude типу Float. Поле note є опціональним і призначене для текстового коментаря до мітки. Зовнішні ключі userId та plantId забезпечують посилальну цілісність між таблицями. Модель PlantProperty реалізує зберігання розширюваних характеристик рослин за патерном EAV: кожен запис містить поле propertyType з назвою характеристики та поле value з її значенням. Реалізацію моделей MapMarker та PlantProperty наведено на рисунку 3.6.

```
model MapMarker {
  id        String   @id @default(uuid())
  userId    String
  plantId   String
  latitude  Float
  longitude Float
  note      String?
  createdAt DateTime @default(now())

  user  User  @relation(fields: [userId], references: [id])
  plant Plant @relation(fields: [plantId], references: [id])
}

model PlantProperty {
  id           String @id @default(uuid())
  plantId      String
  propertyType String
  value        String

  plant Plant @relation(fields: [plantId], references: [id])
}
```

Рисунок 3.6 – Моделі MapMarker та PlantProperty у schema.prisma

3.2.3 Рішення щодо зберігання властивостей рослин

Таблиця PlantProperty реалізує патерн EAV (Entity-Attribute-Value) для зберігання розширюваних характеристик рослин. Кожен запис містить три поля: plantId – зовнішній ключ до таблиці Plant, propertyType – назва характеристики та value – її значення.

При проєктуванні схеми розглядалося два підходи до зберігання додаткових властивостей рослин. Перший передбачав додавання фіксованих колонок безпосередньо до таблиці Plant. Недоліком цього підходу є те, що поява нового типу характеристики потребує внесення змін до схеми основних таблиць та виконання міграції бази даних. Другий підхід – застосування патерну EAV – позбавлений цього обмеження: нові типи характеристик додаються як окремі рядки таблиці PlantProperty без жодних структурних змін схеми. Обраний підхід відповідає нефункціональній вимозі масштабованості, сформульованій у підрозділі 1.5.2.

3.3 Реалізація серверної частини

3.3.1 Налаштування Express-сервера та middleware

Точкою входу серверної частини є файл server/index.js. Після завантаження змінних середовища через dotenv ініціалізується Express-застосунок і підключається ланцюжок middleware.

Middleware cors() забезпечує обробку крос-доменних запитів відповідно до механізму CORS, що дозволяє клієнтській частині, розгорнутій на іншому домені або порті, звертатися до серверного API [43]. Middleware express.json() виконує автоматичний парсинг тіла HTTP-запиту у форматі JSON і поміщає

результат у об'єкт req.body, що дозволяє маршрутизаторам безпосередньо зчитувати передані дані без додаткової обробки.

П'ять маршрутизаторів підключаються до відповідних префіксів: /api/auth, /api/plants, /api/search, /api/map та /api/profile. Кожен маршрутизатор відповідає за окрему предметну область і розвивається незалежно, що забезпечує модульність серверної частини. Реалізацію файлу index.js наведено на рисунку 3.7.

```
require('dotenv').config();
const express = require('express');
const cors = require('cors');

const authRoutes = require('./routes/auth');
const plantsRoutes = require('./routes/plants');
const searchRoutes = require('./routes/search');
const mapRoutes = require('./routes/map');
const profileRoutes = require('./routes/profile');

const app = express();
const PORT = process.env.PORT || 5000;

app.use(cors());
app.use(express.json());

app.use('/api/auth', authRoutes);
app.use('/api/plants', plantsRoutes);
app.use('/api/search', searchRoutes);
app.use('/api/map', mapRoutes);
app.use('/api/profile', profileRoutes);

app.get('/', (req, res) => {
  res.json({ message: 'Medicinal Plants API is running' });
});

app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

Рисунок 3.7 – Ініціалізація Express-сервера та підключення маршрутизаторів (server/index.js)

3.3.2 Реалізація автентифікації та авторизації

Підсистема автентифікації реалізована у файлі server/routes/auth.js і складається з двох маршрутів. Важливо розрізняти поняття автентифікації –

перевірки особи користувача – та авторизації – перевірки прав доступу до ресурсу.

Маршрут POST /api/auth/register реалізує реєстрацію нового користувача. Після валідації обов'язкових полів та перевірки унікальності email пароль хешується функцією bcrypt.hash(password, 10). Параметр 10 – це cost factor, що визначає обчислювальну складність алгоритму: кожне збільшення на одиницю подвоює час генерації хешу. Значення 10 є рекомендованим балансом між захистом від brute-force атак і прийнятним часом відповіді сервера [44]. Хеш зберігається у полі passwordHash таблиці User – відкритий текст паролю ніколи не потрапляє у базу даних.

Маршрут POST /api/auth/login виконує перевірку облікових даних. Після знаходження користувача за email функція bcrypt.compare() порівнює наданий пароль з хешем. Ця функція виконує операцію за постійний час незалежно від результату, що унеможливлює атаки з вимірюванням часу відповіді (timing attacks). У разі збігу генерується JWT-токен з корисним навантаженням {userId}, підписаний секретом зі змінної середовища JWT_SECRET та терміном дії 7 днів. Токен разом з username повертається клієнту і зберігається у localStorage.

Авторизація реалізована через middleware у файлі server/middleware/auth.js. Функція зчитує заголовок Authorization у форматі Bearer <token>, верифікує підпис через jwt.verify() та записує витягнутий userId до об'єкта req. Усі захищені маршрути – POST /api/search/identify, POST та DELETE /api/map, GET /api/profile – підключають цей middleware першим у ланцюжку обробників. Реалізацію middleware та маршруту логіну наведено на рисунках 3.8 та 3.9.

```
const jwt = require('jsonwebtoken');

module.exports = (req, res, next) => {
  const authHeader = req.headers['authorization'];
  const token = authHeader && authHeader.split(' ')[1]; // Bearer <token>

  if (!token) {
    return res.status(401).json({ error: 'Access denied. No token provided.' });
  }

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.userId = decoded.userId;
    next();
  } catch (err) {
    return res.status(401).json({ error: 'Invalid or expired token.' });
  }
};
```

Рисунок 3.8 – JWT middleware для верифікації токена (server/middleware/auth.js)

```
// POST /api/auth/login
router.post('/login', async (req, res) => {
  const { email, password } = req.body;

  if (!email || !password) {
    return res.status(400).json({ error: 'Email and password are required' });
  }

  const user = await prisma.user.findUnique({ where: { email } });
  if (!user) {
    return res.status(401).json({ error: 'Invalid email or password' });
  }

  const passwordMatch = await bcrypt.compare(password, user.passwordHash);
  if (!passwordMatch) {
    return res.status(401).json({ error: 'Invalid email or password' });
  }

  const token = jwt.sign(
    { userId: user.id },
    process.env.JWT_SECRET,
    { expiresIn: '7d' }
  );

  res.json({ token, username: user.username });
});
```

Рисунок 3.9 – Маршрут автентифікації POST /api/auth/login (server/routes/auth.js)

3.3.3 Реалізація REST API маршрутів

Застосунок реалізує десять REST API маршрутів, згрупованих у п'ять маршрутизаторів. Повний перелік маршрутів наведено у таблиці 3.1.

Таблиця 3.1 – Перелік REST API маршрутів застосунку

Метод	Маршрут	Авторизація	Опис
POST	/api/auth/register	Ні	Реєстрація нового користувача
POST	/api/auth/login	Ні	Вхід; повертає JWT-токен та username
GET	/api/plants	Ні	Каталог рослин; параметр search для текстового пошуку
GET	/api/plants/:id	Ні	Повна картка рослини з властивостями
GET	/api/plants/:id/adjacent	Ні	Сусідні рослини для навігації
POST	/api/search/identify	Так	Ідентифікація рослини за фото (Pl@ntNet + Gemini)
GET	/api/map	Ні	Усі геомітки з інформацією про рослину та автора
POST	/api/map	Так	Додати геомітку (plantId, latitude, longitude, note)
DELETE	/api/map/:id	Так	Видалити власну геомітку
GET	/api/profile	Так	Дані користувача + повна історія пошуків

Маршрутизатор /api/auth реалізує реєстрацію та вхід користувача. При успішному вході клієнту повертаються два поля: JWT-токен для авторизації подальших запитів та username для відображення в інтерфейсі. При невдалій спробі входу обидва варіанти помилки – користувач не знайдений та невірний

пароль – повертають однакове повідомлення «Invalid email or password», що унеможливлює визначення зловмисником наявності конкретного email у базі даних.

Маршрутизатор /api/plants реалізує три ендпоінти. Маршрут GET /api/plants підтримує текстовий пошук через необов'язковий параметр ?search=, який виконується одночасно по полях nameUa та nameLatin з режимом insensitive, що забезпечує нечутливість до регістру символів. За замовчуванням рослини сортуються за алфавітом полем nameUa. Маршрут GET /api/plants/:id повертає повну картку рослини з включенням пов'язаних записів з таблиці PlantProperty через Prisma include. Маршрут GET /api/plants/:id/adjacent реалізує алфавітну навігацію між рослинами: через два паралельні запити у Promise.all() знаходяться попередня і наступна рослини за значенням поля nameUa, що скорочує загальний час відповіді порівняно з послідовним виконанням запитів. Реалізацію текстового пошуку у каталозі рослин наведено на рисунку 3.10.

```
const express = require('express');
const router = express.Router();
const { PrismaClient } = require('@prisma/client');

const prisma = new PrismaClient();

// GET /api/plants
router.get('/', async (req, res) => {
  const { search } = req.query;

  const plants = await prisma.plant.findMany({
    where: search
      ? {
          OR: [
            { nameUa: { contains: search, mode: 'insensitive' } },
            { nameLatin: { contains: search, mode: 'insensitive' } },
          ],
        }
      : undefined,
    orderBy: { nameUa: 'asc' },
  });

  res.json(plants);
});
```

Рисунок 3.10 – Реалізація текстового пошуку у каталозі рослин (server/routes/plants.js)

Маршрутизатор /api/map реалізує три ендпоінти для роботи з геопросторовими мітками. Публічний маршрут GET /api/map повертає всі мітки

з агрегованими даними: для кожного маркера через Prisma include включаються поля пов'язаної рослини та ім'я автора. Захищений маршрут POST /api/map виконує валідацію вхідних даних у два етапи: спочатку перевіряється наявність обов'язкових полів plantId, latitude та longitude, після чого координати перетворюються через parseFloat() і перевіряються функцією isNaN() для захисту від передачі нечислових значень. Перед збереженням мітки додатково перевіряється існування рослини з вказаним plantId. Маршрут DELETE /api/map/:id реалізує авторизоване видалення: система порівнює marker.userId з req.userId, витягнутим з JWT-токена, і повертає HTTP 403 Forbidden у разі невідповідності, що унеможливлює видалення чужих міток навіть за наявності валідного токена. Реалізацію маршруту POST /api/map наведено на рисунку 3.11.

```
// POST /api/map — захищений, додати маркер
router.post('/', authMiddleware, async (req, res) => {
  const { plantId, latitude, longitude, note } = req.body;

  if (!plantId || latitude === undefined || longitude === undefined) {
    return res.status(400).json({ error: 'plantId, latitude and longitude are required.' });
  }

  const lat = parseFloat(latitude);
  const lon = parseFloat(longitude);

  if (isNaN(lat) || isNaN(lon)) {
    return res.status(400).json({ error: 'latitude and longitude must be valid numbers.' });
  }

  const plant = await prisma.plant.findUnique({ where: { id: plantId } });
  if (!plant) {
    return res.status(404).json({ error: 'Plant not found.' });
  }

  const marker = await prisma.mapMarker.create({
    data: {
      userId: req.userId,
      plantId,
      latitude: lat,
      longitude: lon,
      note: note || null,
    },
    include: {
      plant: { select: { nameUa: true, nameLatin: true, photoUrl: true } },
      user: { select: { username: true } },
    },
  });

  res.status(201).json(marker);
});
```

Рисунок 3.11 – Маршрут додавання геопросторової мітки (server/routes/map.js)

Маршрутизатор /api/profile реалізує єдиний захищений маршрут GET /api/profile. Запит до бази даних використовує вкладений select для вибірки лише необхідних полів: з таблиці User повертаються username, email та createdAt без поля passwordHash. Для кожного запису SearchHistory через include включаються поля пов'язаної рослини – id, nameUa, nameLatin та photoUrl. Записи відсортовані за полем searchedAt у спадному порядку. Реалізацію маршруту GET /api/profile наведено на рисунку 3.12.

```
// GET /api/profile
router.get('/', authMiddleware, async (req, res) => {
  const user = await prisma.user.findUnique({
    where: { id: req.userId },
    select: {
      username: true,
      email: true,
      createdAt: true,
      searchHistory: {
        orderBy: { searchedAt: 'desc' },
        include: {
          plant: { select: { id: true, nameUa: true, nameLatin: true, photoUrl: true } },
        },
      },
    },
  });

  if (!user) {
    return res.status(404).json({ error: 'User not found.' });
  }
  res.json(user);
});
```

Рисунок 3.12 – Маршрут GET /api/profile (server/routes/profile.js)

3.4 Реалізація підсистеми ідентифікації рослин із залученням зовнішніх API

3.4.1 Загальна послідовність обробки запиту та завантаження фотографії

Маршрут POST /api/search/identify є найскладнішим у системі і реалізує послідовність з п'яти кроків. Перший крок – обробка завантаженого файлу та його збереження до Cloudinary. Другий крок – передача зображення до Pl@ntNet API для ботанічної ідентифікації. Третій крок – реалізація патерну

cache-aside: перевірка наявності рослини в базі даних. Четвертий крок – умовне звернення до Gemini API у разі відсутності рослини в кеші. П'ятий крок – незалежно від результату кешування, факт ідентифікації записується у SearchHistory і відповідь повертається клієнту. Блок-схему послідовності обробки запиту наведено на рисунку 3.13.

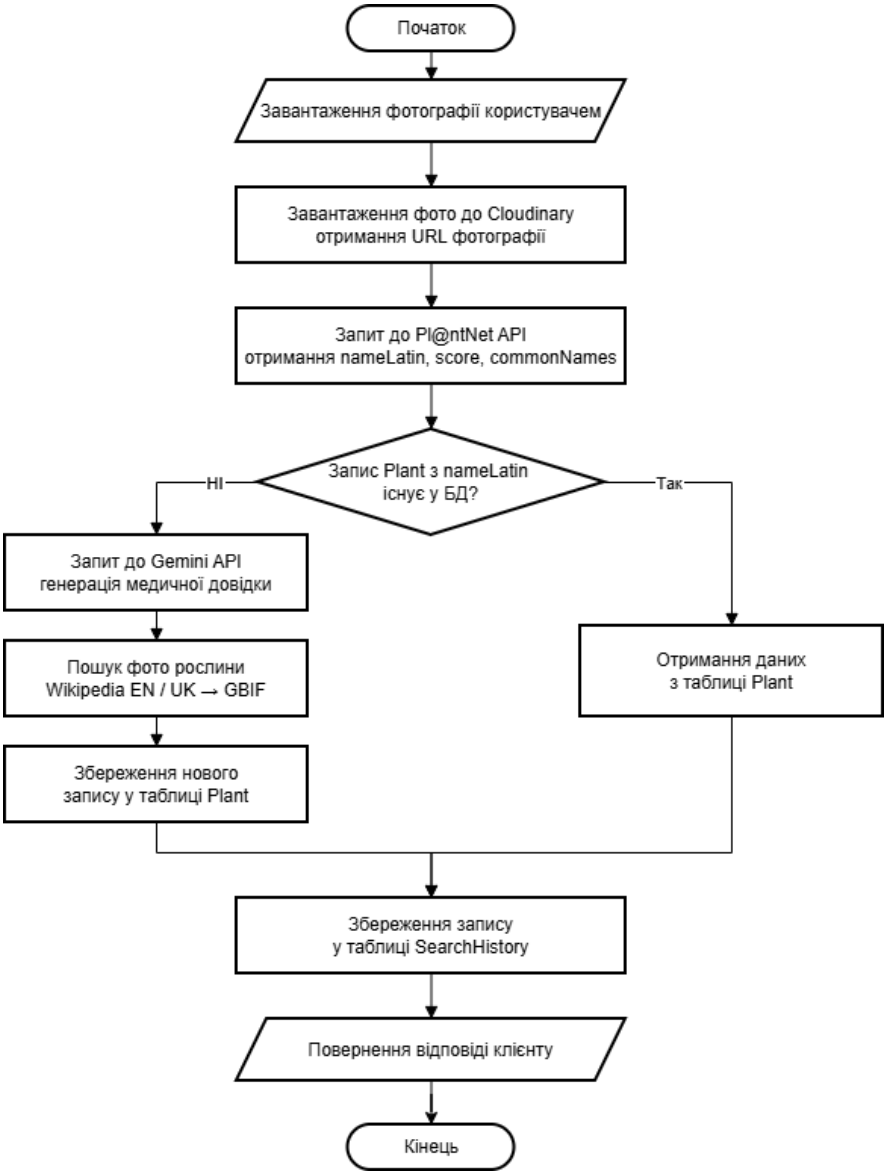


Рисунок 3.13 – Блок-схема алгоритму ідентифікації рослини

Обробка завантаженого файлу реалізована через Multer з конфігурацією memoryStorage. При такому налаштуванні завантажений файл не записується на диск, а зберігається у пам'яті сервера як об'єкт Buffer і стає доступним через req.file.buffer. Таке рішення обумовлене тим, що буфер безпосередньо

передається до Cloudinary та Pl@ntNet API без проміжного збереження, що є необхідним для середовищ розгортання, де тимчасовий файловий запис є ненадійним. Обмеження розміру файлу встановлено на рівні 5 МБ через параметр limits: { fileSize: 5 * 1024 * 1024 }.

Перед основним процесом ідентифікації буфер файлу надсилається до Cloudinary для отримання постійного URL фотографії користувача. Цей URL зберігається у полі photoUrl таблиці SearchHistory і відображається в особистій історії пошуків користувача. У разі помилки завантаження до Cloudinary поле залишається порожнім, однак процес ідентифікації продовжується – збій зовнішнього сервісу не перериває виконання запиту.

3.4.2 Визначення рослини через Pl@ntNet API

Запит до Pl@ntNet API формується як HTTP POST до ендпоінту https://my-api.plantnet.org/v2/identify/all з параметрами lang=uk та nb-results=1 у рядку запиту. Фото передається у форматі multipart/form-data: бінарний буфер додається до об'єкта FormData разом з параметром organs=auto, що дозволяє класифікатору самостійно визначити тип органу рослини (листок, квітка, плід, кора) на зображенні без участі користувача [9].

З відповіді Pl@ntNet витягуються чотири значення. Перше – results[0].species.scientificNameWithoutAuthor: латинська наукова назва без авторства, що використовується як ключ кешу при подальшій перевірці у базі даних. Друге – results[0].score: показник впевненості класифікатора у діапазоні 0.0-1.0. Третє – results[0].species.commonNames: масив офіційних назв обраною мовою; перший елемент зберігається у змінній nameFromApi як пріоритетне джерело для поля nameUa. Четверте – results[0].gbif.id: ідентифікатор у базі GBIF (Global Biodiversity Information Facility – глобальна інформаційна система з біорізноманіття) для каскадного пошуку фотографій. Якщо Pl@ntNet повертає

HTTP 404, маршрут відповідає клієнту HTTP 422 з повідомленням про відсутність рослини на фотографії.

Після отримання латинської назви виконується каскадний пошук офіційного фото рослини через функцію getPlantPhoto(). Функція послідовно звертається до англійської Wikipedia REST API, потім до української Wikipedia, і як запасний варіант – до GBIF occurrence API. Такий трирівневий каскад підвищує ймовірність знаходження якісного зображення для різних видів рослин. Реалізацію запиту до Pl@ntNet API наведено на рисунку 3.14.

