НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
Факультет інформаційних технологій

ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ
Завідувач кафедри
Комп'ютерних наук
_____ Голуб Б.Л.

«___»_____2025 р

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему

«Інформаційна система мережі ветеринарних клінік»

Спеціальність 122 «Комп'ютерні науки»

Гарант освітньої програми

Д.е.н., професор _____ Руденський Р.А.

Керівник бакалаврської кваліфікаційної роботи

К.ф.-м.н., доцент _____ Кириченко В.В.

Виконала _____ Бородай А.А.

КИЇВ – 2025

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
Факультет інформаційних технологій

ЗАТВЕРДЖУЮ
Завідувач кафедри
Комп'ютерних наук
_____ Голуб Б.Л.

"___" _____ 2025 р.

ЗАВДАННЯ

на виконання бакалаврської кваліфікаційної роботи студентці
Бородай Анастасії Андріївні

Спеціальність 122 - «Комп'ютерні науки»

Тема бакалаврської кваліфікаційної роботи Інформаційна система мережі ветеринарних клінік

затверджена наказом ректора НУБіП України від «16» 12 2024 р. № 2246 «С»
Термін подання завершеної роботи на кафедру 2025 . ___. ___
рік, місяць, число

Вихідні дані до бакалаврської кваліфікаційної роботи:

опис інформаційної системи

Перелік питань, які потрібно розробити:

1. Аналіз проблемної області.
2. Вибір та обґрунтування засобів для розробки системи.
3. Проектування інформаційної системи.
4. Розробка та тестування інформаційної системи.
5. Висновки.

Дата видачі завдання "_____" _____ 20___ р.
Керівник бакалаврської кваліфікаційної роботи

К.ф.-м.н., доцент _____ Кириченко В.В.

Завдання прийняла до виконання _____ Бородай.А.А

ЗМІСТ

ВСТУП

У сучасному світі дедалі більше людей усвідомлюють важливість своєчасного догляду за здоров’ям домашніх тварин. Проте власники часто стикаються з труднощами під час пошуку надійної ветеринарної клініки, фахового лікаря або необхідної послуги. Відсутність централізованої цифрової платформи значно ускладнює процес отримання медичної допомоги для тварин — особливо в екстрених випадках або при переїздах до іншого району.

У зв’язку з цим зростає потреба в зручних, швидких і надійних інформаційних системах, які дозволять власникам тварин швидко знаходити найближчі ветеринарні клініки, знайомитись із переліком послуг, фахівцями, графіками роботи та здійснювати онлайн-бронювання. Автоматизація цих процесів дозволяє суттєво знизити навантаження на адміністративний персонал клінік, підвищити точність запису, зменшити ймовірність помилок і покращити взаємодію з власниками тварин.

Розвиток мобільних технологій та сучасних способів програмування створює можливості для впровадження інноваційних рішень. Саме тому було поставлене завдання — розробити багатофункціональну інформаційну систему, яка забезпечувала б зручний пошук, перегляд та бронювання ветеринарних послуг у клініках мережі. В якості інструментів реалізації було обрано фрейморк Flutter — для створення кросплатформеного мобільного додатку з мовою Dart, Golang — як надійний і продуктивний засіб для розробки серверної логіки, та MongoDB — для гнучкого зберігання інформації у форматі документів.

Таким чином, метою бакалаврської кваліфікаційної роботи є створення інформаційної системи для мережі ветеринарних клінік, яка дозволить:

- забезпечити централізований доступ до інформації про клініки, лікарів та послуги;
- підвищити рівень цифровізації та обслуговування в галузі ветеринарії.

Для досягнення цієї мети передбачено виконання таких основних завдань:

- аналіз предметної області та існуючих рішень;
- визначення функціональних вимог до системи;
- проектування архітектури системи та побудова UML-діаграм;
- розробка структури бази даних MongoDB;
- створення мобільного додатку на Flutter;
- реалізація backend-сервісів на Go;
- проведення тестування та оцінка ефективності розробленого продукту.

Результатом роботи стане сучасна, інтуїтивно зрозуміла та ефективна система, яка підвищить доступність ветеринарних послуг для власників тварин і оптимізує роботу ветеринарних закладів.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Постановка завдання

У сфері ветеринарних клінік спостерігається суттєве зростання попиту на цифровизацію, які забезпечують зручну взаємодію між власниками тварин та клініками. Незважаючи на це, значна частина закладів досі не має сучасних засобів для управління записами, реклами, демонстрації послуг і т.к. Власники тварин, у свою чергу, часто змушені витрачати час на пошук контактів клінік у мережі, телефонні дзвінки, уточнення наявності послуг чи вільних годин прийому.

Проблема полягає у відсутності єдиної платформи, яка б поєднувала мережу ветеринарних закладів, надавала актуальну інформацію про лікарів та послуги а також забезпечувала зручний доступ до пошуку данних, без попереднього знання проблеми своїх улюбленців.

Для вирішення цієї проблеми пропонується розробити інформаційну систему, орієнтовану на мобільні пристрої, яка дозволить:

- шукати ветеринарні клініки за геолокацією або спеціалізацією;
- дозволить клінікам вносити свої дані;
- переглядати профілі лікарів із зазначенням графіків, досвіду та послуг;
- сучасні технології NLP пошуку.

Основними вимогами до системи є: мобільність, сучасний інтерфейс, швидкий обмін даними та масштабованість. У ролі технологічної основи обрано такі інструменти:

- **Flutter** — для створення клієнтського мобільного застосунку, що працює як на Android, так і на iOS;
- **Golang** — для побудови серверної частини з RESTful API;

- **MongoDB** — як документоорієнтовану базу даних, що забезпечує гнучке зберігання структурованої та напівструктурованої інформації.

Таким чином, задача полягає у проектуванні та створенні системи, яка сприятиме цифровій трансформації ветеринарної галузі та значно покращить якість сервісу для кінцевого користувача — власника тварини.

1.2 Огляд інформаційних джерел та існуючих рішень

На сучасному етапі розвитку технологій дедалі більше галузей переходять до використання технологій для підвищення ефективності обслуговування клієнтів та оптимізації внутрішніх процесів. Ветеринарна медицина не є винятком, однак рівень цифровізації у цьому секторі залишається недостатнім, особливо в умовах локального українського ринку.

Під час підготовки роботи було проаналізовано низку інформаційних джерел, присвячених телемедицині, управлінню ветеринарними клініками, онлайн-сервісам для запису до лікаря та мобільним застосункам у сфері pet tech. Окрема увага приділялася тим системам, які мають функціонал бронювання, геопозиціювання, інтерактивного розкладу роботи персоналу та розумним пошуком.

На світовому ринку існує кілька платформ, які частково охоплюють описану проблематику:

- **Vetster** — орієнтований на проведення онлайн-консультацій з ветеринарами, не надає доступу до інформації про фізичні клініки, не має функціоналу геопошуку.
- **Airvet** — сервіс для відеозв’язку з ветеринарами, без функцій фільтрації клінік.
- **PetDesk** — мобільний застосунок, що дозволяє запис на прийом до клінік, які є партнерами платформи; не підтримує незалежний пошук нових клінік за локацією.

Усі перераховані рішення мають певні обмеження: вони або вузько спеціалізовані, або географічно недоступні для користувачів в Україні, або не забезпечують повного циклу цифрової взаємодії між клієнтом, клінікою та ветеринаром.

З метою систематизації результатів аналізу сформовано порівняльну таблицю (табл. 1), в якій представлені ключові функціональні характеристики кожного з досліджених сервісів.

Порівняння функціональності інформаційних рішень у сфері ветеринарнарної медицини

Таблиця 1

Назва платформи	**Онлайн-запис**	**Геопошук клінік**	**Профілі лікарів**	**Медична історія тварини**	**Мобільний застосунок**	**Працює в Україні**
Vetster	Є	Немає	Є	Немає	Є	Не працює
Airvet	Є	Немає	Є	Немає	Є	Не працює
PetDesk	Є	Немає	Є	Є	Є	Не працює
Моя інформаційна система	Є	Є	Є	Є	Є	Працює

Як видно з аналізу, жодне з існуючих рішень не забезпечує повноцінну роботу із фізичними клініками у форматі геопошуку, перегляду послуг, доступу до профілів лікарів у межах однієї системи. Саме тому виникла

потреба створення нової інформаційної системи, адаптованої до українського ринку, яка поєднуватиме в собі мобільність, простоту використання та повний набір сервісів для клієнтів і клінік.

1.3 Моделювання предметної області

Моделювання предметної області є основою для створення логічної структури інформаційної системи, яка відображає зв’язки між ключовими об’єктами, ролями користувачів та сценаріями їхньої взаємодії з системою. У межах цієї бакалаврської кваліфікаційної роботи предметною областю виступає цифрова система для пошуку ветеринарних клінік, перегляду послуг, лікарів, відгуків та годин роботи клінік.

На сьогодні велика частина власників тварин користується пошуком через загальні сервіси на кшталт Google Maps або соціальні мережі, що не завжди дозволяє швидко порівняти ветеринарні установи, перевірити графік роботи чи ознайомитися з послугами. Проблему ускладнює відсутність централізованої системи, що дозволяє фільтрувати клініки за спеціалізацією, видами тварин, розташуванням або цінами.

Пропонована система призначена для задоволення потреб двох основних груп користувачів:

- Клієнтів (власників тварин), які шукають клініку за місцем розташування, видом тварини або послугою, переглядають відгуки, графік роботи, рейтинги;
- Адміністраторів клінік, які створюють і оновлюють інформацію про заклад, перелік послуг, лікарів, години роботи, а також обробляють відгуки.

Основні об’єкти предметної області

Таблиця 2

Сутність	**Опис**
Клініка	Назва, адреса, опис, локація на карті, графік роботи, рейтинг, список лікарів і послуг.
Лікар	Ім’я, спеціалізація, досвід, клініка, список послуг, фото та коротка біографія.
Послуга	Назва процедури, короткий опис, ціна, доступність у клініках.
Відгук	Текстовий коментар користувача, рейтинг (1–5), дата, клініка, що коментується.
Години роботи	Дані про графік прийому клініки на кожен день тижня, відображаються у профілі закладу.
Категорія	Назва, опис

Уся інформація зберігається в документоорієнтованій базі MongoDB, що дозволяє легко реалізувати вкладену структуру (наприклад, години роботи всередині документа “клініка”, або відгуки всередині об’єкта “клініка”).