```
// Крок 1: відправляємо фото до PlantNet
const formData = new FormData();
formData.append('images', req.file.buffer, {
  filename: req.file.originalname,
  contentType: req.file.mimetype,
});
formData.append('organs', 'auto');

let plantNetData;
try {
  const plantNetResponse = await axios.post(
    `https://my-api.plantnet.org/v2/identify/all?api-key=${process.env.PLANTNET_API_KEY}&nb-results=1&lang=uk`,
    formData,
    { headers: formData.getHeaders() }
  );
  plantNetData = plantNetResponse.data;
} catch (err) {
  if (err.response?.status === 404) {
    return res.status(422).json({ error: 'No plant detected in this photo. Please try with a clearer photo.' });
  }
  console.error('PlantNet error:', err.response?.data || err.message);
  throw err;
}

const topResult = plantNetData.results?.[0];

if (!topResult) {
  return res.status(422).json({ error: 'No plant detected. Please try with a clearer photo.' });
}

const nameLatin = topResult.species.scientificNameWithoutAuthor;
const confidenceScore = topResult.score;
const commonNames = topResult.species.commonNames;
const nameFromApi = commonNames && commonNames.length > 0 ? commonNames[0] : null;
const gbifId = topResult.gbif?.id || null;
```

Рисунок 3.14 – Запит до Pl@ntNet API та витягування даних з відповіді (server/routes/search.js)

3.4.3 Генерація медичної довідки через Gemini API

Генерація медичної довідки виконується через Gemini API з моделлю gemini-2.5-flash-lite, що надає безкоштовний рівень доступу обсягом 1000 запитів на добу [48]. Промпт формується англійською мовою і містить два ключових елементи. Перший – латинська наукова назва виду, яка є однозначним ботанічним ідентифікатором. Другий – змінна commonNameHint з офіційною українською назвою від Pl@ntNet, що орієнтує модель на використання верифікованої ботанічної номенклатури замість власного перекладу. Промпт вимагає повернути відповідь виключно у форматі JSON українською мовою з шістьма конкретними полями: nameUa, description, medicinalUses, contraindications, preparation та isMedicinal.

Відповідь Gemini API нерідко містить markdown-обгортку, що унеможливлює безпосередній виклик JSON.parse(). Для очищення відповіді застосовується регулярний вираз, що видаляє відкриваючий та закриваючий markdown-блоки. Після успішного парсингу отриманий об'єкт використовується для створення нового запису у таблиці Plant через prisma.plant.create(). У разі якщо JSON.parse() завершується помилкою, маршрут повертає HTTP 500 з оригінальним текстом відповіді для діагностики. Реалізацію формування промпту та надсилання запиту до Gemini API наведено на рисунку 3.15.

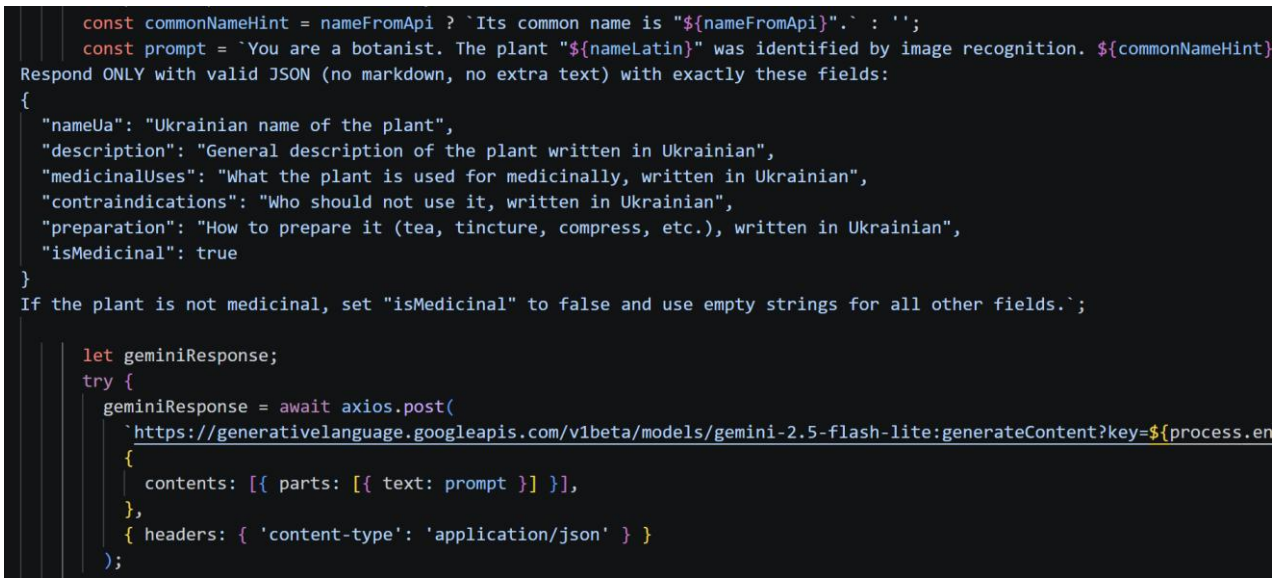


Рисунок 3.15 – Формування промпту та запит до Gemini API (server/routes/search.js)

3.4.4 Патерн cache-aside та логіка пріоритету назви

Після отримання латинської назви від Pl@ntNet система перевіряє наявність запису у таблиці Plant через prisma.plant.findUnique({where: { nameLatin}}). Це реалізує патерн cache-aside: застосунок спочатку перевіряє власне сховище і лише за відсутності запису звертається до зовнішнього API [13].

При cache hit – рослина знайдена у базі даних – медична довідка формується безпосередньо з полів таблиці Plant без виклику Gemini API. При cache miss – рослина відсутня у базі даних – виконується виклик Gemini API, після чого новий запис зберігається у таблиці Plant для повторного використання усіма наступними запитами щодо цього виду. Таким чином кожен вид аналізується Gemini API рівно один раз незалежно від кількості користувачів. Запис у таблиці SearchHistory створюється завжди – незалежно від результату перевірки кешу.

Логіка пріоритету для поля nameUa визначена у такій послідовності: по-перше, nameFromApi – перший елемент масиву commonNames від Pl@ntNet; по-друге, plantData.nameUa – назва згенерована Gemini API; по-третє, nameLatin – латинська назва як крайній резерв. Такий порядок обґрунтований тим, що Pl@ntNet надає офіційні верифіковані ботанічні назви зі своєї наукової бази, тоді як Gemini API може генерувати розмовні або неточні варіанти перекладу.

3.5 Реалізація клієнтської частини

3.5.1 Організація маршрутизації та централізований сервіс HTTP-запитів

Клієнтська маршрутизація реалізована через бібліотеку react-router-dom. Застосунок містить сім маршрутів, перелік яких наведено у таблиці 3.2.

Таблиця 3.2 – Маршрути клієнтської частини застосунку

Маршрут	Компонент	Захищений
/	Home.jsx	Ні
/catalogue	Catalogue.jsx	Ні
/plants/:id	PlantDetail.jsx	Ні
/map	MapPage.jsx	Ні
/profile	Profile.jsx	Так
/login	Login.jsx	Ні
/register	Register.jsx	Ні

Захист маршруту /profile реалізований через компонент ProtectedRoute, який перевіряє наявність JWT-токена у localStorage. За відсутності токена відбувається автоматичне перенаправлення на сторінку входу /login. Такий

підхід централізує логіку захисту маршрутів і унеможливлює доступ до особистих даних неавторизованого користувача.

Усі HTTP-звернення до серверного API централізовані у файлі client/src/services/api.js. Єдиний екземпляр Axios створюється з базовою URL-адресою сервера – завдяки цьому при зміні адреси сервера достатньо оновити одне значення. Interceptor запиту автоматично додає заголовок Authorization: Bearer <token> до кожного запиту, якщо токен наявний у localStorage. Завдяки цьому жоден компонент не передає токен вручну – авторизація відбувається прозоро для всіх компонентів застосунку. Файл експортує іменовані функції для кожного ендпоінту серверного API: register, login, getPlants, getPlant, getAdjacentPlants, identifyPlant, getMarkers, addMarker, deleteMarker та getProfile.

3.5.2 Реалізація сторінок та компонентів застосунку

Головна сторінка реалізована у компоненті Home.jsx. Хук useEffect встановлює заголовок вкладки браузера через document.title при монтуванні компонента. Функція scrollToUpload() реалізує плавне прокручування до секції завантаження через scrollIntoView з параметром behavior: smooth. На головній сторінці розміщено компонент UploadPhoto та статичний блок з трьома картками переваг застосунку, що формуються з масиву FEATURES через метод map().

Компонент UploadPhoto.jsx реалізує повний цикл ідентифікації рослини і закриває функціональні вимоги ФВ-01 та ФВ-02. Стан компонента керується через шість змінних useState: file – вибраний файл, preview – URL для попереднього перегляду, dragging – стан зони drag-and-drop, loading – стан обробки запиту, result – отриманий результат та error – повідомлення про помилку. Посилання на прихований елемент input типу file зберігається через хук useRef, що дозволяє програмно ініціювати відкриття діалогу вибору файлу при кліку на зону завантаження. Підтримка drag-and-drop реалізована без

сторонніх бібліотек через обробники onDragOver, onDragLeave та onDrop – останній отримує файл через e.dataTransfer.files[0] і перевіряє що тип файлу починається з image. Попередній перегляд вибраного зображення формується через URL.createObjectURL(). При надсиланні запиту функція handleIdentify() пакує файл у FormData та передає до identifyPlant(). Функція getConfidenceInfo() приймає числове значення score та повертає об'єкт з відсотком, текстовим повідомленням, іконкою та класами стилізації для чотирьох діапазонів впевненості: 70% і вище, 50-69%, 26-49% та менше 26%. Реалізацію функції getConfidenceInfo() наведено на рисунку 3.16.

```
function getConfidenceInfo(score) {
  const percent = Math.round(score * 100);

  if (percent >= 70) return {
    percent,
    message: 'Рослину визначено точно',
    icon: '✅',
    color: 'bg-green-100 text-green-700 border-green-200',
  };

  if (percent >= 50) return {
    percent,
    message: 'Рослина визначена, можливі схожі підвиди',
    icon: '🌿',
    color: 'bg-yellow-100 text-yellow-700 border-yellow-200',
  };

  if (percent >= 26) return {
    percent,
    message: 'Результат приблизний, рекомендуємо звірити з каталогом',
    icon: '📖',
    color: 'bg-orange-100 text-orange-700 border-orange-200',
  };

  return {
    percent,
    message: 'Низька достовірність — результат може бути неточним',
    icon: '⚠️',
    color: 'bg-red-100 text-red-700 border-red-200',
  };
}
```

Рисунок 3.16 – Функція визначення рівня впевненості класифікатора (UploadPhoto.jsx)

Сторінка каталогу реалізована у компоненті Catalogue.jsx і закриває функціональну вимогу ФВ-04. Компонент використовує три незалежних хуки useEffect. Перший встановлює заголовок сторінки. Другий реалізує debounce-механізм для пошукових запитів: при зміні значення search запускається таймер setTimeout з затримкою 300 мс, функція очищення useEffect скасовує

попередній таймер через clearTimeout при кожному новому введенні символу. Якщо поле пошуку порожнє, затримка не застосовується. Третій useEffect скидає номер поточної сторінки до першої при зміні search або filter. Фільтрація масиву рослин за полем isMedicinal виконується через Array.filter() локально без додаткових запитів. Пагінація реалізована через Array.slice де PAGE_SIZE дорівнює 15. Компонент SkeletonCard відображає анімований placeholder з класом animate-pulse під час завантаження даних. Реалізацію debounce-механізму наведено на рисунку 3.17.

```
useEffect(() => {
  const timer = setTimeout(async () => {
    setLoading(true);
    try {
      const res = await getPlants(search);
      setAllPlants(res.data);
    } catch {
      setAllPlants([]);
    } finally {
      setLoading(false);
    }
  }, search ? 300 : 0);
  return () => clearTimeout(timer);
}, [search]);
```

Рисунок 3.17 – Debounce-механізм пошуку у компоненті Catalogue.jsx

Сторінка деталей рослини реалізована у компоненті PlantDetail.jsx і забезпечує відображення повної медичної довідки, закриваючи функціональну вимогу ФВ-04. При монтуванні компонента або зміні параметра id з URL виконуються два паралельних запити через Promise.all() – getPlant(id) та getAdjacentPlants(id) – що скорочує загальний час завантаження сторінки. Стан photoOpen керує модальним переглядом фотографії у повному розмірі: компонент PhotoModal підписується на подію keydown документа через useEffect для закриття клавішею Escape з очищенням обробника при розмонтуванні. Кнопка переходу на карту формує URL з параметрами запиту plantId та plantName через navigate(), що дозволяє компоненту MapPage автоматично встановити фільтр при переході.

Інтерактивна карта реалізована у компоненті MapPage.jsx на основі бібліотек react-leaflet та react-leaflet-cluster і закриває функціональну вимогу ФВ-05. Карта ініціалізується з центром на координатах України та рівнем масштабування 6, використовуючи тайли OpenStreetMap як безкоштовне джерело картографічних даних [19, 20]. Іконки маркерів реалізовані через L.divIcon з вбудованим SVG-кодом піктограми у вигляді пін-форми зеленого кольору. Кластеризація маркерів реалізована через компонент MarkerClusterGroup – при великій кількості близько розташованих міток вони автоматично об'єднуються у числові кластери.

Компонент ClickHandler реалізований через хук useMapEvents з бібліотеки react-leaflet. Він перехоплює події click об'єкта карти і передає координати натискання у батьківський компонент через властивість onMapClick лише якщо властивість active має значення true. Компонент FlyToLocation отримує координати через властивість coords і викликає map.flyTo() з анімацією тривалістю 1.2 секунди при зміні значення coords через useEffect. Компонент DraggableMarker використовує useRef для отримання посилання на об'єкт маркера і через обробник події dragend зчитує нові координати через marker.getLatLng().

Визначення GPS-координат реалізоване у функції handleGPS() через navigator.geolocation.getCurrentPosition(). Успішний результат записується у стан selectedPosition і переводить модальне вікно до кроку форми. При відмові користувача у доступі до геолокації або відсутності підтримки API відображається відповідне повідомлення про помилку. Реалізацію функції handleGPS() наведено на рисунку 3.18.