Основні дії користувача в системі

1. Пошук клініки за розташуванням (через карту або фільтр);
2. Вибір клініки та перегляд детальної інформації;
3. Ознайомлення з переліком лікарів та доступних послуг;
4. Перегляд графіку роботи клініки;

5. Читання або додавання відгуку про відвідану клініку;
6. Фільтрація клінік за категоріями (вид тварини, послуга, рейтинг, розташування);

Моделювання за допомогою UML

Для формалізації логіки системи будуть використані наступні діаграми UML:

- Діаграма прецедентів (Use Case Diagram) — відображає основні функціональні сценарії, доступні для різних ролей користувачів;
- Діаграма активностей (Activity Diagram) — це графічне представлення послідовності дій або кроків, що виконуються в межах одного процесу або сценарію, яке дозволяє візуалізувати логіку роботи системи, розгалуження, паралельне виконання та завершення процесу.
- Діаграма послідовності (Sequence Diagram) — деталізує взаємодію між елементами системи у часі під час виконання окремої операції (наприклад, пошуку клініки з фільтрами).

Використання цих інструментів забезпечує логічну узгодженість структури, спрощує реалізацію бізнес-логіки, та полегшує тестування системи.

1.4 Діаграма прецедентів

Діаграма прецедентів є одним із ключових засобів функціонального моделювання в UML. Вона дозволяє візуалізувати взаємодію користувачів із системою через опис дій (прецедентів), які кожен тип користувача може виконувати. Такий підхід допомагає на ранньому етапі розробки виявити функціональні вимоги, окреслити межі системи та розподілити відповідальність між різними ролями.

У діаграмі прецедентів основними елементами є:

- **Актори** — зовнішні користувачі або системи, які ініціюють дії;
- **Прецеденти** — функції або сценарії використання, що реалізуються системою;
- **Зв’язки** — асоціації між акторами та прецедентами, які відображають доступні дії.

Побудова такої діаграми дозволяє структурувати логіку системи, забезпечити її зрозумілу візуалізацію та спростити комунікацію між розробниками, замовником і користувачами. У межах даної роботи діаграма прецедентів демонструє, як власник тварини та адміністратор сервісу взаємодіють із функціоналом системи для пошуку ветеринарних клінік, перегляду інформації та керування наповненням бази.

У системі задіяні два основні актори:

- Власник тварини, який взаємодіє із застосунком з метою пошуку клінік, перегляду послуг, лікарів, годин роботи тощо;
- Адміністратор сервісу, який відповідає за створення, оновлення та підтримку інформаційного наповнення системи.

Серед ключових функціональних можливостей системи:

- пошук ветеринарів і послуг за допомогою фільтрів;
- перегляд клінік на карті з використанням геолокації;
- семантичний пошук (через NLP-запити);
- доступ до описів клінік, лікарів та послуг;
- додавання і редагування інформації адміністраторами.

Діаграма прецедентів наведена на рисунку 1.

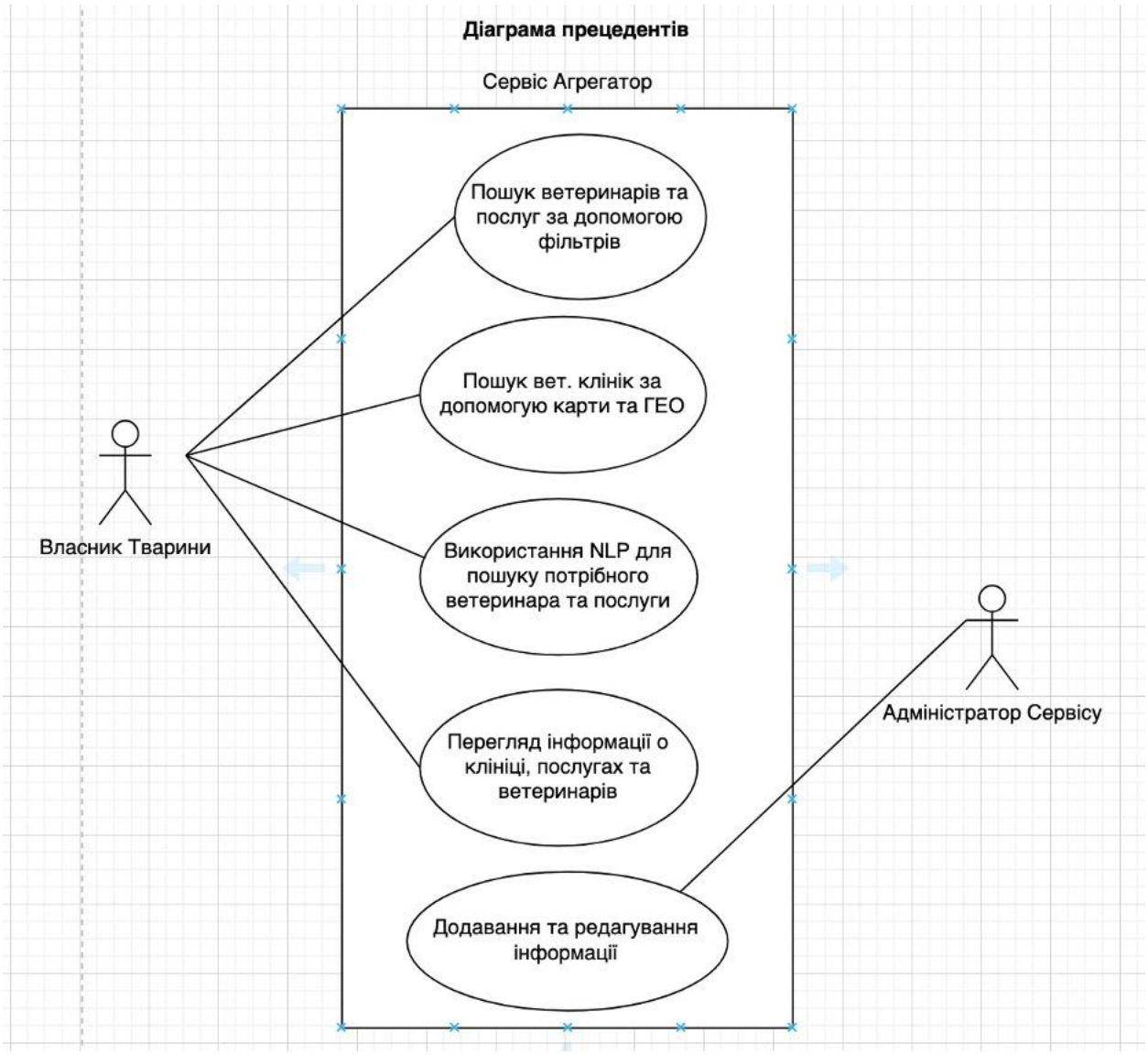

Рис. 1 Діаграма прецедентів

1.5. Діаграма активності

Діаграма активності — це тип діаграми в мові UML, що використовується для моделювання поведінки системи, зокрема — послідовності дій, які виконуються під час реалізації певного бізнес-процесу або сценарію використання. Вона дозволяє графічно зобразити логіку переходів між діями, умовні розгалуження, паралельні гілки та точки завершення процесу.

У рамках даної бакалаврської роботи побудовано діаграму активності, яка ілюструє типову послідовність дій користувача під час пошуку ветеринарної клініки в інформаційній системі. Цей сценарій є одним із ключових для взаємодії клієнта з сервісом, оскільки саме на основі пошуку користувач приймає рішення щодо вибору закладу та послуги.

Процес починається з отримання початкових даних — система визначає локацію користувача, запит або відкриває головний інтерфейс. Далі користувач обирає метод пошуку:

- Пошук за допомогою фільтрів — наприклад, за видом тварини, напрямом лікування, рейтингом клініки;
- Пошук за геолокацією — користувач бачить клініки поруч із собою на мапі;
- Семантичний пошук (NLP) — система інтерпретує запити у природній формі (наприклад, «клініка, що приймає черепах у неділю»).

Після вибору методу запит обробляється, результати виводяться на екран, користувач переглядає інформацію про клініки, лікарів і послуги. Процес завершується, коли користувач закриває інтерфейс або припиняє взаємодію з системою.

Вказана логіка представлена на рисунку 2.