```
function handleGPS() {
  if (!navigator.geolocation) {
    alert('Геолокація не підтримується вашим браузером');
    return;
  }
  navigator.geolocation.getCurrentPosition(
    (pos) => {
      setSelectedPosition({ lat: pos.coords.latitude, lng: pos.coords.longitude });
      setModalStep('form');
      setShowModal(true);
    },
    () => {
      alert('Не вдалось визначити геопозицію. Спробуйте вибрати місце на карті.');
      setModalStep('choice');
    }
  );
}
```

Рисунок 3.18 – Функція handleGPS() (MapPage.jsx)

Пошук місцевості реалізований через Nominatim API від OpenStreetMap з debounce 300 мс. Запит надсилається лише якщо довжина рядка пошуку перевищує два символи. Результати фільтруються за класами place та boundary і сортуються за пріоритетом типу населеного пункту: city, town, village, hamlet. Список підказок приховується при кліку за межами компонента через обробник mousedown на документі [19].

Сторінка профілю реалізована у компоненті Profile.jsx і закриває функціональну вимогу ФВ-07. Дані користувача разом з повною історією пошуків завантажуються єдиним запитом до GET /api/profile у хуку useEffect. Ініціали користувача для аватара формуються через profile.username.slice(0, 2).toUpperCase(). Дата реєстрації форматується через toLocaleDateString() з локаллю uk-UA. Функція getConfidenceInfo() повторно використовується з компонента UploadPhoto, що забезпечує консистентну логіку відображення рівня впевненості без дублювання коду.

Сторінки автентифікації реалізовані у компонентах Login.jsx та Register.jsx. Стан форми в обох компонентах керується через єдиний об'єкт у useState, що оновлюється через spread-оператор у handleChange(). У Register.jsx клієнтська валідація виконується до надсилання запиту на сервер: перевіряється заповненість полів, відповідність формату email та збіг паролів. Компонент PasswordInput виокремлено для повторного використання у полях пароля та

підтвердження – він керує локальним станом видимості пароля через useState. При успішному вході JWT-токен та ім'я користувача зберігаються у localStorage, після чого відбувається перехід на головну сторінку через navigate().

3.5.3 Дизайн-система та адаптивний інтерфейс

Візуальний стиль застосунку базується на мінімалістичній зелено-білій палітрі, що семантично асоціюється з природою та лікарськими рослинами. Основний акцентний колір green-700 (#15803D) використовується для кнопок, активних посилань, бейджів та фонів заголовкових секцій. Колір amber застосовується виключно для секцій протипоказань як семантично вмотивований сигнал небезпеки, що відрізняє їх від інших розділів медичної довідки.

Типографіка реалізована через два шрифти Google Fonts. Playfair Display використовується для заголовків і назв рослин через клас font-heading – він надає застосунку академічний ботанічний характер. Inter використовується для основного тексту і забезпечує оптимальну читабельність на різних розмірах екрана. Система заокруглень базується на двох значеннях: rounded-3xl для карток і контейнерів та rounded-full для кнопок, бейджів і аватарів.

Мікроанімації реалізовані через утилітарні класи Tailwind CSS: картки рослин застосовують hover:-translate-y-1 та hover:shadow-lg з transition-all duration-200, що забезпечує тактильний відгук при наведенні. Кнопки використовують active:scale-95 для імітації натискання.

Адаптивна верстка забезпечує коректне відображення на екранах від 375px відповідно до нефункціональної вимоги зручності з підрозділу 1.5.2. Каталог рослин перемикається між одно-, дво- та трьохколонковою сіткою на брейкпоінтах sm (768px) та lg (1024px) відповідно. Навігаційна панель

перемикається між горизонтальним режимом та гамбургер-меню на брейкпоінті md (768px).

3.6 Забезпечення безпеки застосунку

Безпека застосунку реалізована на кількох незалежних рівнях відповідно до нефункціональних вимог підрозділу 1.5.2.

На рівні зберігання облікових даних паролі користувачів хешуються алгоритмом bcrypt з cost factor 10 перед збереженням у базі даних. Значення cost factor визначає обчислювальну складність алгоритму і унеможливлює відновлення оригінального пароля методом перебору навіть при повній компрометації бази даних [44].

На рівні автентифікації JWT-токен підписується алгоритмом HMAC-SHA256 з секретом зі змінної середовища JWT_SECRET і має термін дії 7 днів. Підпис верифікується middleware auth.js при кожному захищеному запиті без звернення до бази даних, що відповідає stateless-принципу REST-архітектури. У разі відсутності або невалідності токена повертається HTTP 401 до виконання будь-якої бізнес-логіки.

На рівні захисту API-ключів усі ключі зовнішніх сервісів – Pl@ntNet, Gemini та Cloudinary – зберігаються виключно у файлі server/.env, який включено до .gitignore і ніколи не потрапляє до системи контролю версій. Клієнтський код не містить жодних ключів – усі звернення до зовнішніх API виконуються виключно з серверної частини, що унеможливлює їх витік через інструменти розробника браузера.

На рівні обробки вхідних даних розмір завантажуваних файлів обмежено 5 МБ засобами Multer, що запобігає атакам відмови в обслуговуванні через передачу надмірно великих файлів. Видалення геопросторових міток захищено перевіркою відповідності ідентифікатора автора мітки та ідентифікатора

поточного користувача з JWT-токена на серверному рівні – це унеможливлює видалення чужих записів навіть за наявності валідного токена.

ВИСНОВКИ ДО РОЗДІЛУ

У третьому розділі описано практичну реалізацію вебзастосунку «Інтерактивний довідник лікарських рослин на основі застосування штучного інтелекту», що охоплює трирівневу архітектуру системи, схему бази даних з моделями Prisma, серверну та клієнтську частини, підсистему ідентифікації рослин із залученням зовнішніх API та механізми забезпечення безпеки.

Трирівнева клієнт-серверна архітектура реалізована з чіткою ізоляцією зовнішніх сервісів: API-ключі Pl@ntNet, Gemini та Cloudinary зберігаються виключно у серверному середовищі і жодного разу не передаються клієнтській частині. Структура директорій організована за принципом монорепозиторію з незалежними артефактами client/ та server/.

База даних PostgreSQL з п'яти таблиць реалізує нетривіальне архітектурне рішення: таблиця Plant одночасно виконує роль каталогу лікарських рослин і кешу результатів AI-аналізу, тоді як таблиця SearchHistory фіксує персональний журнал дій кожного користувача. Таблиця PlantProperty реалізована за патерном EAV, що забезпечує розширюваність без структурних змін схеми. Управління схемою здійснюється через систему міграцій Prisma.

Серверна частина організована у п'ять модульних маршрутизаторів з десятьма REST API маршрутами. Підсистема автентифікації реалізує хешування паролів алгоритмом bcrypt з cost factor 10 та JWT-автентифікацію з верифікацією без звернення до бази даних. Централізований middleware auth.js забезпечує захист маршрутів /api/search/identify, /api/map та /api/profile.

Ключовим алгоритмом системи є послідовність обробки запиту ідентифікації: Multer → Cloudinary → Pl@ntNet API → cache-aside → Gemini API → SearchHistory. Патерн cache-aside гарантує що кожен вид рослини аналізується Gemini API рівно один раз незалежно від кількості звернень.

Логіка пріоритету назви забезпечує використання верифікованих ботанічних назв від Pl@ntNet як пріоритетного джерела.

Клієнтська частина на React реалізує адаптивний інтерфейс для мобільних та десктоп-пристроїв. Централізований сервіс api.js з Axios interceptor забезпечує автоматичне додавання JWT-заголовка до всіх запитів без дублювання логіки авторизації у компонентах. Реалізовані сторінки охоплюють повний функціонал застосунку: ідентифікацію рослин, перегляд каталогу, інтерактивну карту, профіль користувача та автентифікацію. Безпека реалізована на рівнях зберігання паролів, автентифікації, захисту API-ключів та авторизації операцій з ресурсами.

РОЗДІЛ 4. ТЕСТУВАННЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ РОБОТИ СИСТЕМИ

У даному розділі описано тестування розробленого вебзастосунку та проаналізовано результати його роботи. Перевірено функціонал основних сторінок, адаптивність інтерфейсу на мобільних пристроях та відповідність реалізації визначеним функціональним вимогам.

4.1 Тестування функціоналу застосунку

Тестування функціоналу застосунку проводилось у локальному середовищі розробки: серверна частина запущена на порті 5000, клієнтська – на порті 3000. Перевірка здійснювалась у браузері Google Chrome для всіх основних сторінок застосунку: автентифікації, ідентифікації рослин, каталогу, інтерактивної карти та профілю користувача. Для кожного сценарію перевірялись як коректні вхідні дані, так і граничні випадки.

4.1.1 Автентифікація та реєстрація

Сторінка входу є вхідною точкою до застосунку для неавторизованих користувачів. Вона містить форму з полями для введення email та пароля, кнопку входу, посилання на сторінку реєстрації та посилання на головну сторінку. Після успішної автентифікації користувач перенаправляється на головну сторінку застосунку. Загальний вигляд сторінки входу наведено на рисунку 4.1.

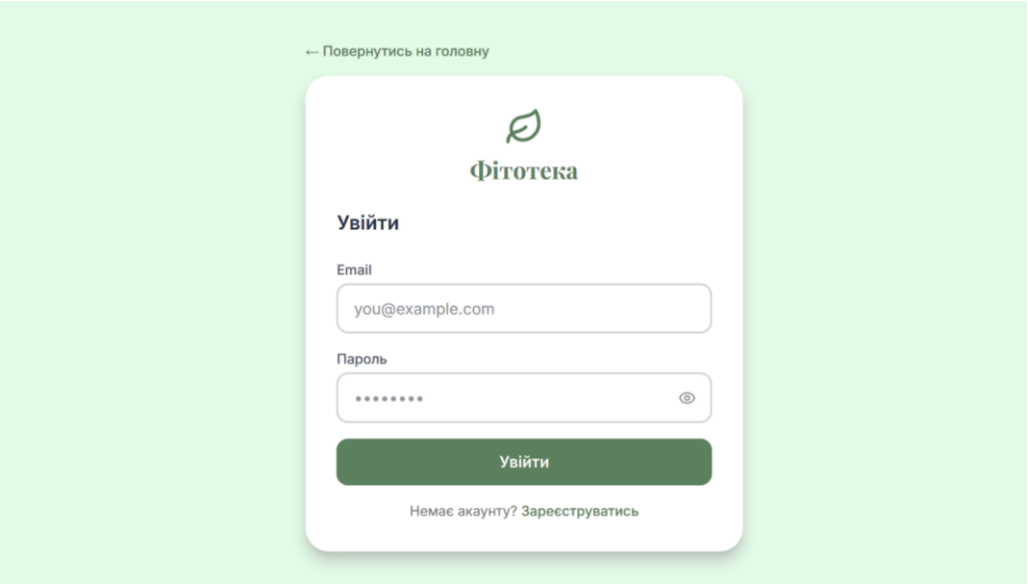


Рисунок 4.1 – Сторінка входу до системи

У разі введення невірного email або пароля система повертає відповідне повідомлення про помилку. При цьому обидва варіанти помилки – неіснуючий користувач та невірний пароль – відображають однакове повідомлення, що унеможливлює визначення наявності конкретного email у базі даних. Результат спроби входу з невірними обліковими даними наведено на рисунку 4.2.

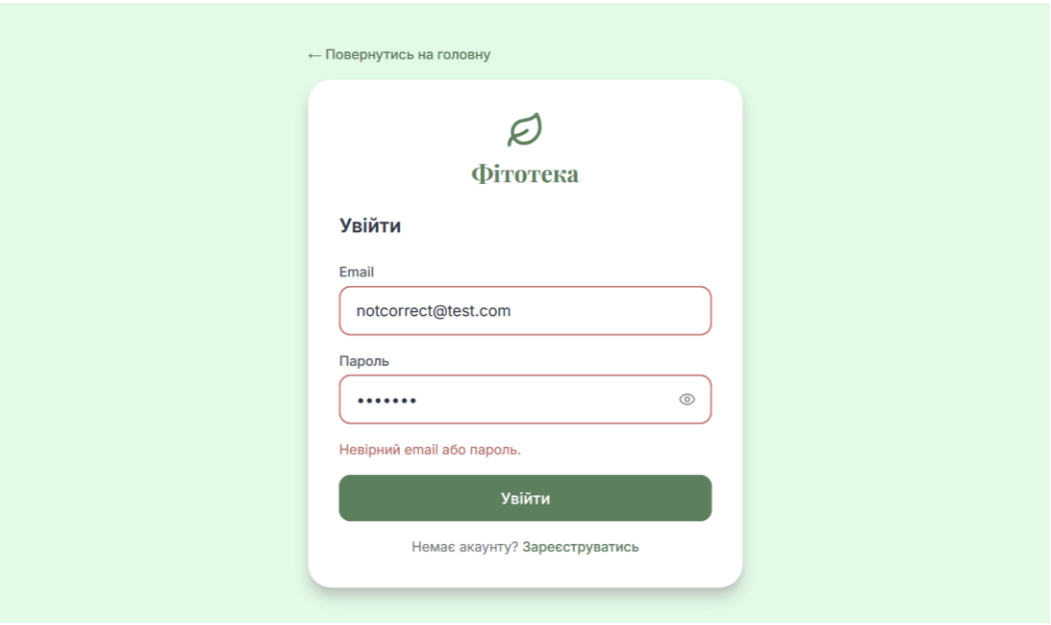


Рисунок 4.2 – Повідомлення про помилку при невірних облікових даних

Сторінка реєстрації містить форму з полями для введення імені користувача, email, пароля та підтвердження пароля. Після успішної реєстрації

користувач перенаправляється на сторінку входу. Загальний вигляд сторінки реєстрації наведено на рисунку 4.3.

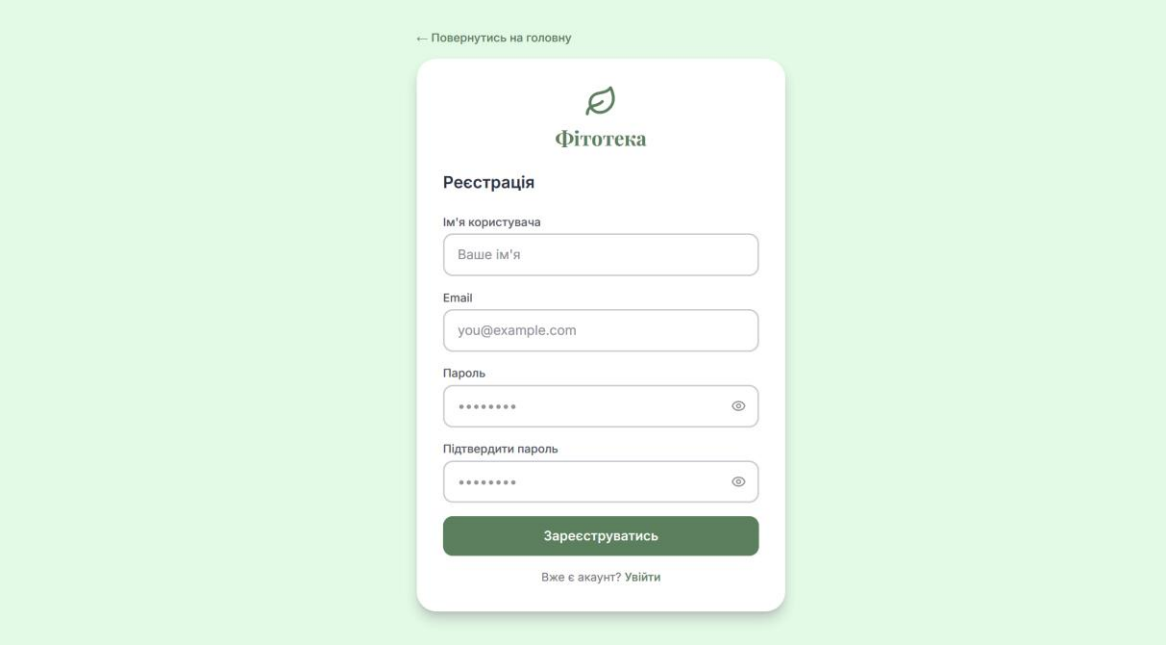


Рисунок 4.3 – Сторінка реєстрації нового користувача

При введенні різних значень у поля пароля та підтвердження пароля клієнтська валідація спрацьовує до надсилання запиту на сервер і відображає відповідне повідомлення про помилку. Результат валідації при незбігу паролів наведено на рисунку 4.4.

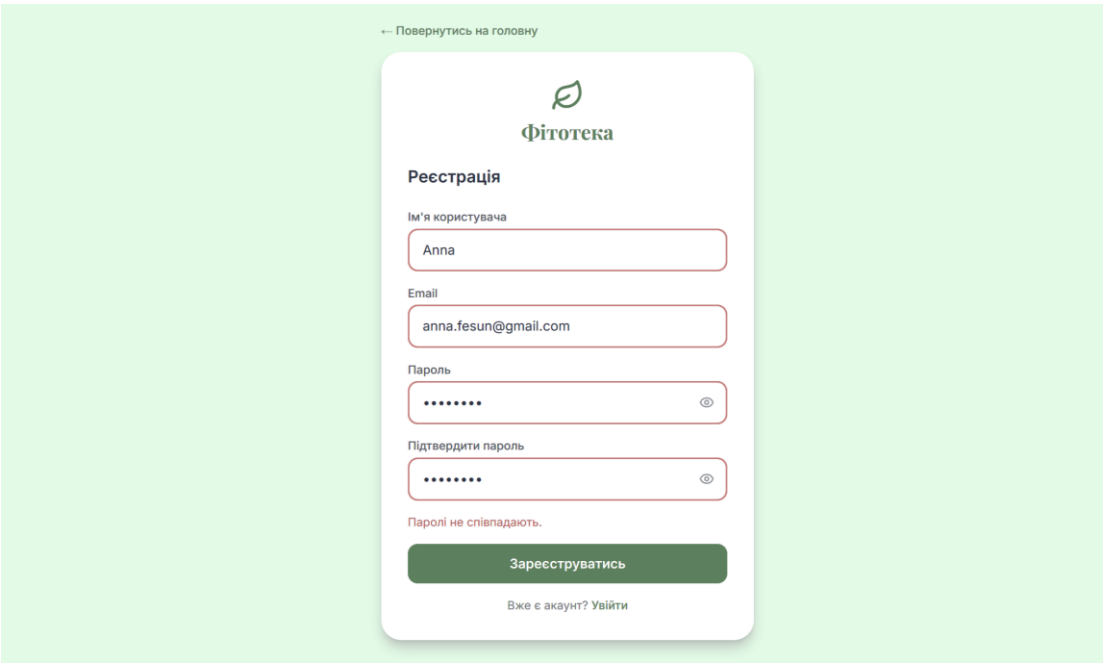


Рисунок 4.4 – Валідаційна помилка при незбігу паролів

4.1.2 Головна сторінка

Головна сторінка містить hero-секцію з назвою застосунку та двома кнопками навігації, секцію ідентифікації рослини та блок з трьома картками переваг застосунку. Для неавторизованих користувачів функція завантаження фотографії недоступна – замість компонента ідентифікації відображається пропозиція увійти до системи. Загальний вигляд головної сторінки для авторизованого користувача наведено на рисунку 4.5.

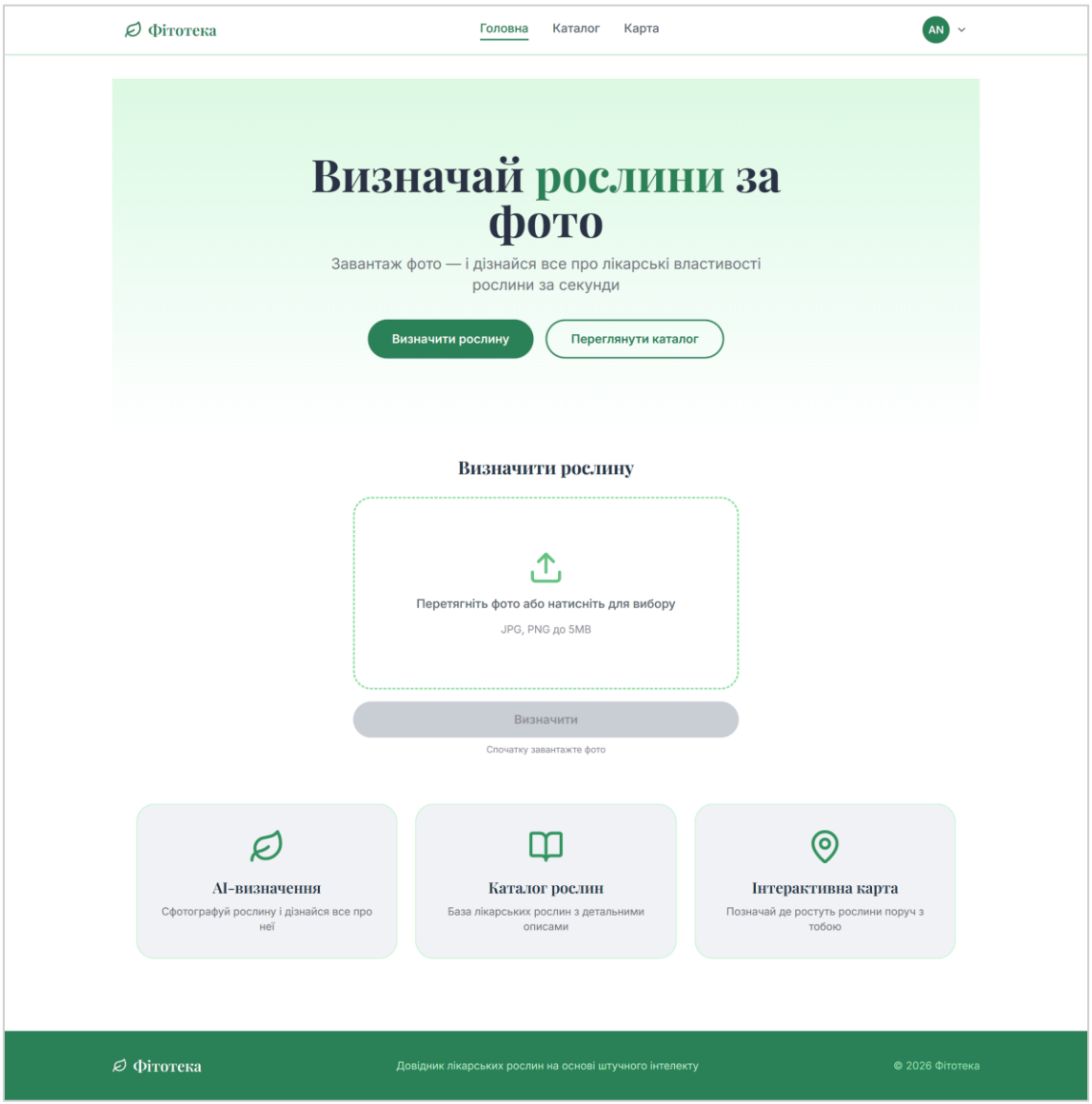


Рисунок 4.5 – Головна сторінка застосунку після входу до системи

4.1.3 Ідентифікація рослини за фотографією

Компонент ідентифікації рослини розміщений на головній сторінці застосунку. У початковому стані відображається зона завантаження з підказкою щодо способів вибору файлу. Початковий стан компонента наведено на рисунку 4.6.