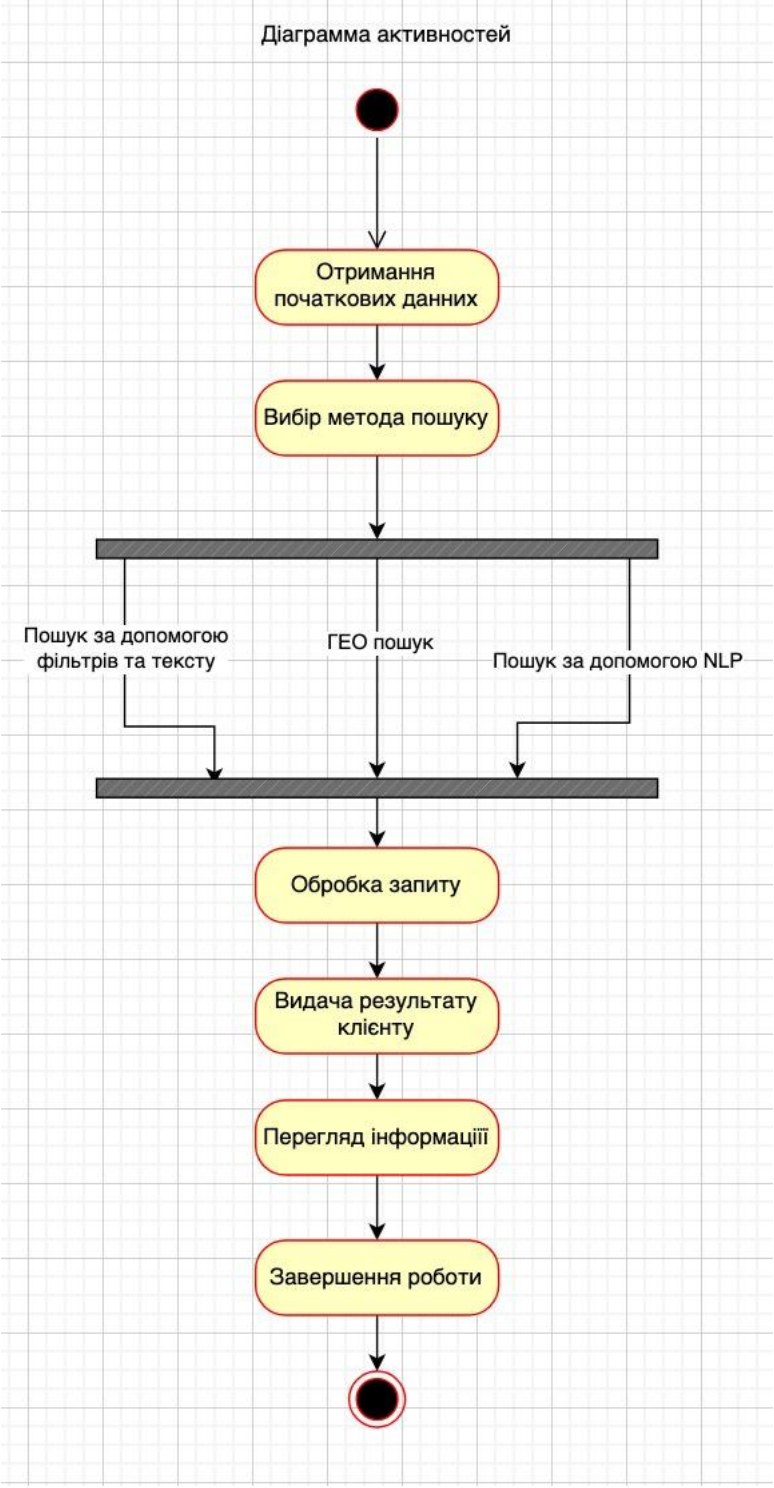

Рис. 2 Діаграма активності

Ця діаграма дозволяє не лише формалізувати поведінку програми, а й слугує орієнтиром для реалізації бізнес-логіки пошуку в мобільному застосунку.

1.6. Діаграма послідовності

Діаграма послідовності (Sequence Diagram) є важливим інструментом у моделюванні поведінки об’єктів в інформаційній системі. Вона демонструє обмін повідомленнями між учасниками системи в межах конкретного сценарію використання, враховуючи часову послідовність подій. Такий тип діаграми особливо корисний для деталізації логіки взаємодії між клієнтом і сервером та для виявлення залежностей між модулями системи.

У даній роботі побудовано діаграму послідовності для одного з найважливіших процесів — пошуку ветеринарної клініки, лікаря або послуги. Цей сценарій охоплює взаємодію між такими учасниками:

- Власник тварини — кінцевий користувач мобільного застосунку, який вводить запит;
- Сервіс-агрегатор — логіка, що виконує обробку запиту, звертається до бази даних, аналізує критерії та формує відповідь;
- Адміністратор сервісу — особа, яка раніше наповнила базу інформацією про клініки, лікарів, години роботи та послуги.

На діаграмі послідовності відображено кілька паралельних сценаріїв пошуку, які може ініціювати користувач:

1. Пошук за допомогою фільтрів — користувач обирає конкретні критерії (тип тварини, категорія послуг, мова обслуговування тощо), які передаються до сервісу-агрегатора, обробляються, і як результат система повертає список релевантних клінік або лікарів.

2. Пошук за допомогою NLP (Natural Language Processing) — користувач формулює запит у вільній текстовій формі (наприклад, «хто лікує морську свинку у вихідні»), який аналізується алгоритмами обробки природної мови, після чого система знаходить релевантні дані та повертає їх користувачу.
3. Пошук із використанням геолокації (GEO) — користувач обирає режим геопошуку, сервіс визначає його поточне розташування (або задану точку на карті) та знаходить клініки в радіусі дії. Дані також обробляються на стороні сервісу та повертаються у вигляді списку результатів.

Після кожного запиту користувач отримує відповідь, аналізує її та за потреби переглядає інформацію про вибрану клініку або лікаря (фото, опис, рейтинг, графік роботи, послуги тощо).

Додатково вказано участь адміністратора сервісу, який попередньо виконав запис відповідної інформації у систему — без цього жоден із типів пошуку не міг би відбутися коректно.

Ця діаграма демонструє не лише черговість операцій, але й логіку переходів між запитами та їх обробкою. Такий підхід дозволяє розробнику:

- краще зрозуміти потік даних у межах системи;
- виявити вузькі місця або дублювання логіки;
- проектувати ефективні API-ендпоінти;
- формалізувати тестові сценарії для backend.

Візуальне представлення цього процесу наведено на рисунку 3.