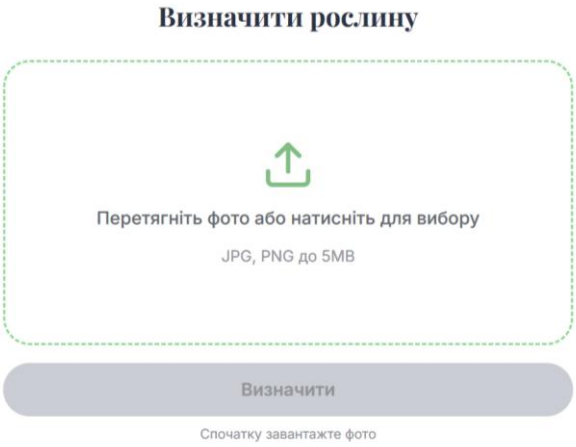


Рисунок 4.6 – Компонент ідентифікації у початковому стані

Після вибору файлу через клік або перетягування у зоні завантаження відображається попередній перегляд обраного зображення разом з його назвою та розміром. За потреби користувач може видалити завантажене зображення і обрати інше. Стан компонента після завантаження фотографії наведено на рисунку 4.7.

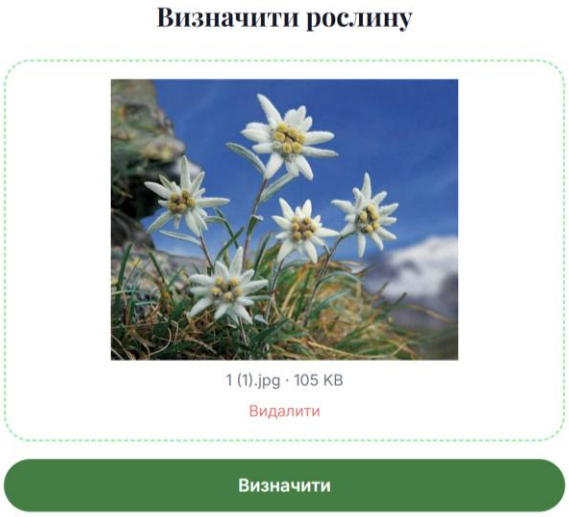


Рисунок 4.7 – Попередній перегляд завантаженого зображення

При отриманні результату ідентифікації з показником впевненості 70% і вище відображається зелений індикатор з повідомленням про точне визначення виду. Поряд відображаються українська та латинська назви рослини та бейдж лікарськості. Результат ідентифікації з високим показником впевненості наведено на рисунку 4.8.

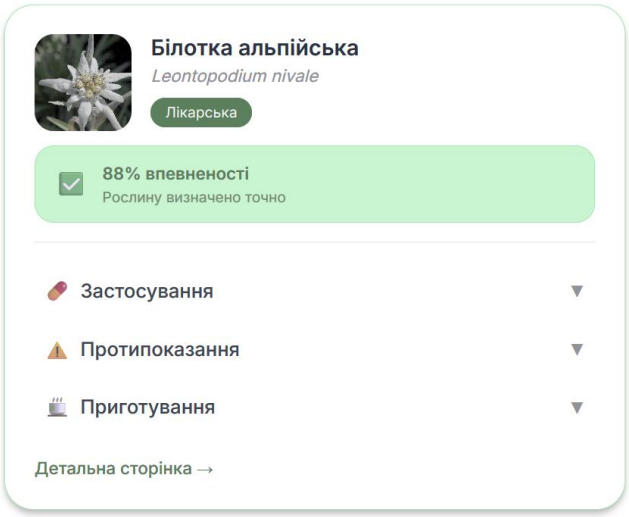


Рисунок 4.8 – Результат ідентифікації з високим показником впевненості

При отриманні результату з показником впевненості нижче 0.3 система відображає червоний індикатор з попередженням про низьку достовірність результату. Користувачу рекомендується завантажити чіткіше фото або звірити результат з каталогом. Результат ідентифікації з низьким показником впевненості наведено на рисунку 4.9.

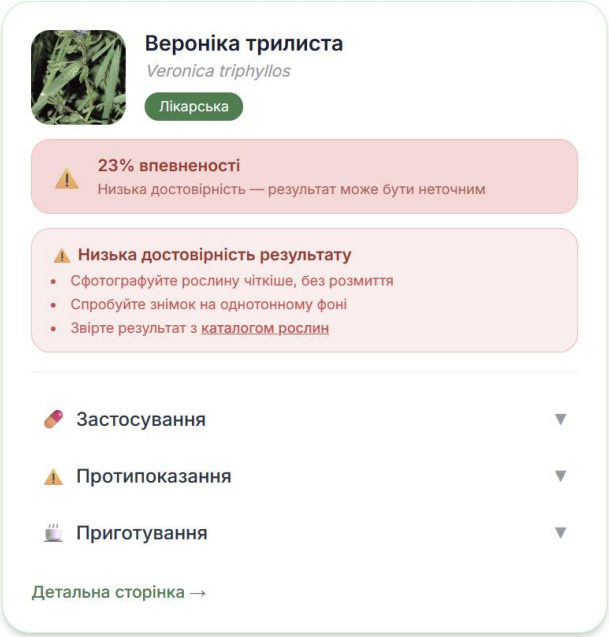


Рисунок 4.9 – Результат ідентифікації з низьким показником впевненості та попередженням

Для лікарських рослин під індикатором впевненості відображається медична довідка з розділами лікарського застосування, протипоказань та приготування, кожен з яких розгортається окремо при натисканні. Розгорнуту медичну довідку наведено на рисунку 4.10.

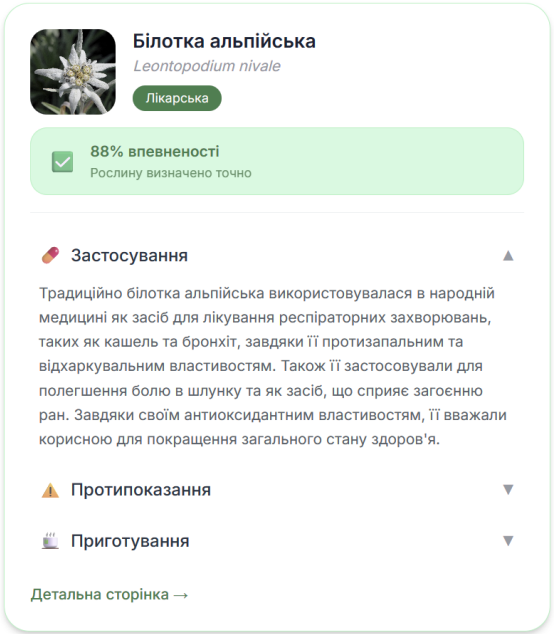


Рисунок 4.10 – Розгорнута медична довідка

У разі якщо ідентифікована рослина не є лікарською, поле isMedicinal у відповіді сервера має значення false і секції медичної довідки не відображаються. Результат ідентифікації нелікарської рослини наведено на рисунку 4.11.

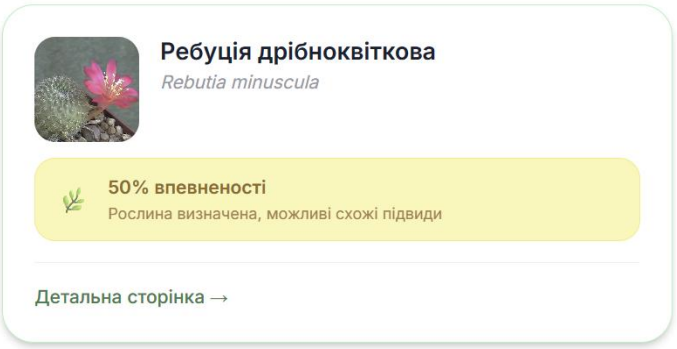


Рисунок 4.11 – Результат ідентифікації нелікарської рослини

4.1.4 Перегляд каталогу рослин

Сторінка каталогу відображає повний перелік рослин у базі даних у вигляді адаптивної сітки карток. Кожна картка містить фотографію рослини, її українську та латинську назви, короткий опис та бейдж лікарськості. У верхній частині сторінки розміщено пошуковий рядок та фільтр-чіпи для вибору категорії відображення: усі рослини, лікарські або нелікарські. При введенні тексту у пошуковий рядок список рослин оновлюється і відображає лише рослини, назва яких відповідає введеному запиту – пошук виконується одночасно по українській та латинській назвах без урахування регістру. При активації фільтру «Лікарські» відображаються виключно лікарські рослини. У нижній частині сторінки розміщено пагінацію для навігації між сторінками каталогу. Загальний вигляд сторінки каталогу наведено на рисунку 4.12.

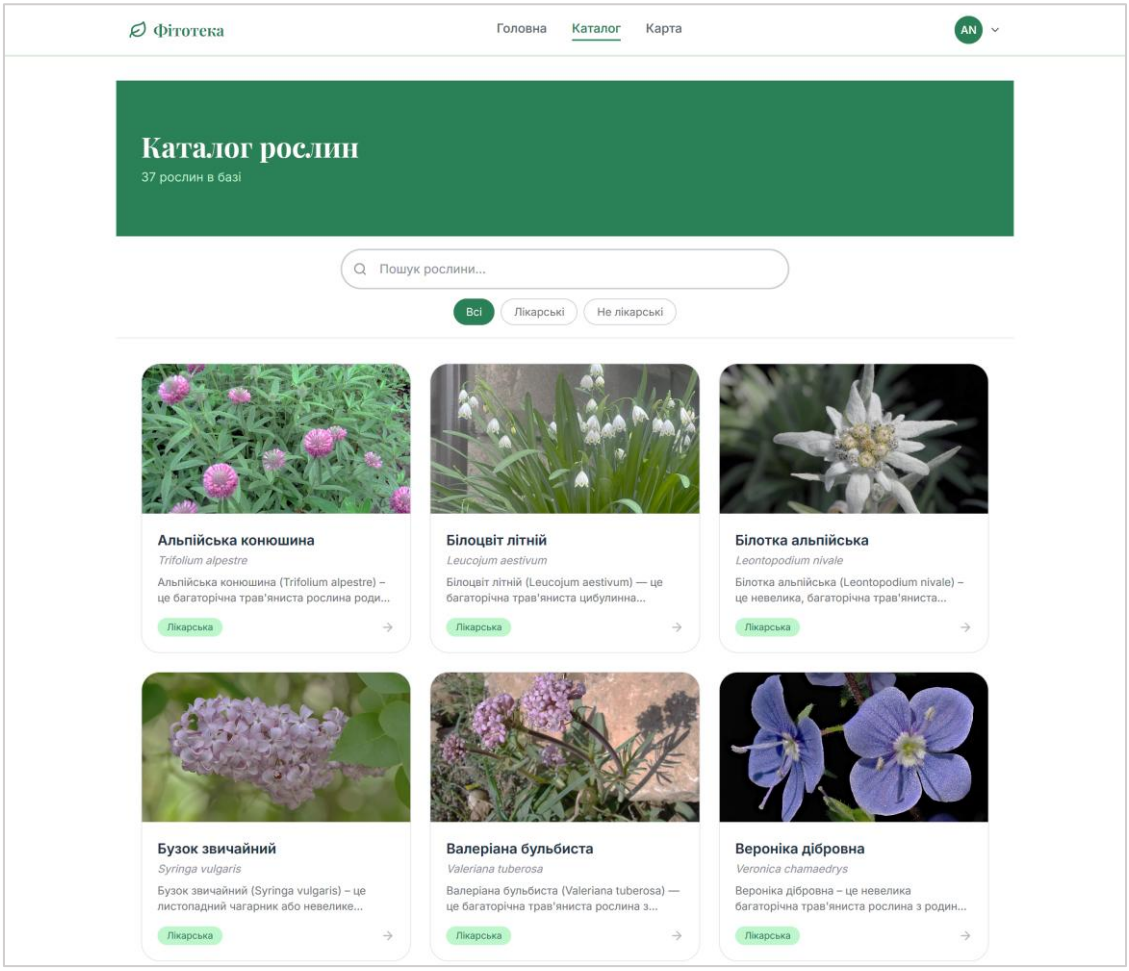


Рисунок 4.12 – Загальний вигляд сторінки каталогу рослин

Сторінка деталей рослини відкривається при натисканні на картку рослини у каталозі. Вона відображає повну медичну довідку: фотографію рослини, загальний опис, лікарські властивості, протипоказання та способи приготування. При натисканні на фотографію відкривається модальне вікно з повним переглядом зображення. Секція протипоказань виділена жовтим фоном для наочного попередження користувача. У нижній частині сторінки розміщено кнопку переходу до інтерактивної карти для перегляду місць знаходження рослини, а також навігаційні кнопки для переходу між сусідніми рослинами в алфавітному порядку. Сторінку деталей рослини наведено на рисунку 4.13.

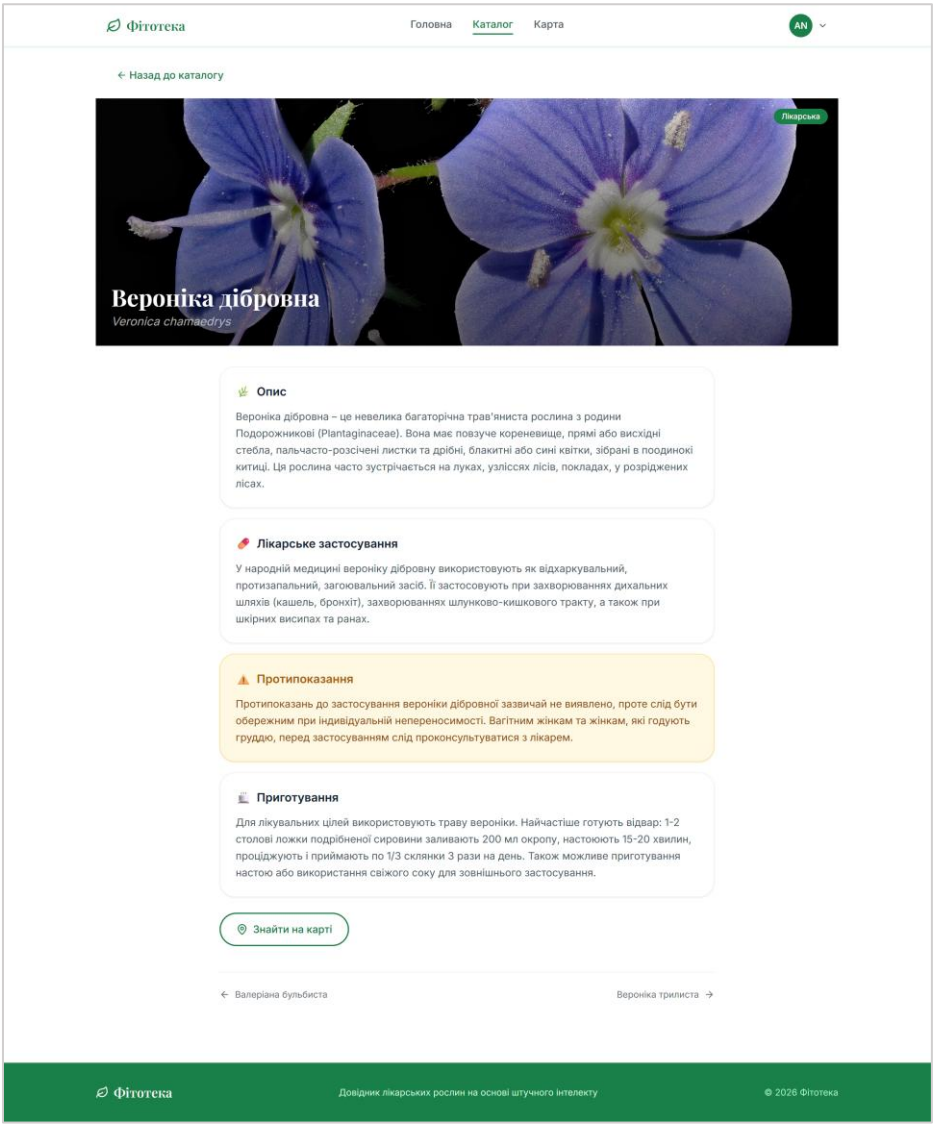


Рисунок 4.13 – Сторінка деталей рослини з медичною довідкою

4.1.5 Інтерактивна карта знаходжень

Сторінка інтерактивної карти відображає геопросторові мітки рослин, додані користувачами спільноти. У верхній частині сторінки розміщено пошуковий рядок для пошуку місцевості, перемикач між усіма та власними мітками, а також фільтр за видом рослини. При великій кількості близько розташованих міток вони автоматично об'єднуються у числові кластери для зручності перегляду. При збільшенні масштабу або натисканні на кластер він

розкривається і відображає окремі маркери рослин. Карту з кластеризованими мітками наведено на рисунку 4.14.

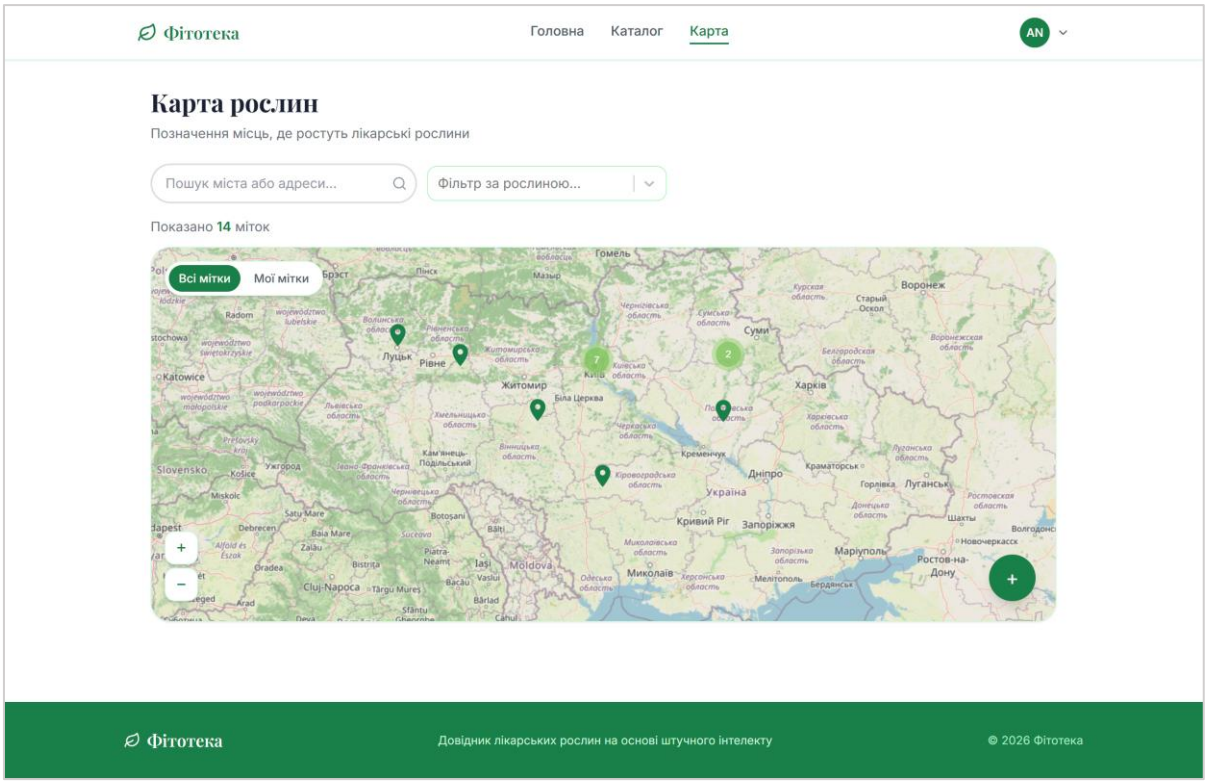


Рисунок 4.14 – Інтерактивна карта з кластеризованими мітками рослин

Клік на окремому маркері відкриває спливаюче вікно з фотографією рослини, її українською та латинською назвою, нотаткою автора та іменем користувача що додав мітку. Вигляд спливаючого вікна наведено на рисунку 4.15.

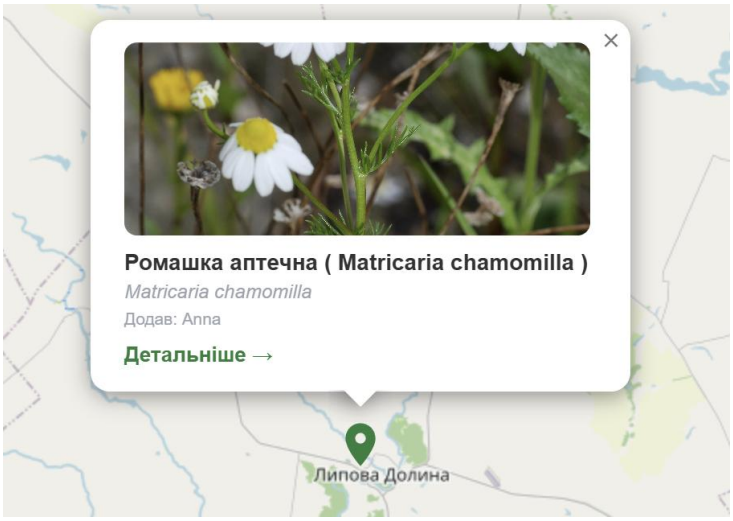


Рисунок 4.15 – Спливаюче вікно з інформацією про рослину

Для додавання нової мітки користувач натискає кнопку з іконкою плюса, після чого відкривається модальне вікно з двома варіантами визначення місцезнаходження: вибір точки на карті або автоматичне визначення через GPS. Вигляд модального вікна наведено на рисунку 4.16.

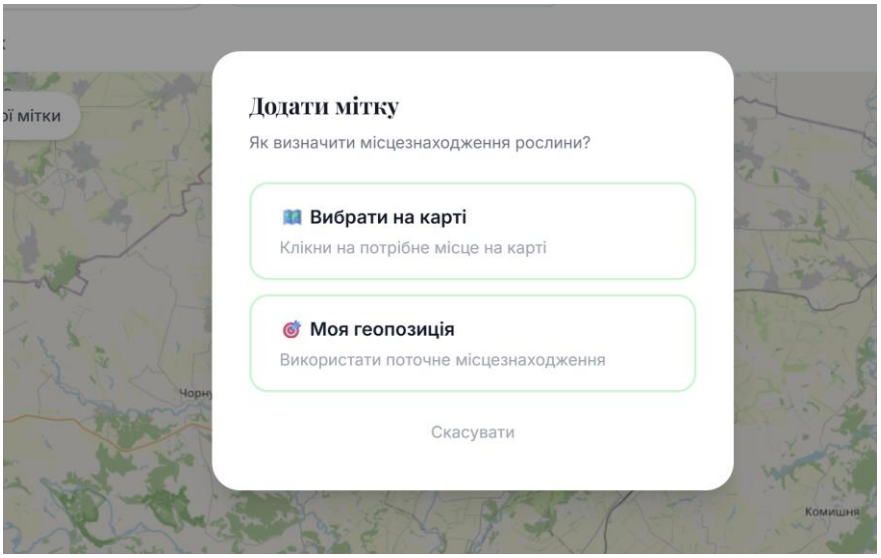


Рисунок 4.16 – Вибір способу додавання мітки

При виборі варіанту «Вибрати на карті» карта переходить у режим вибору координат – користувач натискає на потрібну точку і на обраній позиції з'являється перетягуваний маркер, який можна перемістити для точнішого визначення місця. При виборі варіанту «Моя геопозиція» система запитує дозвіл на доступ до геолокації браузера і автоматично визначає поточні координати користувача, після чого карта переміщується до визначеного місця. Після визначення координат відображається форма для заповнення інформації про мітку: координати обраної точки, обов'язкове поле вибору рослини зі списку всіх рослин каталогу та необов'язкове поле для текстової нотатки. Після підтвердження мітка зберігається і відображається на карті. Вигляд форми додавання мітки наведено на рисунку 4.17.

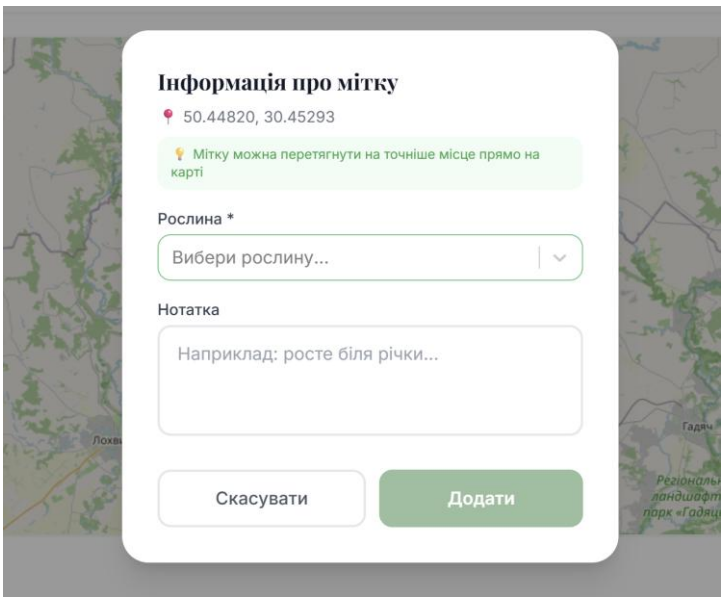


Рисунок 4.17 – Форма заповнення інформації про геопросторову мітку

4.1.6 Особистий профіль користувача

Сторінка профілю відображає інформацію про користувача: аватар з ініціалами, ім'я, email та дату реєстрації. У нижній частині сторінки розміщено хронологічну історію пошуків користувача. При відсутності попередніх пошуків відображається порожній стан з підказкою та кнопкою переходу до функції ідентифікації рослини.

Після виконання пошуків кожен запис в історії відображається у вигляді клікабельної картки з фотографією завантаженого користувачем зображення, українською та латинською назвою ідентифікованої рослини, датою та часом пошуку, а також кольоровим індикатором впевненості класифікатора. При натисканні на запис відбувається перехід до сторінки деталей відповідної рослини. Сторінку профілю з історією пошуків наведено на рисунку 4.18.

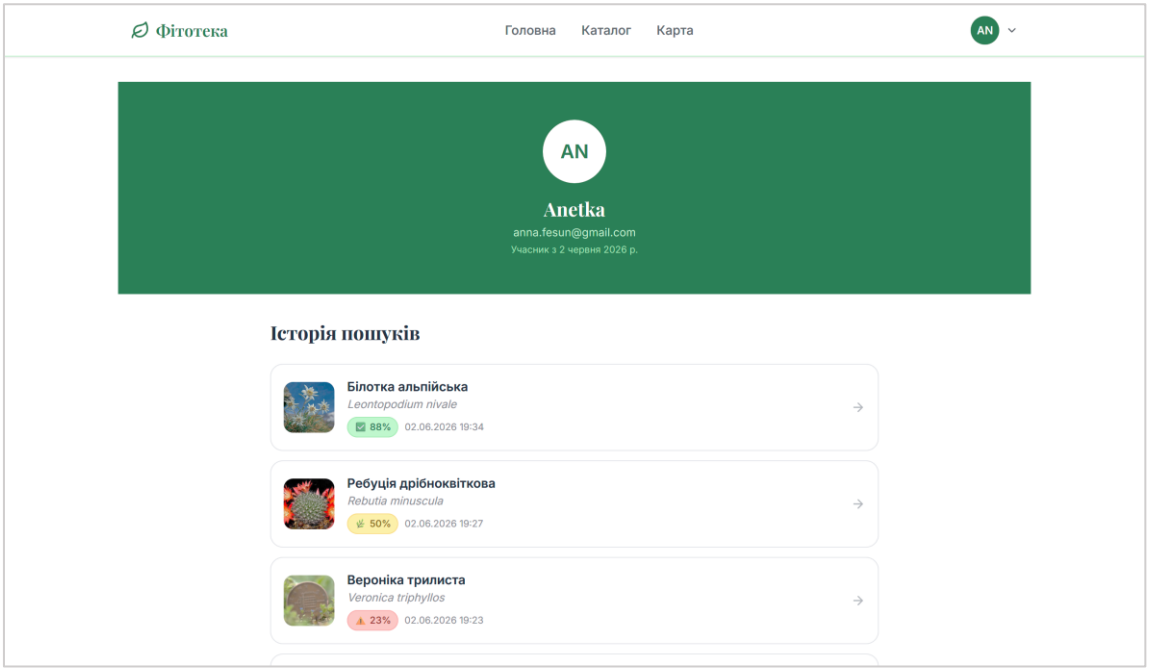


Рисунок 4.18 – Сторінка профілю з хронологічною історією пошуків

4.2 Тестування адаптивності інтерфейсу

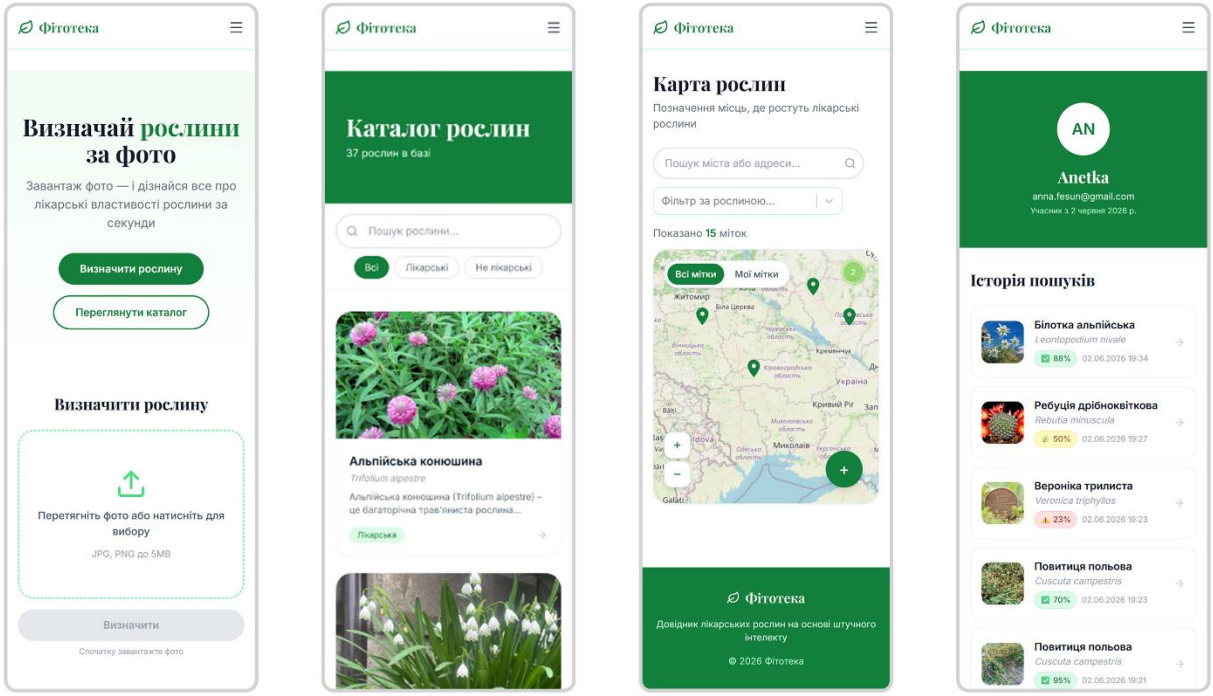
Тестування адаптивності проводилось в інструментах розробника браузера Google Chrome в режимі емуляції мобільного пристрою з шириною екрана 375px. Перевірялась коректність відображення елементів інтерфейсу, збереження функціональності та читабельність контенту на вузьких екранах.

Головна сторінка на мобільному пристрої відображає hero-секцію з адаптованим розміром заголовку та компонент ідентифікації у повноширинному режимі. Кнопки навігації перебудовуються у вертикальний порядок для зручності натискання (рис. 4.19 а).

Сторінка каталогу перемикається з трьохколонкової сітки на одноколонкову, що забезпечує зручний перегляд карток рослин. Пошуковий рядок та фільтр-чіпи зберігають коректне відображення і залишаються повністю функціональними (рис. 4.19 б).

Інтерактивна карта відображається у повноширинному режимі зі збереженням повної функціональності: маркери, кластеризація та елементи керування коректно відображаються і залишаються доступними (рис. 4.19 в).

Сторінка профілю відображає інформацію про користувача та картки історії пошуків у одноколонковому режимі зі збереженням усіх елементів кожного запису (рис. 4.19 г).



а) б) в) г)

Рисунок 4.19 – Відображення основних сторінок застосунку на мобільному пристрої (375px): а) головна сторінка; б) каталог; в) карта; г) профіль

Навігаційна панель на мобільному пристрої замінює горизонтальне меню на гамбургер-кнопку. При натисканні розгортається вертикальне меню з усіма посиланнями та кнопкою автентифікації. Мобільне меню навігаційної панелі наведено на рисунку 4.20.

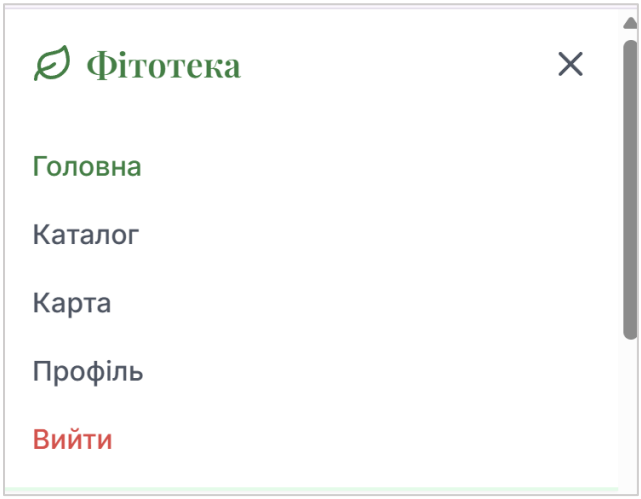
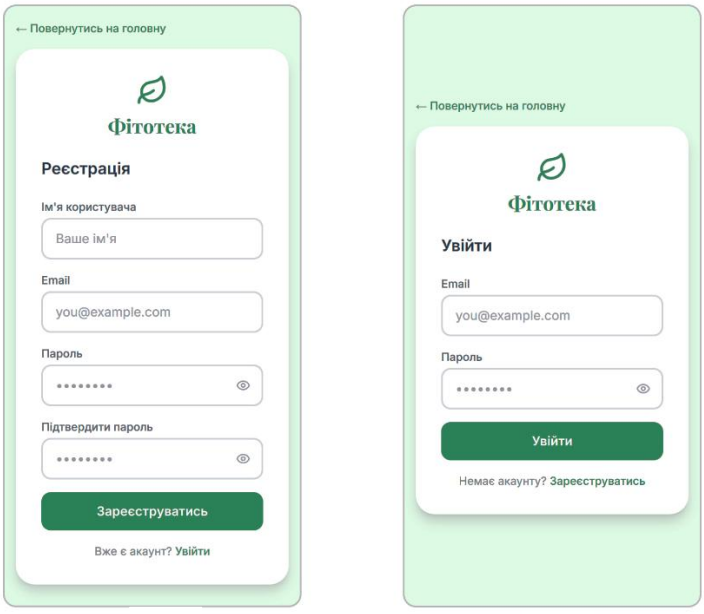


Рисунок 4.20 – Мобільне меню навігаційної панелі

Сторінки автентифікації на мобільному пристрої відображають форми у повноширинному режимі з адаптованими відступами. Сторінка входу зберігає коректне відображення полів email та пароля, кнопки входу та посилань на реєстрацію і головну сторінку (рис. 4.21 а). Сторінка реєстрації відображає всі чотири поля форми у вертикальному порядку зі збереженням повної функціональності валідації (рис. 4.21 б). Відображення сторінок автентифікації на мобільному пристрої наведено на рисунку 4.21.



а) б)

Рисунок 4.21 – Сторінки автентифікації на мобільному пристрої (375px): а) вхід; б) реєстрація

4.3 Аналіз результатів тестування

За результатами проведеного тестування складено зведену таблицю відповідності реалізованого застосунку функціональним вимогам, сформульованим у підрозділі 1.5.1. Результати тестування наведено у таблиці 4.1.

Таблиця 4.1 – Результати тестування функціональних вимог

Вимога	**Опис**	**Результат**
ФВ-01	Визначення виду рослини за фотографією з відображенням назви та показника впевненості	Виконано
ФВ-02	Автоматична генерація структурованої медичної довідки	Виконано
ФВ-03	Кешування результатів AI-аналізу за латинською назвою виду	Виконано
ФВ-04	Каталог рослин з текстовим пошуком та фільтрацією	Виконано
ФВ-05	Інтерактивна карта зі спільнотними мітками та можливістю їх додавання	Виконано
ФВ-06	Реєстрація та автентифікація користувачів на основі JWT	Виконано
ФВ-07	Збереження та перегляд особистої історії пошукових запитів	Виконано

Результати тестування підтверджують повну відповідність реалізованого застосунку усім семи функціональним вимогам, визначеним у розділі 1. Усі основні сценарії взаємодії користувача з системою функціонують коректно: ідентифікація рослин за фотографією з відображенням показника впевненості, генерація та відображення структурованих медичних довідок, текстовий пошук і фільтрація у каталозі, додавання геопросторових міток через клік на карті або GPS, а також збереження і перегляд персональної історії пошуків. Тестування адаптивності підтвердило коректне відображення інтерфейсу на екранах шириною 375px для всіх основних сторінок застосунку.

ВИСНОВКИ ДО РОЗДІЛУ

У четвертому розділі проведено тестування розробленого вебзастосунку у локальному середовищі розробки методом ручного тестування у браузері Google Chrome. Перевірено всі основні сторінки та сценарії взаємодії користувача з системою.

Тестування функціоналу підтвердило коректну роботу підсистем автентифікації, ідентифікації рослин, каталогу, інтерактивної карти та профілю користувача. Позитивні сценарії виконуються відповідно до визначених функціональних вимог. Граничні випадки – введення невірних облікових даних, низька впевненість класифікатора, ідентифікація нелікарської рослини – обробляються коректно з відображенням відповідних повідомлень користувачу.

Тестування адаптивності підтвердило коректне відображення усіх сторінок застосунку на екранах шириною 375px. Елементи інтерфейсу адаптуються до вузьких екранів: сітка каталогу перемикається на одноколонкову, навігаційна панель замінюється мобільним меню, форми автентифікації відображаються у повноширинному режимі.

Аналіз результатів тестування підтвердив повну відповідність розробленого застосунку усім семи функціональним вимогам, визначеним у підрозділі 1.5.1.

ВИСНОВКИ

У ході виконання дипломного проєкту розроблено вебзастосунок «Інтерактивний довідник лікарських рослин на основі застосування штучного інтелекту», що забезпечує автоматизоване визначення лікарських рослин за фотографією та надання структурованої медичної інформації українською мовою.

У першому розділі проведено аналіз предметної області та виконано порівняльний огляд існуючих рішень для визначення рослин. Встановлено що наявні сервіси орієнтовані переважно на ботанічну класифікацію і не забезпечують комплексного поєднання автоматизованої ідентифікації, структурованих медичних довідок та повноцінного україномовного контенту. Виявлені недоліки існуючих рішень стали основою для формулювання функціональних та нефункціональних вимог до розроблюваної системи.

У другому розділі проведено аналіз існуючих архітектурних підходів та обґрунтовано вибір технологічного стеку для реалізації повнофункціональної клієнт-серверної системи. Визначено засоби інтеграції зовнішніх сервісів, зокрема Pl@ntNet API для ботанічної ідентифікації рослин та API великої мовної моделі для генерації медичних довідок. Обраний стек забезпечує необхідну продуктивність, масштабованість та безкоштовний рівень доступу до зовнішніх сервісів.

У третьому розділі описано практичну реалізацію системи. Спроєктовано реляційну базу даних, що поєднує функції каталогу лікарських рослин та кешу результатів AI-аналізу. Реалізовано серверну частину з модульною структурою маршрутизаторів та підсистемою автентифікації. Розроблено ключовий алгоритм ідентифікації рослин, що реалізує послідовне звернення до зовнішніх API із застосуванням патерну cache-aside – кожен вид рослини аналізується рівно один раз незалежно від кількості звернень до системи, що суттєво

скорочує витрати на зовнішні сервіси. Клієнтська частина реалізує повний набір сторінок з адаптивним інтерфейсом для роботи на різних пристроях.

У четвертому розділі проведено тестування розробленого застосунку методом ручного тестування у браузері. Перевірено функціонал усіх основних сторінок та сценарії взаємодії користувача з системою, включаючи граничні випадки – введення невірних облікових даних, низька впевненість класифікатора та ідентифікація нелікарської рослини. Підтверджено коректну адаптивність інтерфейсу на мобільних пристроях та повну відповідність реалізації всім визначеним функціональним вимогам.

Таким чином, у результаті виконання дипломного проєкту реалізовано всі технічні вимоги, визначені у технічному завданні: адаптивний інтерфейс з підтримкою мобільних пристроїв, автентифікація та реєстрація користувачів, визначення виду рослини за фотографією з відображенням ступеня впевненості результату, автоматична генерація структурованих медичних довідок, каталог рослин з пошуком та фільтрацією, інтерактивна карта знаходжень рослин з можливістю додавання власних міток та персональна історія пошукових запитів. Розроблений застосунок може бути використаний як україномовний сервіс для ідентифікації лікарських рослин та отримання медичної інформації про них, а також має потенціал для подальшого розвитку та вдосконалення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. World Health Organization. Traditional medicine has a long history of contributing to conventional medicine and continues to hold promise. URL: https://www.who.int/news-room/feature-stories/detail/traditional-medicine-has-a-long-history-of-contributing-to-conventional-medicine-and-continues-to-hold-promise
2. Davis C. C. et al. Medicinal plants meet modern biodiversity science // Current Biology. 2024. Vol. 34. Issue 1.
3. Лікарські рослини України // Енциклопедія Сучасної України : електронна версія / НАН України, НТШ. URL: https://esu.com.ua/article-55467
4. Wäldchen J., Mäder P. Plant Species Identification Using Computer Vision Techniques: A Systematic Literature Review // Archives of Computational Methods in Engineering. 2018. Vol. 25. P. 507–543.
5. Goodfellow I., Bengio Y., Courville A. Deep Learning. Cambridge : MIT Press, 2016.
6. Державна фармакопея України. Харків : ДП «Український науковий фармакопейний центр якості лікарських засобів».
7. LeCun Y., Bengio Y., Hinton G. Deep learning // Nature. 2015. Vol. 521. P. 436–444.
8. Herrera-Flores T. et al. Automatic Fused Multimodal Deep Learning for Plant Identification. arXiv preprint arXiv:2406.01455. 2024.
9. Pl@ntNet. Pl@ntNet Plant Identification API. URL: https://my.plantnet.org
10. Brown T. et al. Language Models are Few-Shot Learners // Advances in Neural Information Processing Systems. 2020. Vol. 33. P. 1877–1901.
11. Vaswani A. et al. Attention Is All You Need // Advances in Neural Information Processing Systems. 2017. Vol. 30.
12. Thirunavukarasu A. et al. Large language models in medicine // Nature Medicine. 2023. Vol. 29. P. 1930–1940.
13. Richardson C. Microservices Patterns. Shelter Island : Manning Publications, 2018.

14. Fielding R. T., Taylor R. N. Principled Design of the Modern Web Architecture // ACM Transactions on Internet Technology. 2002. Vol. 2, No. 2. P. 115–150.
15. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures : дис. … д-ра філос. наук. University of California, Irvine, 2000.
16. Crockford D. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627. Internet Engineering Task Force, 2006.
17. Jones M. et al. JSON Web Token (JWT). RFC 7519. Internet Engineering Task Force, 2015. URL: https://datatracker.ietf.org/doc/html/rfc7519
18. Longley P. A. et al. Geographic Information Science and Systems. 4th ed. Hoboken : Wiley, 2015.
19. Haklay M., Weber P. OpenStreetMap: User-Generated Street Maps // IEEE Pervasive Computing. 2008. Vol. 7, No. 4. P. 12–18.
20. Leaflet.js. Leaflet – a JavaScript library for interactive maps. URL: https://leafletjs.com
21. Grand View Research. Herbal Medicine Market Size & Growth Report, 2030. 2024. URL: https://www.grandviewresearch.com/industry-analysis/herbal-medicine-market-report
22. iNaturalist. About iNaturalist. URL: https://www.inaturalist.org/pages/about
23. PlantSnap. PlantSnap – Identify Plants, Flowers, Trees & More. URL: https://www.plantsnap.com
24. PictureThis. PictureThis – Plant Identifier. URL: https://www.picturethisai.com
25. Firt J. Progressive Web Apps: The advantages over native apps. 2022. URL: https://www.smashingmagazine.com/2022/06/progressive-web-apps-advantages
26. Microsoft. Windows application development documentation. URL: https://learn.microsoft.com/en-us/windows/apps/
27. Richards M., Ford N. Fundamentals of Software Architecture: A Modern Engineering Approach. 2nd ed. Sebastopol: O'Reilly Media, 2025. 550 p.
28. Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd ed. Sebastopol: O'Reilly Media, 2021.
29. Python Software Foundation. Python Documentation. URL: https://docs.python.org/3/

30. Oracle. Java Documentation. URL: https://docs.oracle.com/en/java/
31. MDN Web Docs. JavaScript Guide. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide
32. Google. Angular Documentation. URL: https://angular.dev/
33. Vue.js. Vue 3 Documentation. URL: https://vuejs.org/guide/
34. Meta. React Documentation. URL: https://react.dev/
35. Tailwind CSS. Tailwind CSS Documentation. URL: https://tailwindcss.com/docs
36. Fastify. Fastify Documentation. URL: https://fastify.dev/docs/latest/
37. OpenJS Foundation. Express.js Documentation. URL: https://expressjs.com/
38. GraphQL Foundation. GraphQL Documentation. URL: https://graphql.org/learn/
39. PostgreSQL Global Development Group. PostgreSQL Documentation. URL: https://www.postgresql.org/docs/
40. Oracle. MySQL Documentation. URL: https://dev.mysql.com/doc/
41. Sequelize. Sequelize ORM Documentation. URL: https://sequelize.org/docs/v6/
42. Prisma. Prisma ORM Documentation. URL: https://www.prisma.io/docs
43. Rescorla E. HTTP Over TLS. RFC 2818. Internet Engineering Task Force, 2000.
44. Provos N., Mazières D. A Future-Adaptable Password Scheme // USENIX Annual Technical Conference. 1999.
45. Google Cloud. Vision AI Documentation. URL: https://cloud.google.com/vision/docs
46. OpenAI. OpenAI Platform Documentation. URL: https://platform.openai.com/docs
47. Anthropic. Claude API Documentation. URL: https://docs.anthropic.com/
48. Google. Gemini API Documentation. URL: https://ai.google.dev/docs

ДОДАТОК А

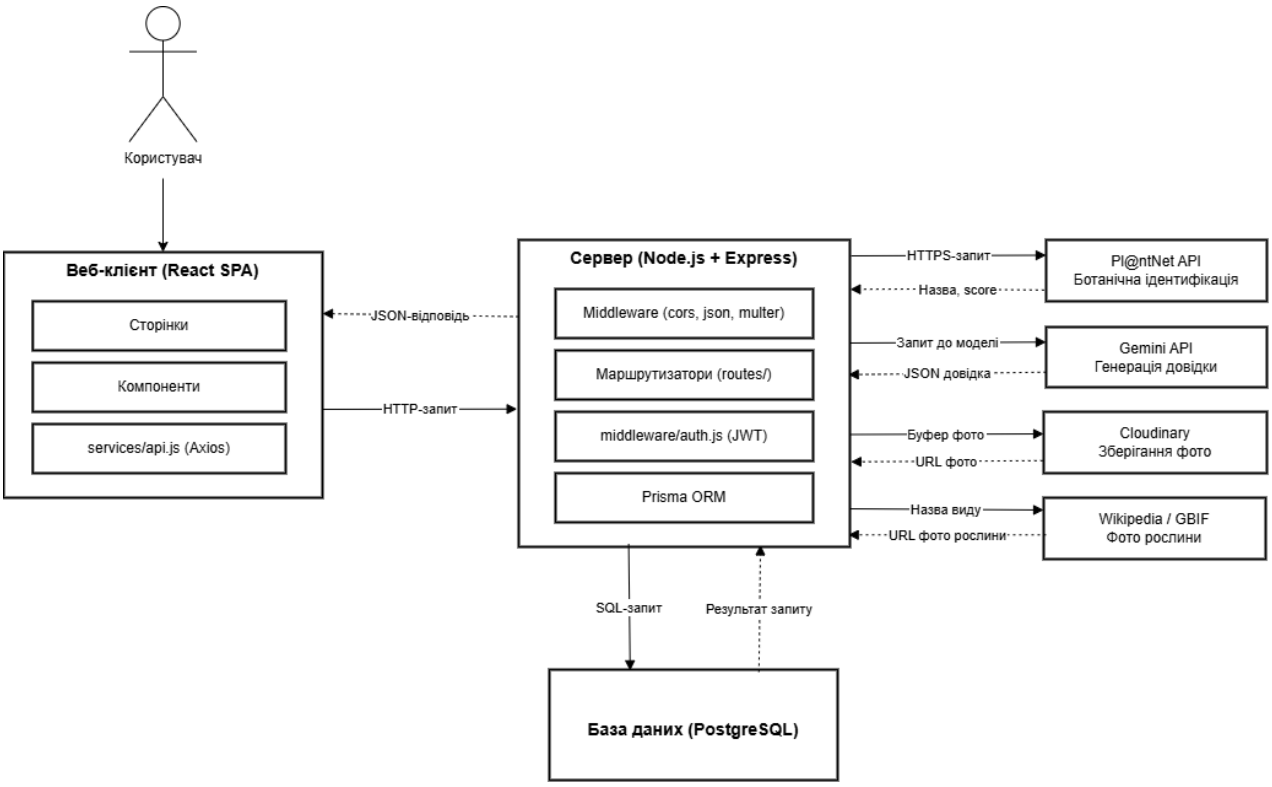
Інтерактивний довідник лікарських рослин на основі застосування штучного інтелекту

Структурна схема системи

ІАЛЦ.467200.004 Д1

Аркушів 1

Київ 2026 р



					ІАЛЦ.467200.004 Д1		
Зм.	Арк.	№ докум.	Підпис	Дата			
Розробила		Фесун А.В.			Інтерактивний довідник лікарських рослин на основі застосування штучного інтелекту	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Павлов В.Г.					
Реценз.		Корнієнко Б.Я.				КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-21	
Н. Контр.		Нікольський С.С.			Структурна схема системи		
Затвердив		Волокита А.М.					

ДОДАТОК Б

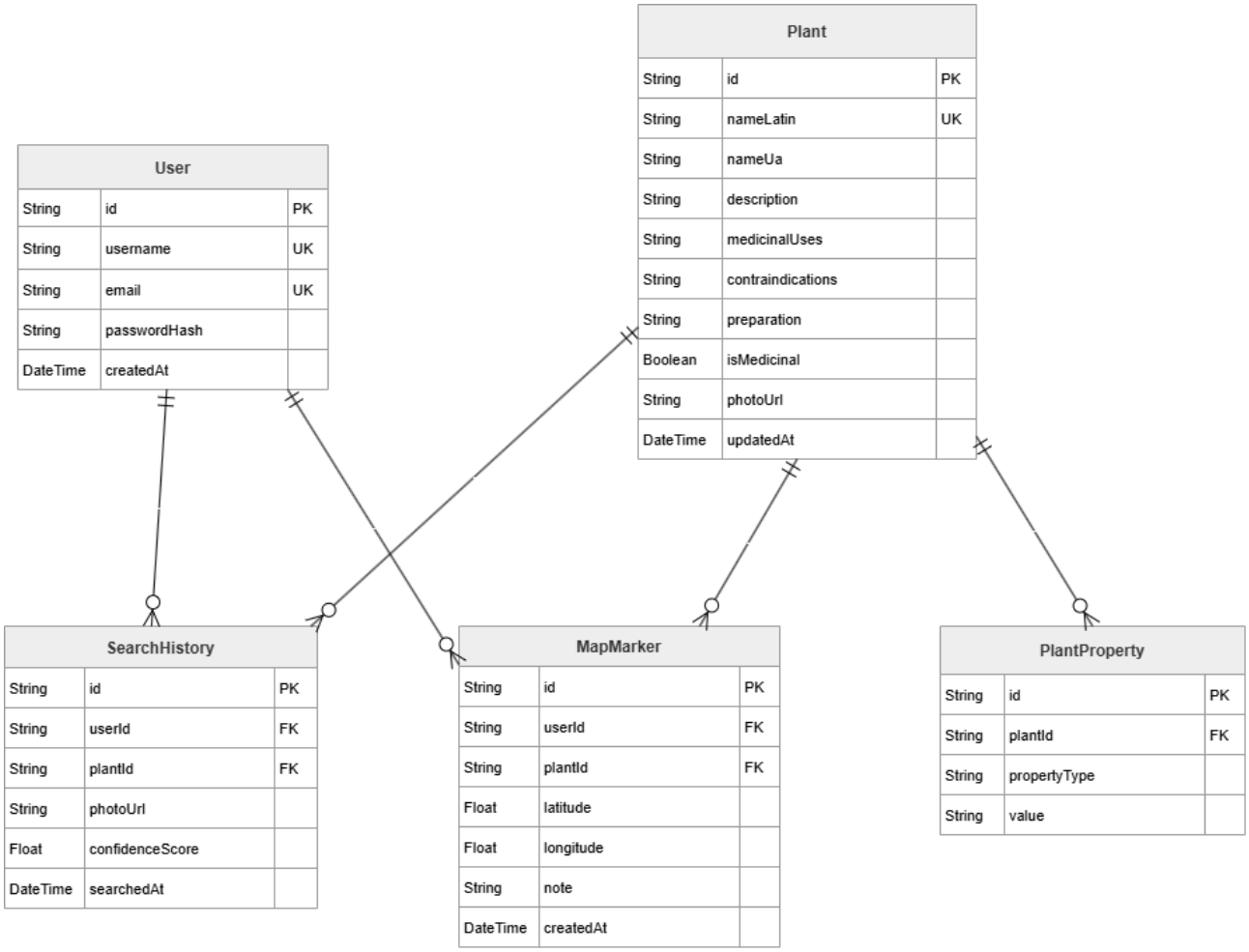
Інтерактивний довідник лікарських рослин на основі застосування штучного інтелекту

ER-діаграма бази даних системи

ІАЛЦ.467200.005 Д2

Аркушів 1

Київ 2026 р



					ІАЛЦ.467200.005 Д2	
Зм.	Арк.	№ докум.	Підпис	Дата		
Розробила		Фесун А.В.			Інтерактивний довідник лікарських рослин на основі застосування штучного інтелекту ER-діаграма бази даних системи	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Павлов В.Г.				
Реценз.		Корнієнко Б.Я.				КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-21
Н. Контр.		Нікольський С.С.				
Затвердив		Волокита А.М.				

ДОДАТОК В

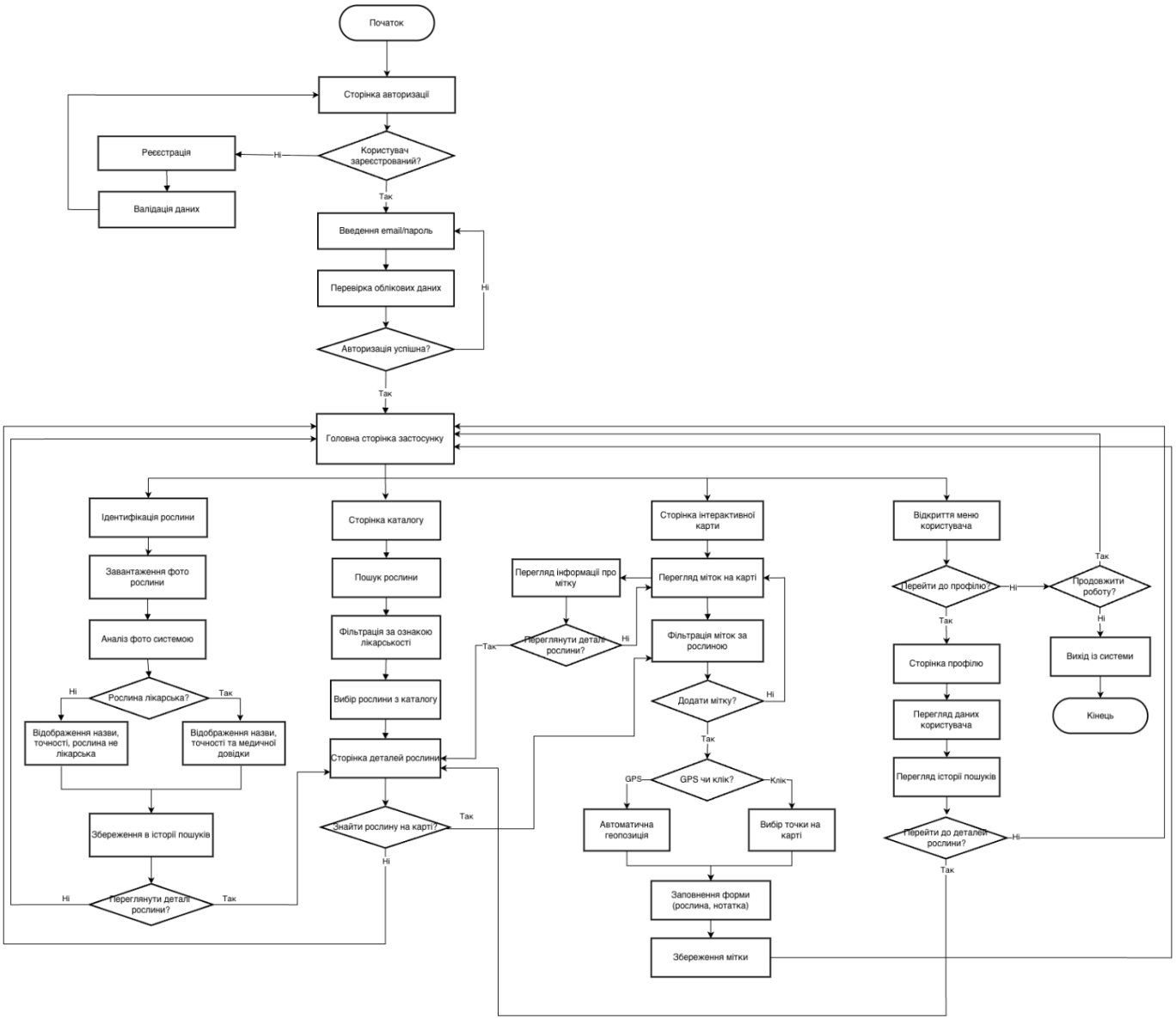
Інтерактивний довідник лікарських рослин на основі застосування штучного інтелекту

Алгоритм дій користувача в системі

ІАЛЦ.467200.006 Д3

Аркушів 1

Київ 2026 р



					ІАЛЦ.467200.006 Д3	
Зм.	Арк.	№ докум.	Підпис	Дата		
Розробила		Фесун А.В.			Інтерактивний довідник лікарських рослин на основі застосування штучного інтелекту Алгоритм дій користувача в системі	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Павлов В.Г.				
Реценз.		Корнієнко Б.Я.				КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-21
Н. Контр.		Нікольський С.С.				
Затвердив		Волокита А.М.				

ДОДАТОК Г

Інтерактивний довідник лікарських рослин на основі застосування штучного інтелекту

Текст програмного коду

ІАЛЦ.467200.007 Д4

Аркушів 47

Київ 2026 р

\server\index.js