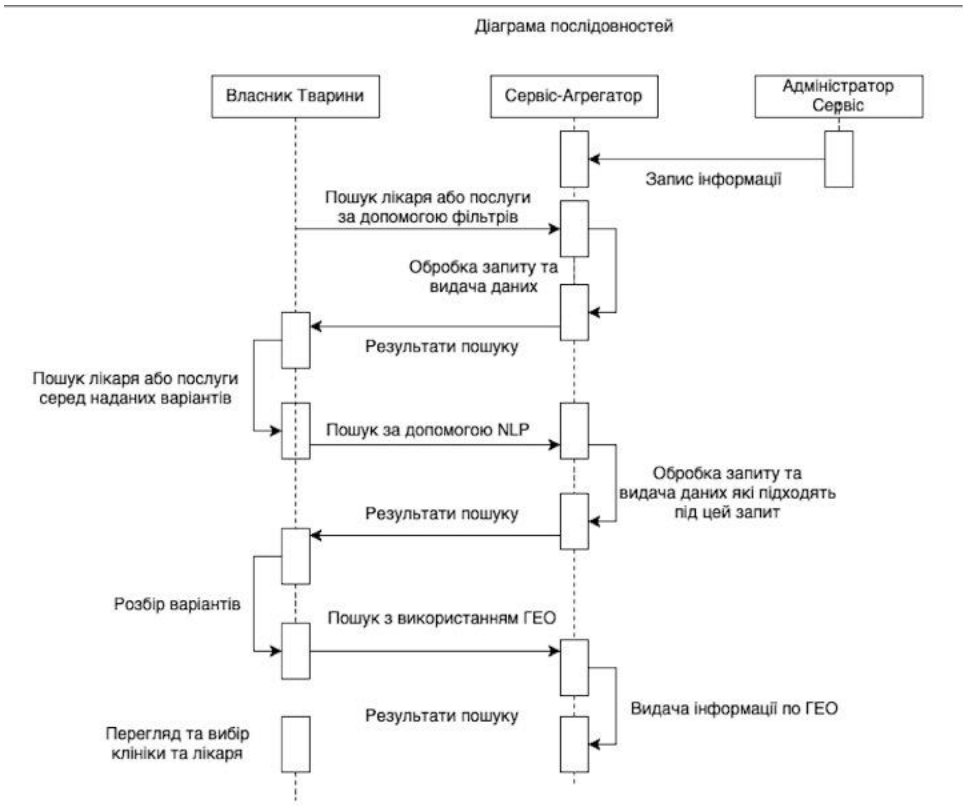

Рис. 3 Діаграма послідовності

2 ПРОЕКТУВАННЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ ДАНИХ

2.1 Загальні відомості про СУБД (системи управління базами даних)

Бази даних є ключовим компонентом будь-якої системи, оскільки забезпечують зберігання, структурування та доступ до даних. Вони дозволяють автоматизувати обробку великої кількості інформації, зберігати історію операцій, пов’язувати об’єкти між собою та організовувати швидкий пошук за заданими критеріями.

База даних — це сукупність даних, що зберігаються у структурованій формі для подальшого використання. Дані в базі організовуються за певною моделлю, яка визначає їх внутрішню структуру, типи полів, зв’язки між об’єктами та правила обробки.

Керування базою даних виконується за допомогою систем управління базами даних (СУБД) — програмного забезпечення, яке забезпечує взаємодію між користувачем, прикладною програмою та фізичним сховищем даних. СУБД виконує такі функції:

- додавання, редагування та видалення даних;
- збереження та захист інформації;
- підтримка цілісності й актуальності даних;
- формування запитів та звітів.

Бази даних застосовуються в усіх галузях, де виникає потреба в зберіганні великої кількості впорядкованої інформації: у фінансах, медицині, освіті, електронній комерції, логістиці, мобільних додатках тощо. Залежно від потреб системи та структури даних обираються різні типи баз — з чіткою або

гнучкою схемою, централізовані чи розподілені, табличні або документ орієнтовані.

У наступних підпунктах було розглянуто основні підходи до побудови баз даних, їх типи, особливості, а також обґрунтовано вибір конкретної СУБД для розробки даної інформаційної системи.

2.2 Класифікація баз даних: реляційні та нереляційні

У проектуванні сучасних інформаційних систем важливо правильно обрати тип бази даних, оскільки саме від цього залежить не лише продуктивність системи, а й простота її реалізації, адаптації до змін, масштабування та зручність у роботі з різними структурами даних.

Найбільш поширеними є два підходи до організації баз даних:
реляційна модель (SQL);
нереляційна модель (NoSQL).

Реляційні бази даних

Реляційна модель даних виникла ще у 1970-х роках і залишається актуальною дотепер. У таких базах дані організовано у вигляді таблиць, де кожен запис (рядок) представляє окрему сутність, а кожен стовпчик — атрибут.

Головні характеристики:

- Суворо визначена схема: типи даних, обов’язковість полів, ключі;
- Чіткі зв’язки між таблицями: реалізовані за допомогою первинних та зовнішніх ключів;
- Мова запитів SQL: стандартизована мова доступу до даних;
- Нормалізація: зменшення дублювання шляхом розділення сутностей на окремі таблиці.