```
require('dotenv').config();
const express = require('express');
const cors = require('cors');

const authRoutes = require('./routes/auth');
const plantsRoutes = require('./routes/plants');
const searchRoutes = require('./routes/search');
const mapRoutes = require('./routes/map');
const profileRoutes = require('./routes/profile');

const app = express();
const PORT = process.env.PORT || 5000;

app.use(cors());
app.use(express.json());

app.use('/api/auth', authRoutes);
app.use('/api/plants', plantsRoutes);
app.use('/api/search', searchRoutes);
app.use('/api/map', mapRoutes);
app.use('/api/profile', profileRoutes);

app.get('/', (req, res) => {
  res.json({ message: 'Medicinal Plants API is running' });
});

app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

\server\prisma\schema.prisma

```
generator client {
  provider = "prisma-client-js"
}
datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}
```

```
model User {
  id           String          @id @default(uuid())
  username     String          @unique
  email        String          @unique
  passwordHash String
  createdAt    DateTime        @default(now())

  searchHistory SearchHistory[]
  mapMarkers    MapMarker[]
}

model Plant {
  id             String          @id @default(uuid())
  nameUa         String
  nameLatin      String          @unique
  description    String?
  medicinalUses String?
  contraindications String?
  preparation    String?
  isMedicinal    Boolean         @default(false)
  photoUrl       String?
  externalApiId String?
  updatedAt      DateTime        @updatedAt

  searchHistory SearchHistory[]
  mapMarkers    MapMarker[]
  properties    PlantProperty[]
}

model SearchHistory {
  id              String   @id @default(uuid())
  userId          String
  plantId         String?
  photoUrl        String?
  confidenceScore Float?
  aiAnalysis      String?
  searchedAt      DateTime @default(now())

  user  User   @relation(fields: [userId], references: [id])
  plant Plant? @relation(fields: [plantId], references: [id])
}

model MapMarker {
```

```
  id        String   @id @default(uuid())
  userId    String
  plantId   String
  latitude  Float
  longitude Float
  note      String?
  createdAt DateTime @default(now())

  user  User  @relation(fields: [userId], references: [id])
  plant Plant @relation(fields: [plantId], references: [id])
}

model PlantProperty {
  id           String @id @default(uuid())
  plantId      String
  propertyType String
  value        String

  plant Plant @relation(fields: [plantId], references: [id])
}
```

\server\routes\auth.js

```
       const express = require('express');
       const router = express.Router();
       const bcrypt = require('bcryptjs');
       const jwt = require('jsonwebtoken');
       const { PrismaClient } = require('@prisma/client');

       const prisma = new PrismaClient();

       // POST /api/auth/register
       router.post('/register', async (req, res) => {
         const { username, email, password } = req.body;

         if (!username || !email || !password) {
           return res.status(400).json({ error: 'All fields are required' });
         }

         const existingUser = await prisma.user.findUnique({ where: { email } });
         if (existingUser) {
           return res.status(400).json({ error: 'User with this email already exists' });
         }
```

```
    const passwordHash = await bcrypt.hash(password, 10);

    const user = await prisma.user.create({
      data: { username, email, passwordHash },
    });

    res.status(201).json({ message: 'User registered successfully', userId: user.id
});
  });

  // POST /api/auth/login
  router.post('/login', async (req, res) => {
    const { email, password } = req.body;

    if (!email || !password) {
      return res.status(400).json({ error: 'Email and password are required' });
    }

    const user = await prisma.user.findUnique({ where: { email } });
    if (!user) {
      return res.status(401).json({ error: 'Invalid email or password' });
    }

    const passwordMatch = await bcrypt.compare(password, user.passwordHash);
    if (!passwordMatch) {
      return res.status(401).json({ error: 'Invalid email or password' });
    }

    const token = jwt.sign(
      { userId: user.id },
      process.env.JWT_SECRET,
      { expiresIn: '7d' }
    );

    res.json({ token, username: user.username });
  });

  module.exports = router;
```

\server\routes\map.js

```
  const express = require('express');
  const router = express.Router();
  const { PrismaClient } = require('@prisma/client');
```

```
const authMiddleware = require('../middleware/auth');

const prisma = new PrismaClient();

// GET /api/map — публічний, всі маркери
router.get('/', async (req, res) => {
  const markers = await prisma.mapMarker.findMany({
    include: {
      plant: { select: { id: true, nameUa: true, nameLatin: true, photoUrl: true } },
      user: { select: { username: true } },
    },
    orderBy: { createdAt: 'desc' },
  });

  res.json(markers);
});

// POST /api/map — захищений, додати маркер
router.post('/', authMiddleware, async (req, res) => {
  const { plantId, latitude, longitude, note } = req.body;

  if (!plantId || latitude === undefined || longitude === undefined) {
    return res.status(400).json({ error: 'plantId, latitude and longitude are required.' });
  }

  const lat = parseFloat(latitude);
  const lon = parseFloat(longitude);

  if (isNaN(lat) || isNaN(lon)) {
    return res.status(400).json({ error: 'latitude and longitude must be valid numbers.' });
  }

  const plant = await prisma.plant.findUnique({ where: { id: plantId } });
  if (!plant) {
    return res.status(404).json({ error: 'Plant not found.' });
  }

  const marker = await prisma.mapMarker.create({
    data: {
      userId: req.userId,
```

```
            plantId,
            latitude: lat,
            longitude: lon,
            note: note || null,
          },
          include: {
            plant: { select: { id: true, nameUa: true, nameLatin: true, photoUrl: true }
},
            user: { select: { username: true } },
          },
        });

        res.status(201).json(marker);
      });

      // DELETE /api/map/:id — видалити власну мітку
      router.delete('/:id', authMiddleware, async (req, res) => {
        const marker = await prisma.mapMarker.findUnique({ where: { id: req.params.id }
});
        if (!marker) return res.status(404).json({ error: 'Marker not found.' });
         if (marker.userId !== req.userId) return res.status(403).json({ error:
'Forbidden.' });

        await prisma.mapMarker.delete({ where: { id: req.params.id } });
        res.json({ success: true });
      });

      module.exports = router;
```

\server\routes\plants.js

```
      const express = require('express');
      const router = express.Router();
      const { PrismaClient } = require('@prisma/client');

      const prisma = new PrismaClient();

      // GET /api/plants?search=ромашка
      router.get('/', async (req, res) => {
        const { search } = req.query;

        const plants = await prisma.plant.findMany({
          where: search
            ? {
```

```
              OR: [
                { nameUa: { contains: search, mode: 'insensitive' } },
                { nameLatin: { contains: search, mode: 'insensitive' } },
              ],
            }
          : undefined,
        orderBy: { nameUa: 'asc' },
      });

      res.json(plants);
    });

    // GET /api/plants/:id/adjacent
    router.get('/:id/adjacent', async (req, res) => {
      const { id } = req.params;

      const current = await prisma.plant.findUnique({ where: { id }, select: { nameUa:
true } });
      if (!current) return res.status(404).json({ error: 'Plant not found.' });

      const [prev, next] = await Promise.all([
        prisma.plant.findFirst({
          where: { nameUa: { lt: current.nameUa } },
          orderBy: { nameUa: 'desc' },
          select: { id: true, nameUa: true },
        }),
        prisma.plant.findFirst({
          where: { nameUa: { gt: current.nameUa } },
          orderBy: { nameUa: 'asc' },
          select: { id: true, nameUa: true },
        }),
      ]);

      res.json({ prev: prev || null, next: next || null });
    });

    // GET /api/plants/:id
    router.get('/:id', async (req, res) => {
      const { id } = req.params;

      const plant = await prisma.plant.findUnique({
        where: { id },
        include: { properties: true },
```

```
    });

    if (!plant) {
      return res.status(404).json({ error: 'Plant not found.' });
    }

    res.json(plant);
  });

  module.exports = router;
```

\server\routes\profile.js

```
  const express = require('express');
  const router = express.Router();
  const { PrismaClient } = require('@prisma/client');
  const authMiddleware = require('../middleware/auth');

  const prisma = new PrismaClient();

  // GET /api/profile
  router.get('/', authMiddleware, async (req, res) => {
    const user = await prisma.user.findUnique({
      where: { id: req.userId },
      select: {
        username: true,
        email: true,
        createdAt: true,
        searchHistory: {
          orderBy: { searchedAt: 'desc' },
          include: {
            plant: { select: { id: true, nameUa: true, nameLatin: true, photoUrl: true
} },
          },
        },
      },
    });

    if (!user) {
      return res.status(404).json({ error: 'User not found.' });
    }

    res.json(user);
  });
```

```
module.exports = router;
```

\server\routes\search.js

```
const express = require('express');
const router = express.Router();
const multer = require('multer');
const axios = require('axios');
const FormData = require('form-data');
const { PrismaClient } = require('@prisma/client');
const authMiddleware = require('../middleware/auth');
const cloudinary = require('../config/cloudinary');

const prisma = new PrismaClient();

async function uploadToCloudinary(fileBuffer) {
  return new Promise((resolve, reject) => {
    const stream = cloudinary.uploader.upload_stream(
      {
        folder: 'medicinal-plants/user-uploads',
        transformation: [{ width: 800, crop: 'limit' }],
      },
      (error, result) => {
        if (error) reject(error);
        else resolve(result.secure_url);
      }
    );
    stream.end(fileBuffer);
  });
}

const upload = multer({
  storage: multer.memoryStorage(),
  limits: { fileSize: 5 * 1024 * 1024 }, // 5MB
});

async function getPlantPhoto(nameLatin, gbifId) {
  const wikiName = encodeURIComponent(nameLatin.replace(/ /g, '_'));

  // Крок 1: англійська Wikipedia
  try {
    const response = await axios.get(
      `https://en.wikipedia.org/api/rest_v1/page/summary/${wikiName}`,
      { headers: { 'User-Agent': 'PhytotecaApp/1.0' } }
```

```
          );
                  const     photo     =     response.data?.originalimage?.source     ||
response.data?.thumbnail?.source;
          if (photo) return photo;
        } catch {
          // fallback
        }

        // Крок 2: українська Wikipedia
        try {
          const response = await axios.get(
            `https://uk.wikipedia.org/api/rest_v1/page/summary/${wikiName}`,
            { headers: { 'User-Agent': 'PhytotecaApp/1.0' } }
          );
                  const     photo     =     response.data?.originalimage?.source     ||
response.data?.thumbnail?.source;
          if (photo) return photo;
        } catch {
          // fallback to GBIF
        }

        // Запасний варіант — GBIF occurrence
        if (gbifId) {
          try {
            const response = await axios.get(
              `https://api.gbif.org/v1/occurrence/search?taxonKey=${gbifId}&mediaType=St
illImage&limit=1`
            );
            const results = response.data?.results;
            if (results && results.length > 0) {
              const media = results[0].media;
              if (media && media.length > 0) {
                return media[0].identifier || null;
              }
            }
          } catch {
            // ignore
          }
        }

        return null;
      }
```

```
// POST /api/search/identify
router.post('/identify', authMiddleware, upload.single('photo'), async (req, res)
=> {
  if (!req.file) {
    return res.status(400).json({ error: 'No photo uploaded.' });
  }

  // Завантажуємо фото користувача в Cloudinary
  let userPhotoUrl = null;
  try {
    userPhotoUrl = await uploadToCloudinary(req.file.buffer);
  } catch (err) {
    console.error('Cloudinary upload failed:', err.message);
  }

  try {
    // Крок 1: відправляємо фото до PlantNet
    const formData = new FormData();
    formData.append('images', req.file.buffer, {
      filename: req.file.originalname,
      contentType: req.file.mimetype,
    });
    formData.append('organs', 'auto');

    let plantNetData;
    try {
      const plantNetResponse = await axios.post(
                                    `https://my-api.plantnet.org/v2/identify/all?api-
key=${process.env.PLANTNET_API_KEY}&nb-results=1&lang=uk`,
        formData,
        { headers: formData.getHeaders() }
      );
      plantNetData = plantNetResponse.data;
    } catch (err) {
      if (err.response?.status === 404) {
       return res.status(422).json({ error: 'No plant detected in this photo. Please
try with a clearer photo.' });
      }
      console.error('PlantNet error:', err.response?.data || err.message);
      throw err;
    }

    const topResult = plantNetData.results?.[0];
```

```
            if (!topResult) {
               return res.status(422).json({ error: 'No plant detected. Please try with a
clearer photo.' });
            }

            const nameLatin = topResult.species.scientificNameWithoutAuthor;
            const confidenceScore = topResult.score;
            const commonNames = topResult.species.commonNames;
             const nameFromApi = commonNames && commonNames.length > 0 ? commonNames[0] :
null;
            const gbifId = topResult.gbif?.id || null;
            const plantPhoto = await getPlantPhoto(nameLatin, gbifId);

            const lowConfidence = confidenceScore < 0.3;

            // Крок 2: шукаємо рослину в нашій БД
            let plant = await prisma.plant.findUnique({ where: { nameLatin } });
            let geminiAnalysis = '';

            if (plant) {
              // Крок 3а: рослина є в БД — Gemini не викликаємо
              geminiAnalysis = JSON.stringify({
                nameUa: plant.nameUa,
                description: plant.description,
                medicinalUses: plant.medicinalUses,
                contraindications: plant.contraindications,
                preparation: plant.preparation,
                isMedicinal: plant.isMedicinal,
              });
            } else {
              // Крок 3б: рослини немає — запитуємо Gemini
              const commonNameHint = nameFromApi ? `Its common name is "${nameFromApi}".` :
'';
               const prompt = `You are a botanist. The plant "${nameLatin}" was identified
by image recognition. ${commonNameHint}
       Respond ONLY with valid JSON (no markdown, no extra text) with exactly these fields:
       {
         "nameUa": "Ukrainian name of the plant",
         "description": "General description of the plant written in Ukrainian",
         "medicinalUses": "What the plant is used for medicinally, written in Ukrainian",
         "contraindications": "Who should not use it, written in Ukrainian",
```

```
        "preparation": "How to prepare it (tea, tincture, compress, etc.), written in
Ukrainian",
        "isMedicinal": true
      }
      If the plant is not medicinal, set "isMedicinal" to false and use empty strings for
all other fields.`;

            let geminiResponse;
            try {
              geminiResponse = await axios.post(
               `https://generativelanguage.googleapis.com/v1beta/models/gemini-2.5-flash-
lite:generateContent?key=${process.env.GEMINI_API_KEY}`,
                {
                  contents: [{ parts: [{ text: prompt }] }],
                },
                { headers: { 'content-type': 'application/json' } }
              );
            } catch (err) {
              const detail = err.response?.data || err.message;
              console.error('Gemini error:', detail);
              return res.status(500).json({ error: 'Gemini API failed.', detail });
            }

            const rawText = geminiResponse.data.candidates[0].content.parts[0].text;
            // прибираємо markdown-блок якщо Gemini загорнув JSON у ```json ... ```
             geminiAnalysis = rawText.replace(/^```(?:json)?\s*/i, '').replace(/\s*```$/,
'').trim();

            let plantData;
            try {
              plantData = JSON.parse(geminiAnalysis);
            } catch {
                return res.status(500).json({ error: 'AI returned invalid JSON.', raw:
rawText });
            }

            // Зберігаємо нову рослину в БД
            plant = await prisma.plant.create({
              data: {
                nameLatin,
                nameUa: nameFromApi || plantData.nameUa || nameLatin,
                description: plantData.description || '',
                medicinalUses: plantData.medicinalUses || '',
```

```
            contraindications: plantData.contraindications || '',
            preparation: plantData.preparation || '',
            isMedicinal: plantData.isMedicinal ?? false,
            photoUrl: plantPhoto,
          },
        });
      }

      // Крок 4: завжди зберігаємо в SearchHistory
      await prisma.searchHistory.create({
        data: {
          userId: req.userId,
          plantId: plant.id,
          photoUrl: userPhotoUrl,
          confidenceScore,
          aiAnalysis: geminiAnalysis,
        },
      });

      res.json({
        plant,
        confidenceScore,
        warning: lowConfidence ? 'Low confidence — result may be inaccurate' : null,
      });

    } catch (err) {
      const detail = err.response?.data || err.message;
      console.error('IDENTIFY ERROR:', detail);
      res.status(500).json({ error: 'Identification failed.', detail });
    }
  });

  module.exports = router;
```

\server\middleware\auth.js

```
  const jwt = require('jsonwebtoken');

  module.exports = (req, res, next) => {
    const authHeader = req.headers['authorization'];
    const token = authHeader && authHeader.split(' ')[1]; // Bearer <token>

    if (!token) {
      return res.status(401).json({ error: 'Access denied. No token provided.' });
```

```
  }

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.userId = decoded.userId;
    next();
  } catch (err) {
    return res.status(401).json({ error: 'Invalid or expired token.' });
  }
};
```

\client\src\App.js

```
import { BrowserRouter, Routes, Route, Outlet } from 'react-router-dom';
import Layout from './components/Layout';
import ProtectedRoute from './components/ProtectedRoute';
import Home from './pages/Home';
import Catalogue from './pages/Catalogue';
import PlantDetail from './pages/PlantDetail';
import MapPage from './pages/MapPage';
import Profile from './pages/Profile';
import Login from './pages/Login';
import Register from './pages/Register';

function AppLayout() {
  return (
    <Layout>
      <Outlet />
    </Layout>
  );
}

function App() {
  return (
    <BrowserRouter>
      <Routes>
        {/* Auth-сторінки — без Navbar і Footer */}
        <Route path="/login" element={<Login />} />
        <Route path="/register" element={<Register />} />

        {/* Основні сторінки — з Layout */}
        <Route element={<AppLayout />}>
          <Route path="/" element={<Home />} />
          <Route path="/catalogue" element={<Catalogue />} />
```

```
            <Route path="/plants/:id" element={<PlantDetail />} />
            <Route path="/map" element={<MapPage />} />
            <Route path="/profile" element={
              <ProtectedRoute>
                <Profile />
              </ProtectedRoute>
            } />
          </Route>
        </Routes>
      </BrowserRouter>
    );
  }

  export default App;
```

\client\src\services\api.js

```
  import axios from 'axios';

  const api = axios.create({
    baseURL: 'http://localhost:5000/api',
  });

  api.interceptors.request.use((config) => {
    const token = localStorage.getItem('token');
    if (token) {
      config.headers.Authorization = `Bearer ${token}`;
    }
    return config;
  });

  export function register(data) {
    return api.post('/auth/register', data);
  }

  export function login(data) {
    return api.post('/auth/login', data);
  }

  export function getPlants(search) {
    return api.get('/plants', { params: search ? { search } : {} });
  }

  export function getPlant(id) {
```

```
  return api.get(`/plants/${id}`);
}

export function getAdjacentPlants(id) {
  return api.get(`/plants/${id}/adjacent`);
}

export function identifyPlant(formData) {
  return api.post('/search/identify', formData, {
    headers: { 'Content-Type': 'multipart/form-data' },
  });
}

export function getMarkers() {
  return api.get('/map');
}

export function addMarker(data) {
  return api.post('/map', data);
}

export function deleteMarker(id) {
  return api.delete(`/map/${id}`);
}

export function getProfile() {
  return api.get('/profile');
}

export default api;
```

\client\src\components\UploadPhoto.jsx

```
import { useState, useRef } from 'react';
import { Link } from 'react-router-dom';
import { Upload, Leaf, Loader2 } from 'lucide-react';
import { identifyPlant } from '../services/api';

function AccordionRow({ icon, label, text, colorClass = 'text-gray-600' }) {
  const [open, setOpen] = useState(false);
  if (!text) return null;
  return (
    <div>
      <button
```

```
            onClick={() => setOpen(!open)}
              className="w-full flex items-center justify-between py-3 px-2 text-left
hover:bg-gray-50 rounded-xl transition"
          >
            <span className="flex items-center gap-2 font-medium text-gray-800">
              <span>{icon}</span> {label}
            </span>
            <span className="text-gray-400 text-sm">{open ? '▲' : '▼'}</span>
          </button>
               {open && <p className={`px-2 pb-3 text-sm leading-relaxed
${colorClass}`}>{text}</p>}
        </div>
      );
    }

    function formatSize(bytes) {
      return bytes > 1024 * 1024
        ? (bytes / (1024 * 1024)).toFixed(1) + ' MB'
        : (bytes / 1024).toFixed(0) + ' KB';
    }

    function getConfidenceInfo(score) {
      const percent = Math.round(score * 100);

      if (percent >= 70) return {
        percent,
        message: 'Рослину визначено точно',
        icon: '☑',
        color: 'bg-green-100 text-green-700 border-green-200',
      };

      if (percent >= 50) return {
        percent,
        message: 'Рослина визначена, можливі схожі підвиди',
        icon: '🌿',
        color: 'bg-yellow-100 text-yellow-700 border-yellow-200',
      };

      if (percent >= 26) return {
        percent,
        message: 'Результат приблизний, рекомендуємо звірити з каталогом',
        icon: '☎',
        color: 'bg-orange-100 text-orange-700 border-orange-200',
```

```
  };

  return {
    percent,
    message: 'Низька достовірність — результат може бути неточним',
    icon: '⚠',
    color: 'bg-red-100 text-red-700 border-red-200',
  };
}

function UploadPhoto() {
  const [file, setFile] = useState(null);
  const [preview, setPreview] = useState(null);
  const [dragging, setDragging] = useState(false);
  const [loading, setLoading] = useState(false);
  const [result, setResult] = useState(null);
  const [error, setError] = useState(null);
  const inputRef = useRef(null);

  const isLoggedIn = !!localStorage.getItem('token');

  function applyFile(selected) {
    setFile(selected);
    setPreview(URL.createObjectURL(selected));
    setResult(null);
    setError(null);
  }

  function handleFileChange(e) {
    const f = e.target.files[0];
    if (f) applyFile(f);
  }

  function handleClear(e) {
    e.stopPropagation();
    setFile(null);
    setPreview(null);
    setResult(null);
    setError(null);
    if (inputRef.current) inputRef.current.value = '';
  }

  function handleDragOver(e) {
```

```
    e.preventDefault();
    setDragging(true);
  }

  function handleDragLeave() {
    setDragging(false);
  }

  function handleDrop(e) {
    e.preventDefault();
    setDragging(false);
    const f = e.dataTransfer.files[0];
    if (f && f.type.startsWith('image/')) applyFile(f);
  }

  async function handleIdentify() {
    if (!file) return;
    setLoading(true);
    setError(null);
    setResult(null);
    const formData = new FormData();
    formData.append('photo', file);
    try {
      const res = await identifyPlant(formData);
      setResult(res.data);
    } catch (err) {
        setError(err.response?.data?.error || 'Identification failed. Please try
again.');
    } finally {
      setLoading(false);
    }
  }

  if (!isLoggedIn) {
    return (
        <div className="text-center py-12 border-2 border-dashed border-green-200
rounded-3xl bg-green-50 max-w-lg mx-auto">
          <Leaf className="text-green-300 mx-auto mb-3" size={48} />
          <p className="text-gray-600">
            Будь ласка,{' '}
            <Link to="/login" className="text-green-700 font-medium hover:underline">
              увійдіть
            </Link>
```

```
            , щоб визначати рослини.
          </p>
        </div>
      );
    }

    return (
      <div className="max-w-lg mx-auto">
        {/* Зона завантаження */}
        <div
          onClick={() => !file && inputRef.current?.click()}
          onDragOver={handleDragOver}
          onDragLeave={handleDragLeave}
          onDrop={handleDrop}
           className={`border-2 border-dashed rounded-3xl transition min-h-64 flex
flex-col items-center justify-center gap-3
            ${file ? 'cursor-default' : 'cursor-pointer'}
            ${dragging ? 'border-green-500 bg-green-50' : 'border-green-300 bg-white
hover:bg-green-50'}`}
        >
          {!file ? (
            <>
              <Upload className="text-green-400" size={48} />
               <p className="text-gray-600 font-medium text-center">Перетягніть фото
або натисніть для вибору</p>
              <p className="text-gray-400 text-sm">JPG, PNG до 5MB</p>
            </>
          ) : (
            <div className="w-full p-4 flex flex-col items-center gap-2">
              <img
                src={preview}
                alt="Попередній перегляд"
                className="w-full h-64 object-contain rounded-2xl"
              />
                           <p   className="text-gray-500   text-sm">{file.name}   ·
{formatSize(file.size)}</p>
              <button
                onClick={handleClear}
                className="text-red-400 hover:text-red-600 text-sm transition"
              >
                Видалити
              </button>
            </div>
```

```
            )}
          </div>

          <input
            ref={inputRef}
            type="file"
            accept="image/*"
            onChange={handleFileChange}
            className="hidden"
          />

          {/* Кнопка визначення */}
          <button
            onClick={handleIdentify}
            disabled={!file || loading}
             className={`mt-4 w-full rounded-full py-3 font-semibold flex items-center
justify-center gap-2 transition
              ${!file || loading
                ? 'bg-gray-200 text-gray-400 cursor-not-allowed'
                : 'bg-green-700 text-white hover:bg-green-800 hover:scale-105'}`}
          >
            {loading ? (
              <>
                <Loader2 className="animate-spin" size={18} />
                Визначаємо...
              </>
            ) : (
              'Визначити'
            )}
          </button>

          {!file && !loading && (
             <p className="mt-2 text-center text-xs text-gray-400">Спочатку завантажте
фото</p>
          )}

          {error && <p className="mt-3 text-red-500 text-sm text-center">{error}</p>}

          {/* Картка результату */}
          {result && (
            <div className="mt-6 bg-white border border-green-200 rounded-3xl shadow-md
p-6 animate-fade-in">
              <div className="flex items-start gap-4">
```

```
                    <div className="w-20 h-20 rounded-2xl flex-shrink-0 bg-green-50 flex
items-center justify-center overflow-hidden">
                     {result.plant.photoUrl
                            ? <img src={result.plant.photoUrl} alt={result.plant.nameUa}
className="w-full h-full object-cover" />
                       : <Leaf className="text-green-300" size={36} />
                     }
                    </div>
                    <div className="flex-1 min-w-0">
                            <p  className="font-semibold  text-gray-900  text-lg  leading-
tight">{result.plant.nameUa}</p>
                                    <p   className="text-gray-400   text-sm   italic   mt-
0.5">{result.plant.nameLatin}</p>
                     <div className="flex flex-wrap gap-2 mt-2">
                       {result.plant.isMedicinal && (
                         <span className="bg-green-700 text-white rounded-full text-xs px-
3 py-1">
                           Лікарська
                         </span>
                       )}
                     </div>
                   </div>
                 </div>

                 {(() => {
                   const confidence = getConfidenceInfo(result.confidenceScore);
                   return (
                     <div className={`flex items-center gap-3 rounded-2xl border px-4 py-3
mt-3 ${confidence.color}`}>
                       <span className="text-xl">{confidence.icon}</span>
                       <div>
                             <div  className="font-semibold  text-sm">{confidence.percent}%
впевненості</div>
                                          <div   className="text-xs   mt-0.5   opacity-
80">{confidence.message}</div>
                       </div>
                     </div>
                   );
                 })()}

                 {result.confidenceScore < 0.3 && (
                    <div className="mt-3 bg-red-50 border border-red-200 rounded-2xl px-4
py-3">
```

```
                      <p className="text-red-700 font-semibold text-sm mb-1">⚠ Низька
достовірність результату</p>
                    <ul className="text-red-600 text-xs space-y-1 list-disc list-inside
opacity-90">
                      <li>Сфотографуйте рослину чіткіше, без розмиття</li>
                      <li>Спробуйте знімок на однотонному фоні</li>
                      <li>
                        Звірте результат з{' '}
                        <Link to="/catalogue" className="underline font-medium">
                          каталогом рослин
                        </Link>
                      </li>
                    </ul>
                  </div>
                )}

                <div className="border-t border-gray-100 my-4" />

                {result.plant.isMedicinal && (
                  <>
                                        <AccordionRow    icon="💊"    label="Застосування"
text={result.plant.medicinalUses} />
                                      <AccordionRow    icon="⚠"    label="Протипоказання"
text={result.plant.contraindications} colorClass="text-amber-600" />
                                        <AccordionRow    icon="♨"    label="Приготування"
text={result.plant.preparation} />
                  </>
                )}

                <Link
                  to={`/plants/${result.plant.id}`}
                   className="block mt-4 text-green-700 hover:text-green-800 font-medium
text-sm transition"
                >
                  Детальна сторінка →
                </Link>
              </div>
            )}
          </div>
        );
      }

      export default UploadPhoto;
```

\client\src\pages\Home.jsx

```
import { useEffect } from 'react';
import { Link } from 'react-router-dom';
import { Leaf, BookOpen, MapPin } from 'lucide-react';
import UploadPhoto from '../components/UploadPhoto';

const FEATURES = [
  {
    icon: <Leaf size={48} className="text-green-600 mx-auto" />,
    title: 'AI-визначення',
    desc: 'Сфотографуй рослину і дізнайся все про неї',
  },
  {
    icon: <BookOpen size={48} className="text-green-600 mx-auto" />,
    title: 'Каталог рослин',
    desc: 'База лікарських рослин з детальними описами',
  },
  {
    icon: <MapPin size={48} className="text-green-600 mx-auto" />,
    title: 'Інтерактивна карта',
    desc: 'Позначай де ростуть рослини поруч з тобою',
  },
];

function Home() {
  useEffect(() => { document.title = 'Фітотека — Визначення лікарських рослин'; },
[]);

  function scrollToUpload() {
    document.getElementById('upload-section')?.scrollIntoView({ behavior: 'smooth'
});
  }

  return (
    <div>
      {/* Герой */}
      <section className="-mx-4 sm:-mx-6 lg:-mx-8 px-4 sm:px-6 lg:px-8 py-16 md:py-
24 bg-gradient-to-b from-green-50 to-white">
        <div className="text-center max-w-3xl mx-auto">
          <h1 className="font-heading text-4xl md:text-6xl font-bold text-gray-900">
            Визначай <span className="text-green-700">рослини</span> за фото
          </h1>
```

```
                <p className="text-gray-500 text-lg md:text-xl mt-4 max-w-xl mx-auto">
                    Завантаж фото — і дізнайся все про лікарські властивості рослини за
секунди
                </p>
                <div className="mt-8 flex gap-4 justify-center flex-wrap">
                  <button
                    onClick={scrollToUpload}
                   className="bg-green-700 text-white rounded-full px-8 py-3 font-semibold
hover:bg-green-800 active:scale-95 transition duration-200 hover:scale-105"
                  >
                    Визначити рослину
                  </button>
                  <Link
                    to="/catalogue"
                    className="border-2 border-green-700 text-green-700 rounded-full px-8
py-3 font-semibold hover:bg-green-50 transition"
                  >
                    Переглянути каталог
                  </Link>
                </div>
              </div>
            </section>

            {/* Секція завантаження */}
            <section id="upload-section" className="mt-8">
              <h2 className="font-heading text-2xl font-bold text-gray-900 text-center mb-
6">
                Визначити рослину
              </h2>
              <UploadPhoto />
            </section>

            {/* Переваги */}
            <section className="mt-16">
              <div className="grid grid-cols-1 md:grid-cols-3 gap-6">
                {FEATURES.map(({ icon, title, desc }) => (
                 <div key={title} className="bg-gray-50 rounded-3xl p-8 text-center border
border-green-100">
                    {icon}
                     <h3 className="font-heading font-semibold text-xl text-gray-900 mt-
4">{title}</h3>
                    <p className="text-gray-500 mt-2 text-sm">{desc}</p>
                  </div>
```

```
            ))}
          </div>
        </section>

      </div>
    );
  }

  export default Home;
```

\client\src\pages\Catalogue.jsx

```
  import { useState, useEffect } from 'react';
  import { Search, X } from 'lucide-react';
  import { getPlants } from '../services/api';
  import PlantCard from '../components/PlantCard';

  const PAGE_SIZE = 15;

  const FILTERS = [
    { label: 'Всі', value: 'all' },
    { label: 'Лікарські', value: 'medicinal' },
    { label: 'Не лікарські', value: 'nonmedicinal' },
  ];

  function plantCount(n) {
    if (n % 10 === 1 && n % 100 !== 11) return `${n} рослина`;
    if ([2, 3, 4].includes(n % 10) && ![12, 13, 14].includes(n % 100)) return `${n}
рослини`;
    return `${n} рослин`;
  }

  function SkeletonCard() {
    return (
      <div  className="bg-white  rounded-3xl  overflow-hidden  border  border-gray-100
animate-pulse">
        <div className="bg-gray-200 h-48 w-full" />
        <div className="p-5 space-y-3">
          <div className="bg-gray-200 h-5 rounded-full w-3/4" />
          <div className="bg-gray-200 h-4 rounded-full w-1/2" />
          <div className="bg-gray-200 h-4 rounded-full w-full" />
          <div className="bg-gray-200 h-4 rounded-full w-5/6" />
        </div>
      </div>
```

```
  );
}

function Catalogue() {
  const [allPlants, setAllPlants] = useState([]);
  const [loading, setLoading] = useState(true);
  const [search, setSearch] = useState('');
  const [filter, setFilter] = useState('all');
  const [page, setPage] = useState(1);

  useEffect(() => { document.title = 'Каталог рослин · Фітотека'; }, []);

  useEffect(() => {
    const timer = setTimeout(async () => {
      setLoading(true);
      try {
        const res = await getPlants(search);
        setAllPlants(res.data);
      } catch {
        setAllPlants([]);
      } finally {
        setLoading(false);
      }
    }, search ? 300 : 0);
    return () => clearTimeout(timer);
  }, [search]);

  useEffect(() => { setPage(1); }, [search, filter]);

  const plants = allPlants.filter((p) => {
    if (filter === 'medicinal') return p.isMedicinal;
    if (filter === 'nonmedicinal') return !p.isMedicinal;
    return true;
  });

  const totalPages = Math.ceil(plants.length / PAGE_SIZE);
  const paginated = plants.slice((page - 1) * PAGE_SIZE, page * PAGE_SIZE);

  return (
    <div>
      {/* Зелений заголовок */}
       <section className="-mx-4 sm:-mx-6 lg:-mx-8 px-8 py-16 bg-green-700 text-
white">
```

```
            <h1 className="font-heading text-4xl font-bold">Каталог рослин</h1>
            <p className="mt-2 text-green-100">
              {loading ? 'Завантаження...' : `${plantCount(allPlants.length)} в базі`}
            </p>
          </section>

          {/* Sticky пошук + фільтри */}
           <div className="sticky top-16 bg-white z-40 py-4 border-b border-gray-100 -
mx-4 sm:-mx-6 lg:-mx-8 px-4 sm:px-6 lg:px-8">
            <div className="relative max-w-xl mx-auto">
             <Search className="absolute left-4 top-1/2 -translate-y-1/2 text-gray-400"
size={18} />
              <input
                type="text"
                placeholder="Пошук рослини..."
                value={search}
                onChange={(e) => setSearch(e.target.value)}
                 className="w-full rounded-full border-2 border-gray-200 focus:border-
green-500 pl-12 pr-10 py-3 outline-none transition"
              />
              {search && (
                <button
                  onClick={() => setSearch('')}
                    className="absolute right-4 top-1/2 -translate-y-1/2 text-gray-400
hover:text-gray-600 transition"
                >
                  <X size={18} />
                </button>
              )}
            </div>

            {/* Фільтр-чіпи */}
            <div className="flex gap-2 flex-wrap justify-center mt-3">
              {FILTERS.map(({ label, value }) => (
                <button
                  key={value}
                  onClick={() => setFilter(value)}
                  className={`rounded-full px-4 py-1.5 text-sm transition ${
                    filter === value
                      ? 'bg-green-700 text-white'
                        : 'border border-gray-200 text-gray-500 hover:border-green-500
hover:text-green-700'
                  }`}
```

```
            >
              {label}
            </button>
          ))}
        </div>
      </div>

      {/* Сітка карток */}
      <div className="grid grid-cols-1 sm:grid-cols-2 lg:grid-cols-3 gap-6 mt-8">
        {loading
          ? Array.from({ length: 6 }).map((_, i) => <SkeletonCard key={i} />)
          : paginated.map((plant) => <PlantCard key={plant.id} plant={plant} />)
        }
      </div>

      {/* Порожній стан */}
      {!loading && plants.length === 0 && (
        <div className="text-center py-20">
          <p className="text-6xl mb-4">🌿</p>
          <p className="font-semibold text-gray-700 text-lg">Рослин не знайдено</p>
          <p className="text-gray-400 mt-1">Спробуйте інший запит</p>
        </div>
      )}

      {/* Пагінація */}
      {!loading && totalPages > 1 && (
        <div className="flex items-center justify-center gap-2 mt-10 mb-4">
          <button
            onClick={() => { setPage(page - 1); window.scrollTo({ top: 0, behavior:
'smooth' }); }}
            disabled={page === 1}
             className="px-4 py-2 rounded-full border-2 border-gray-200 text-gray-
500 text-sm font-medium hover:border-green-500 hover:text-green-700 transition
disabled:opacity-30 disabled:cursor-not-allowed"
          >
            ← Назад
          </button>

          {Array.from({ length: totalPages }, (_, i) => i + 1)
            .filter((p) => p === 1 || p === totalPages || Math.abs(p - page) <= 1)
            .reduce((acc, p, idx, arr) => {
              if (idx > 0 && p - arr[idx - 1] > 1) acc.push('...');
              acc.push(p);
```

```
                    return acc;
                  }, [])
                  .map((item, idx) =>
                    item === '...' ? (
                      <span key={`dots-${idx}`} className="text-gray-400 px-1">…</span>
                    ) : (
                      <button
                        key={item}
                       onClick={() => { setPage(item); window.scrollTo({ top: 0, behavior:
'smooth' }); }}
                        className={`w-10 h-10 rounded-full text-sm font-medium transition
${
                          page === item
                            ? 'bg-green-700 text-white'
                            : 'border-2 border-gray-200 text-gray-500 hover:border-green-
500 hover:text-green-700'
                        }`}
                      >
                        {item}
                      </button>
                    )
                  )}

                <button
                  onClick={() => { setPage(page + 1); window.scrollTo({ top: 0, behavior:
'smooth' }); }}
                  disabled={page === totalPages}
                   className="px-4 py-2 rounded-full border-2 border-gray-200 text-gray-
500 text-sm font-medium hover:border-green-500 hover:text-green-700 transition
disabled:opacity-30 disabled:cursor-not-allowed"
                >
                  Вперед →
                </button>
              </div>
            )}
          </div>
        );
      }
      export default Catalogue;
```