Найпоширеніші реляційні СУБД:

- MySQL — відкрита й легка у використанні система, особливо популярна в малих і середніх проєктах.
- PostgreSQL — потужна об’єктно-реляційна система з підтримкою складних типів даних, тригерів і функцій.
- SQLite — вбудована легка СУБД, що не потребує окремого сервера, використовується в мобільних додатках і автономних системах.
- Microsoft SQL Server — комерційна система від Microsoft з широкими можливостями для підприємств.
- Oracle Database — одна з найпотужніших і найнадійніших СУБД, орієнтована на корпоративний сегмент і великі навантаження.

Сфера застосування реляційних БД охоплює:

- системи бухгалтерського обліку;
- банківські транзакції;
- управління персоналом і ресурсами;
- освітні електронні журнали та платформи.

Переваги:

- високий рівень структурованості;
- гарантована цілісність даних;
- підтримка транзакцій.

Недоліки:

- складність при роботі з вкладеними або гнучкими структурами;
- потреба в постійній синхронізації таблиць через JOIN-запити;
- низька гнучкість при зміні вимог до структури даних.

Нереляційні бази даних

На противагу класичним SQL-рішенням, нереляційні БД були створені для вирішення завдань, які погано лягають у табличну форму. Особливо це стосується:

- зберігання об’єктів зі змінною структурою;
- великого обсягу динамічних даних;
- високих вимог до масштабованості.

Нереляційні бази поділяються на чотири основні типи:

1. Документоорієнтовані — зберігають записи як документи формату JSON/BSON. Кожен документ — це самостійна структура зі своїми вкладеними об’єктами. Приклади: MongoDB, CouchDB.
2. Ключ–значення — найпростіша форма, де ключі пов’язані з єдиним значенням. Швидкий доступ, але відсутня складна структура. Приклади: Redis, Amazon DynamoDB.
3. Колонкові — орієнтовані на аналітику, де дані зчитаються стовпцями замість рядків, що дає перевагу при обробці великих обсягів. Приклади: Apache Cassandra, HBase.
4. Графові — зберігають сутності у вигляді вузлів і зв’язків, що ідеально підходить для побудови соціальних мереж, логістики або рекомендаційних систем. Приклади: Neo4j, ArangoDB.

Переваги NoSQL:

- гнучка структура документів — не вимагає єдиної схеми;
- легке масштабування (горизонтальне);
- швидкий доступ до вкладених об’єктів без JOIN-запитів;
- адаптивність до змін у структурі даних.

Недоліки NoSQL:

- відсутність жорсткої валідації типів на рівні БД;

- обмеженість транзакційної підтримки (хоча сучасні NoSQL-платформи це частково вирішили);
- менш зрозумілий підхід для традиційних проектів.

2.3 Обґрунтування вибору MongoDB для даної інформаційної системи

На основі аналізу загальних характеристик реляційних та нереляційних систем зберігання даних, мною було прийнято рішення використовувати саме MongoDB як основну систему керування базами даних для проекту системи пошуку ветеринарних клінік, лікарів і послуг.

MongoDB є документ орієнтованою базою даних, яка зберігає інформацію у форматі JSON-подібних документів (BSON). Це дозволяє представляти складні вкладені структури у вигляді одного документа, що значно спрощує обробку та доступ до даних у порівнянні з реляційними рішеннями.

Причини вибору MongoDB для даної системи:

1. Вкладена структура даних

У контексті даного проекту кожна ветеринарна клініка містить велику кількість пов’язаної інформації: опис, розташування, список лікарів, години роботи, перелік доступних послуг, відгуки користувачів тощо. MongoDB дозволяє зберігати всі ці дані в одному документі без необхідності створювати окремі таблиці або складні зовнішні зв’язки.

2. Гнучкість структури

Інформаційна система передбачає динамічну змінюваність структури даних — можуть додаватися нові категорії послуг, мови обслуговування, спеціалізації лікарів або додаткові атрибути клінік. MongoDB не потребує попереднього визначення суворої схеми, тому її структура легко адаптується

до нових умов без зміни всього, що дуже корисно для проектів які тільки починаються.

3. Швидкий доступ до повного запису

Завдяки зберіганню всієї інформації про клініку у межах одного документа, система може миттєво повертати повний об’єкт, не виконуючи кількох запитів або JOIN-операцій, як у випадку з реляційними базами. Це особливо важливо для сервісів агрегаторів, де критичними є швидкість і мінімізація звернень до серверу.

4. Масштабованість та розширюваність

MongoDB підтримує горизонтальне масштабування (через шардінг), що забезпечує стабільну роботу системи при збільшенні обсягів даних, кількості клінік або користувачів. Це дозволяє платформі зростати без потреби радикального перепроектування бази.

5. Швидка інтеграція з сучасними технологіями

MongoDB добре інтегрується з сучасними backend-фреймворками, зокрема з Golang, який використовується у даному проекті для реалізації API. Також формат даних (JSON/BSON) легко передається у мобільний клієнт, створений за допомогою Flutter, що значно спрощує обмін інформацією між клієнтською і серверною частинами.

6. Сховище картинок та мапа

MongoDB має вбудовану підтримку геолокаційних даних. Це дозволяє створювати запити типу «знайти найближчу клініку», «відсортувати за відстанню», «знайти в межах певного радіусу» тощо, а також завдяки особливому сховищу GridFS можна зберігати картинки що допоможе зберегти гроші на S3.