\client\src\pages\MapPage.jsx

```
      import { useState, useEffect, useRef } from 'react';
```

```
import { MapContainer, TileLayer, Marker, Popup, useMapEvents, useMap } from 'react-leaflet';
import Select from 'react-select';
import L from 'leaflet';
import 'leaflet/dist/leaflet.css';
import { useSearchParams } from 'react-router-dom';
import { Search } from 'lucide-react';
import MarkerClusterGroup from 'react-leaflet-cluster';
import 'react-leaflet-cluster/dist/assets/MarkerCluster.css';
import 'react-leaflet-cluster/dist/assets/MarkerCluster.Default.css';
import { getMarkers, addMarker, deleteMarker, getPlants } from '../services/api';

const UKRAINE_CENTER = [48.3794, 31.1656];

const greenIcon = L.divIcon({
  html: `<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 24 24" width="32" height="40" fill="#15803D">
    <path d="M12 2C8.13 2 5 5.13 5 9c0 5.25 7 13 7 13s7-7.75 7-13c0-3.87-3.13-7-7-7zm0 9.5c-1.38 0-2.5-1.12-2.5-2.5s1.12-2.5 2.5-2.5 2.5 1.12 2.5 2.5-1.12 2.5-2.5 2.5z"/>
  </svg>`,
  className: '',
  iconSize: [32, 40],
  iconAnchor: [16, 40],
  popupAnchor: [0, -40],
});

const selectedIcon = L.divIcon({
  html: `<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 24 24" width="32" height="40" fill="#DC2626">
    <path d="M12 2C8.13 2 5 5.13 5 9c0 5.25 7 13 7 13s7-7.75 7-13c0-3.87-3.13-7-7-7zm0 9.5c-1.38 0-2.5-1.12-2.5-2.5s1.12-2.5 2.5-2.5 2.5 1.12 2.5 2.5-1.12 2.5-2.5 2.5z"/>
  </svg>`,
  className: '',
  iconSize: [32, 40],
  iconAnchor: [16, 40],
  popupAnchor: [0, -40],
});

function FlyToLocation({ coords }) {
  const map = useMap();
  useEffect(() => {
    if (coords) map.flyTo(coords, 13, { duration: 1.2 });
  }, [coords, map]);
```

```
    return null;
  }

  function ClickHandler({ active, onMapClick }) {
    useMapEvents({
      click(e) {
        if (active) onMapClick(e.latlng);
      },
    });
    return null;
  }

  function ZoomControls() {
    const map = useMap();
    return (
      <div className="absolute bottom-6 left-4 z-[400] flex flex-col gap-1">
        <button
          onClick={() => map.zoomIn()}
          className="w-10 h-10 bg-white border border-green-100 rounded-xl shadow-md flex items-center justify-center text-green-700 text-xl font-light hover:bg-green-50 transition"
        >
          +
        </button>
        <button
          onClick={() => map.zoomOut()}
          className="w-10 h-10 bg-white border border-green-100 rounded-xl shadow-md flex items-center justify-center text-green-700 text-xl font-light hover:bg-green-50 transition"
        >
          −
        </button>
      </div>
    );
  }

  function DraggableMarker({ position, onPositionChange }) {
    const markerRef = useRef(null);

    const eventHandlers = {
      dragend() {
        const marker = markerRef.current;
        if (marker) onPositionChange(marker.getLatLng());
```

```
      },
    };

    return (
      <Marker
        draggable
        eventHandlers={eventHandlers}
        position={position}
        icon={selectedIcon}
        ref={markerRef}
      >
        <Popup>Перетягни мітку на точне місце</Popup>
      </Marker>
    );
  }

  const selectStyles = {
    control: (base) => ({
      ...base,
      borderRadius: '12px',
      borderColor: '#BBF7D0',
      padding: '2px',
      '&:hover': { borderColor: '#14a84d' },
    }),
    option: (base, state) => ({
      ...base,
      backgroundColor: state.isSelected ? '#14a84d' : state.isFocused ? '#DCFCE7' :
'white',
      color: state.isSelected ? 'white' : '#111827',
    }),
    menuPortal: (base) => ({ ...base, zIndex: 9999 }),
  };

  function MapPage() {
    const [markers, setMarkers] = useState([]);
    const [plants, setPlants] = useState([]);
    const [showModal, setShowModal] = useState(false);
    const [modalStep, setModalStep] = useState('choice');
    const [selectedPosition, setSelectedPosition] = useState(null);
    const [selectedPlant, setSelectedPlant] = useState(null);
    const [note, setNote] = useState('');
    const [filterPlant, setFilterPlant] = useState(null);
    const [showMyOnly, setShowMyOnly] = useState(false);
```

```
  const [deleteConfirmId, setDeleteConfirmId] = useState(null);
  const [locationQuery, setLocationQuery] = useState('');
  const [flyToCoords, setFlyToCoords] = useState(null);
  const [locationError, setLocationError] = useState(null);
  const [suggestions, setSuggestions] = useState([]);
  const [showSuggestions, setShowSuggestions] = useState(false);
  const searchRef = useRef(null);
  const [searchParams] = useSearchParams();

  const isLoggedIn = !!localStorage.getItem('token');

  useEffect(() => { document.title = 'Карта рослин · Фітотека'; }, []);

  useEffect(() => {
    if (!locationQuery.trim() || locationQuery.length < 2) {
      setSuggestions([]);
      return;
    }
    const timer = setTimeout(async () => {
      try {
        const res = await fetch(
          `https://nominatim.openstreetmap.org/search?q=${encodeURIComponent(locat
ionQuery)}&format=json&limit=10&countrycodes=ua&accept-language=uk`,
          { headers: { 'Accept-Language': 'uk' } }
        );
        const data = await res.json();
         const priority = ['city', 'town', 'village', 'hamlet', 'municipality',
'administrative'];
        const settlements = data
          .filter((item) => ['place', 'boundary'].includes(item.class))
          .sort((a, b) => {
            const ai = priority.indexOf(a.type);
            const bi = priority.indexOf(b.type);
            return (ai === -1 ? 99 : ai) - (bi === -1 ? 99 : bi);
          });
        setSuggestions(settlements);
        setShowSuggestions(true);
      } catch {
        setSuggestions([]);
      }
    }, 300);
    return () => clearTimeout(timer);
  }, [locationQuery]);
```

```
  useEffect(() => {
    function handleClickOutside(e) {
      if (searchRef.current && !searchRef.current.contains(e.target)) {
        setShowSuggestions(false);
      }
    }
    document.addEventListener('mousedown', handleClickOutside);
    return () => document.removeEventListener('mousedown', handleClickOutside);
  }, []);

  useEffect(() => {
    const plantId = searchParams.get('plantId');
    const plantName = searchParams.get('plantName');
    if (plantId && plantName) {
      setFilterPlant({ value: plantId, label: decodeURIComponent(plantName) });
    }
  }, [searchParams]);

  useEffect(() => {
    getMarkers().then((res) => setMarkers(res.data)).catch(() => setMarkers([]));
    getPlants().then((res) => {
       setPlants(res.data.map((plant) => ({ value: plant.id, label: plant.nameUa
}))));
    }).catch(() => setPlants([]));
  }, []);

  function resetModal() {
    setShowModal(false);
    setModalStep('choice');
    setSelectedPosition(null);
    setSelectedPlant(null);
    setNote('');
  }

  function handleGPS() {
    if (!navigator.geolocation) {
      alert('Геолокація не підтримується вашим браузером');
      return;
    }
    navigator.geolocation.getCurrentPosition(
      (pos) => {
```

```
                setSelectedPosition({ lat: pos.coords.latitude, lng: pos.coords.longitude
});
                setModalStep('form');
                setShowModal(true);
              },
              () => {
                alert('Не вдалось визначити геопозицію. Спробуйте вибрати місце на карті.');
                setModalStep('choice');
              }
            );
          }

          async function handleDeleteMarker() {
            try {
              await deleteMarker(deleteConfirmId);
              setMarkers((prev) => prev.filter((m) => m.id !== deleteConfirmId));
              setDeleteConfirmId(null);
            } catch (err) {
              console.error('Помилка видалення мітки:', err);
            }
          }

          async function handleAddMarker() {
            if (!selectedPlant || !selectedPosition) return;
            try {
              await addMarker({
                plantId: selectedPlant.value,
                latitude: selectedPosition.lat,
                longitude: selectedPosition.lng,
                note,
              });
              const res = await getMarkers();
              setMarkers(res.data);
              resetModal();
            } catch (err) {
              console.error('Помилка додавання мітки:', err);
            }
          }

          function handleSelectSuggestion(item) {
            setLocationQuery(item.display_name.split(',')[0]);
            setFlyToCoords([parseFloat(item.lat), parseFloat(item.lon)]);
            setSuggestions([]);
```

```
            setShowSuggestions(false);
            setLocationError(null);
          }

          async function handleLocationSearch(e) {
            e.preventDefault();
            if (!locationQuery.trim()) return;
            setLocationError(null);
            setShowSuggestions(false);
            if (suggestions.length > 0) {
              handleSelectSuggestion(suggestions[0]);
              return;
            }
            try {
              const res = await fetch(
                 `https://nominatim.openstreetmap.org/search?q=${encodeURIComponent(locatio
nQuery)}&format=json&limit=1&countrycodes=ua`,
                { headers: { 'Accept-Language': 'uk' } }
              );
              const data = await res.json();
              if (data.length === 0) {
                setLocationError('Місце не знайдено. Спробуйте іншу назву.');
                return;
              }
              setFlyToCoords([parseFloat(data[0].lat), parseFloat(data[0].lon)]);
            } catch {
              setLocationError('Помилка пошуку. Перевірте з\'єднання.');
            }
          }

          const filteredMarkers = markers.filter((marker) => {
            if (filterPlant && marker.plant?.id !== filterPlant.value) return false;
             if (showMyOnly && marker.user?.username !== localStorage.getItem('username'))
return false;
            return true;
          });

          return (
            <div>
                <h1 className="font-heading text-3xl font-bold text-gray-900 mb-2">Карта
рослин</h1>
                <p className="text-gray-500 mb-6">Позначення місць, де ростуть лікарські
рослини</p>
```

```
            {/* Пошук і фільтр в одному рядку */}
            <div className="flex flex-wrap gap-3 mb-4 items-center">
              <form onSubmit={handleLocationSearch} className="flex items-center">
                <div className="relative" ref={searchRef}>
                  <input
                    type="text"
                    value={locationQuery}
                                onChange={(e)  =>  {  setLocationQuery(e.target.value);
setShowSuggestions(true); }}
                    onFocus={() => suggestions.length > 0 && setShowSuggestions(true)}
                    placeholder="Пошук міста або адреси..."
                     className="rounded-full border-2 border-gray-200 focus:border-green-
500 pl-4 pr-11 py-2.5 outline-none transition w-80"
                  />
                  <button
                    type="submit"
                      className="absolute right-3 top-1/2 -translate-y-1/2 text-gray-400
hover:text-green-700 transition"
                  >
                    <Search size={18} />
                  </button>

                  {showSuggestions && suggestions.length > 0 && (
                     <ul className="absolute top-full left-0 mt-1 w-full bg-white border
border-gray-200 rounded-2xl shadow-lg z-50 overflow-hidden">
                      {suggestions.map((item) => (
                        <li
                          key={item.place_id}
                          onClick={() => handleSelectSuggestion(item)}
                           className="px-4 py-2.5 text-sm text-gray-700 hover:bg-green-50
hover:text-green-700 cursor-pointer transition"
                        >
                          {item.display_name.split(',').slice(0, 2).join(',')}
                        </li>
                      ))}
                    </ul>
                  )}
                </div>
              </form>

              <div className="w-72">
                <Select
```

```
                options={plants}
                value={filterPlant}
                onChange={setFilterPlant}
                placeholder="Фільтр за рослиною..."
                isClearable
                noOptionsMessage={() => 'Рослину не знайдено'}
                styles={selectStyles}
                menuPortalTarget={document.body}
                menuPosition="fixed"
              />
            </div>
          </div>
              {locationError && <p className="text-red-500 text-sm -mt-2 mb-3">{locationError}</p>}

          {/* Лічильник міток */}
          <p className="text-gray-500 font-light mb-3">
            {filteredMarkers.length === markers.length ? (
                        <>Показано <span className="text-green-700 font-extrabold">{markers.length}</span> міток</>
            ) : (
                        <>Показано <span className="text-green-700 font-extrabold">{filteredMarkers.length}</span> з <span className="text-green-700 font-extrabold">{markers.length}</span> міток</>
            )}
          </p>

          {/* Підказка режиму вибору */}
          {modalStep === 'selecting' && (
            <div className="mb-4 bg-green-50 border border-green-200 rounded-2xl px-5 py-3 flex items-center justify-between">
              <span className="text-green-700 font-medium">🗺 Клікни на карту щоб вибрати місце</span>
              <button
                onClick={resetModal}
                className="text-gray-400 hover:text-gray-600 transition text-sm"
              >
                Скасувати
              </button>
            </div>
          )}

          {/* Карта */}
```

```
            <div className="relative rounded-3xl overflow-hidden border border-green-100
shadow-sm z-0">
              <div className="h-[60vh] md:h-[70vh]">
                <MapContainer
                  center={UKRAINE_CENTER}
                  zoom={6}
                  zoomControl={false}
                 style={{ height: '100%', width: '100%', cursor: modalStep === 'selecting'
? 'crosshair' : undefined }}
                >
                  <ZoomControls />
                  <FlyToLocation coords={flyToCoords} />
                  <TileLayer
                    url="https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png"
                                                          attribution='&copy;       <a
href="https://www.openstreetmap.org/copyright">OpenStreetMap</a>'
                  />

                  <ClickHandler
                    active={modalStep === 'selecting'}
                    onMapClick={(latlng) => {
                      setSelectedPosition(latlng);
                      setModalStep('form');
                      setShowModal(true);
                    }}
                  />

                  {/* Перетягуваний маркер */}
                  {selectedPosition && modalStep !== 'choice' && (
                    <DraggableMarker
                      position={selectedPosition}
                      onPositionChange={(latlng) => setSelectedPosition(latlng)}
                    />
                  )}

                  {/* Існуючі маркери з кластеризацією */}
                  <MarkerClusterGroup chunkedLoading maxClusterRadius={60}>
                    {filteredMarkers.map((marker) => (
                        <Marker
                          key={marker.id}
                          position={[marker.latitude, marker.longitude]}
                          icon={greenIcon}
                        >
```

```
                        <Popup minWidth={200}>
                          {marker.plant?.photoUrl && (
                            <img
                              src={marker.plant.photoUrl}
                              alt={marker.plant?.nameUa}
                                  style={{ width: '100%', height: '120px', objectFit:
'cover', borderRadius: '8px', marginBottom: '8px' }}
                            />
                          )}
                                                  <strong   style={{   fontSize:   '14px'
}}>{marker.plant?.nameUa}</strong><br />
                                    <em  style={{  color:  '#9CA3AF',  fontSize:  '12px'
}}>{marker.plant?.nameLatin}</em><br />
                          {marker.note && (
                            <span style={{ fontSize: '13px', color: '#4B5563' }}>
                               {marker.note}<br />
                            </span>
                          )}
                                       <small  style={{  color:  '#9CA3AF'  }}>Додав:
{marker.user?.username}</small><br />
                              <div  style={{  display:  'flex',  alignItems:  'center',
justifyContent: 'space-between', marginTop: '6px' }}>
                          <a
                            href={`/plants/${marker.plantId}`}
                             style={{ color: '#15803D', fontSize: '13px', fontWeight:
'600' }}
                          >
                            Детальніше →
                          </a>
                          {marker.user?.username === localStorage.getItem('username')
&& (
                            <button
                              onClick={() => setDeleteConfirmId(marker.id)}
                               style={{ color: '#9CA3AF', fontSize: '12px', background:
'none', border: 'none', cursor: 'pointer', padding: '0' }}
                              onMouseOver={(e) => e.target.style.color = '#EF4444'}
                              onMouseOut={(e) => e.target.style.color = '#9CA3AF'}
                            >
                              Видалити
                            </button>
                          )}
                        </div>
                      </Popup>
```

```
                        </Marker>
                      ))}
                    </MarkerClusterGroup>
                  </MapContainer>
                </div>

                {/* Перемикач Всі / Мої мітки */}
                {isLoggedIn && (
                  <div className="absolute top-4 left-4 z-[400] flex gap-1 bg-white rounded-
full shadow-md p-1 border border-green-100">
                    <button
                      onClick={() => setShowMyOnly(false)}
                       className={`rounded-full px-3 py-1.5 text-sm font-medium transition
${
                         !showMyOnly ? 'bg-green-700 text-white' : 'text-gray-600 hover:bg-
gray-100'
                      }`}
                    >
                      Всі мітки
                    </button>
                    <button
                      onClick={() => setShowMyOnly(true)}
                       className={`rounded-full px-3 py-1.5 text-sm font-medium transition
${
                         showMyOnly ? 'bg-green-700 text-white' : 'text-gray-600 hover:bg-
gray-100'
                      }`}
                    >
                      Мої мітки
                    </button>
                  </div>
                )}

                {/* FAB кнопка */}
                {isLoggedIn && (
                  <button
                    onClick={() => { setShowModal(true); setModalStep('choice'); }}
                     className="absolute bottom-6 right-6 bg-green-700 text-white rounded-
full w-14 h-14 flex items-center justify-center shadow-lg hover:bg-green-800 transition
hover:scale-110 z-[400] text-2xl"
                  >
                    +
                  </button>
```

```
          )}
        </div>

        {/* Підтвердження видалення мітки */}
        {deleteConfirmId && (
          <div
            className="fixed inset-0 bg-black/40 z-[1000] flex items-center justify-
center px-4"
            onClick={() => setDeleteConfirmId(null)}
          >
            <div
              className="bg-white rounded-3xl p-8 w-full max-w-sm shadow-xl"
              onClick={(e) => e.stopPropagation()}
            >
                  <h2 className="font-heading text-xl font-bold text-gray-900 mb-
2">Видалити мітку?</h2>
               <p className="text-gray-500 text-sm mb-6">Цю дію неможливо скасувати.</p>
                <div className="flex gap-3">
                  <button
                    onClick={handleDeleteMarker}
                    className="flex-1 border-2 border-gray-200 text-gray-500 rounded-xl
py-3 font-medium hover:bg-gray-50 transition"
                  >
                    Видалити
                  </button>
                  <button
                    onClick={() => setDeleteConfirmId(null)}
                      className="flex-1 bg-green-700 text-white rounded-xl py-3 font-
semibold hover:bg-green-800 transition"
                  >
                    Скасувати
                  </button>
                </div>
              </div>
            </div>
        )}

        {/* Модальне вікно */}
        {showModal && (
          <div
            className="fixed inset-0 bg-black/40 z-[1000] flex items-center justify-
center px-4"
            onClick={resetModal}
```

```
            >
              <div
                className="bg-white rounded-3xl p-8 w-full max-w-md shadow-xl"
                onClick={(e) => e.stopPropagation()}
              >
                {/* Крок 1 — вибір способу */}
                {modalStep === 'choice' && (
                  <>
                       <h2 className="font-heading text-xl font-bold text-gray-900 mb-
2">Додати мітку</h2>
                            <p  className="text-gray-500  text-sm  mb-6">Як  визначити
місцезнаходження рослини?</p>

                     <div className="flex flex-col gap-3">
                       <button
                        onClick={() => { setShowModal(false); setModalStep('selecting');
}}
                        className="w-full border-2 border-green-200 rounded-2xl px-6 py-
4 text-left hover:border-green-500 hover:bg-green-50 transition"
                       >
                            <div  className="font-semibold text-gray-900">🗺  Вибрати на
карті</div>
                         <div className="text-gray-400 text-sm mt-1">Клікни на потрібне
місце на карті</div>
                       </button>

                       <button
                         onClick={() => { setModalStep('gps'); handleGPS(); }}
                        className="w-full border-2 border-green-200 rounded-2xl px-6 py-
4 text-left hover:border-green-500 hover:bg-green-50 transition"
                       >
                                 <div  className="font-semibold  text-gray-900">🎯  Моя
геопозиція</div>
                         <div className="text-gray-400 text-sm mt-1">Використати поточне
місцезнаходження</div>
                       </button>
                     </div>

                     <button
                       onClick={resetModal}
                      className="w-full mt-4 text-gray-400 hover:text-gray-600 transition
text-sm py-2"
                     >
```

```
                    Скасувати
                  </button>
                </>
              )}

              {/* Крок 2 — форма */}
              {modalStep === 'form' && (
                <>
                    <h2 className="font-heading text-xl font-bold text-gray-900 mb-
2">Інформація про мітку</h2>
                  <p className="text-gray-500 text-sm mb-2">
                                          📍 {selectedPosition?.lat.toFixed(5)},
{selectedPosition?.lng.toFixed(5)}
                  </p>
                  <p className="text-green-600 text-xs bg-green-50 rounded-xl px-3 py-
2 mb-4">
                    💡 Мітку можна перетягнути на точніше місце прямо на карті
                  </p>

                    <label  className="block  text-sm  font-medium  text-gray-700  mb-
1">Рослина *</label>
                  <Select
                    options={plants}
                    value={selectedPlant}
                    onChange={setSelectedPlant}
                    placeholder="Вибери рослину..."
                    isClearable
                    noOptionsMessage={() => 'Рослину не знайдено'}
                    styles={selectStyles}
                    className="mb-4"
                  />

                    <label  className="block  text-sm  font-medium  text-gray-700  mb-
1">Нотатка</label>
                  <textarea
                    value={note}
                    onChange={(e) => setNote(e.target.value)}
                    placeholder="Наприклад: росте біля річки..."
                    rows={3}
                     className="w-full border-2 border-gray-200 rounded-xl px-4 py-3
focus:border-green-500 outline-none transition resize-none mb-6"
                  />
```

```
                        <div className="flex gap-3">
                          <button
                            onClick={resetModal}
                           className="flex-1 border-2 border-gray-200 text-gray-600 rounded-
xl py-3 font-medium hover:bg-gray-50 transition"
                          >
                            Скасувати
                          </button>
                          <button
                            onClick={handleAddMarker}
                            disabled={!selectedPlant}
                            className="flex-1 bg-green-700 text-white rounded-xl py-3 font-
semibold hover:bg-green-800 transition disabled:opacity-50 disabled:cursor-not-allowed"
                          >
                            Додати
                          </button>
                        </div>
                      </>
                    )}
                  </div>
                </div>
              )}
            </div>
          );
        }

        export default MapPage;
```