Прикладна користь у межах проекту:

- Кожна клініка зберігається як єдиний документ;

- Всередині документа є масиви: doctors, services, workingHours, reviews;
- Користувач одразу отримує всю необхідну інформацію для прийняття рішення без додаткових запитів;
- Адміністратор сервісу може оновлювати дані гнучко, без перенесення структури або модифікацій таблиць.

2.4. Структура даних та взаємозв’язки між об’єктами

Інформаційна модель системи розроблена з урахуванням вимог MongoDB до зберігання вкладених і пов’язаних структур. Основна ідея реалізації — групування пов’язаних сутностей у межах одного документа, що забезпечує швидкий доступ до повної інформації про клініку без необхідності виконання складних запитів.

Основними колекціями є clinics, doctors, services, categories та reviews. Їх структура та взаємозв’язки зображено на рисунку 4.

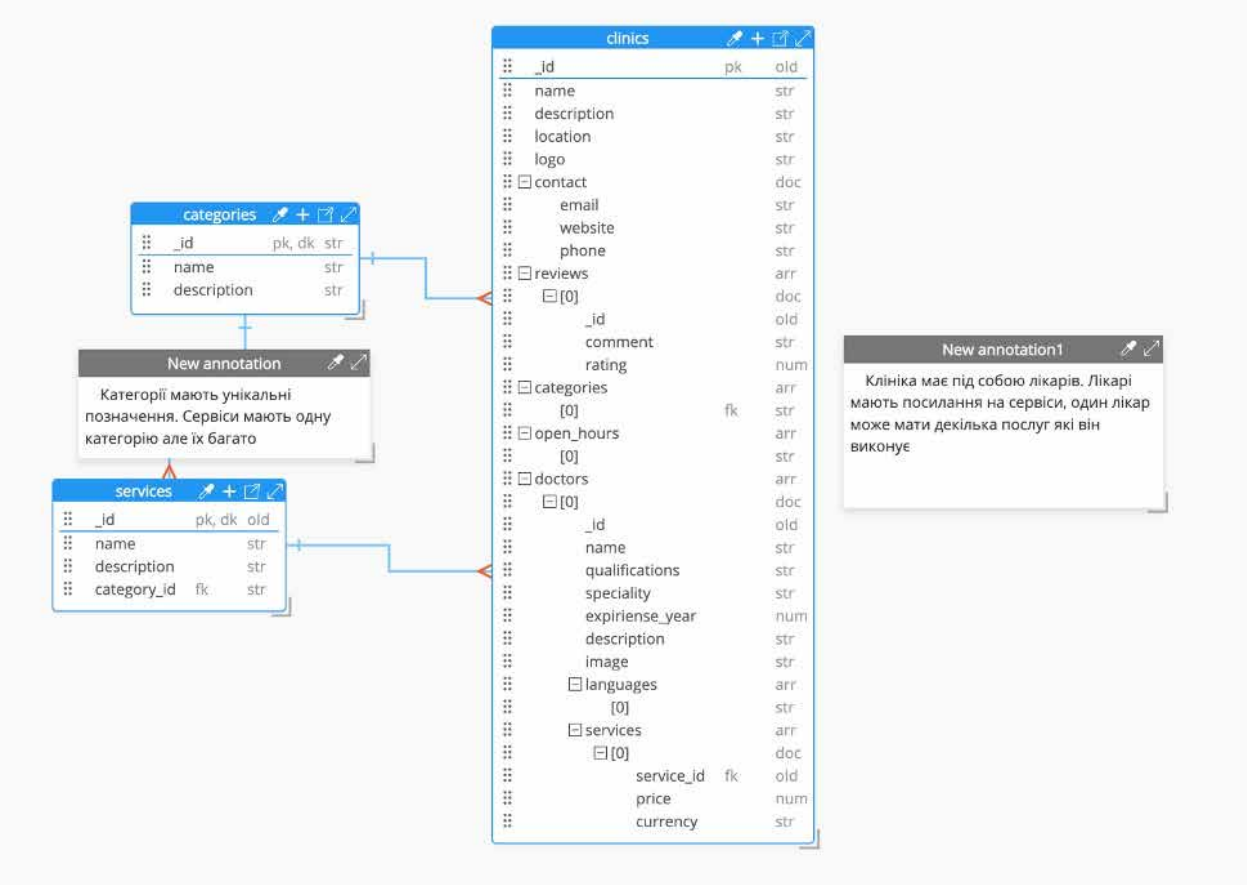

Рис. 4 Структура даних та зв’язки між об’єктами у MongoDB

Опис колекцій

- Clinics (клініки) — центральна колекція, що містить усю основну інформацію про заклад: назву, адресу, контакти, розклад роботи, перелік лікарів, послуг, рейтинг, відгуки користувачів, мови обслуговування та геолокацію. Вона також містить масиви doctors, open_hours, categories, reviews.
- Doctors (лікарі) — лікарі зберігаються як вкладені об’єкти всередині клінік. Для кожного лікаря вказується ПІБ, спеціалізація, кваліфікація, досвід, опис, мови, зображення, а також послуги, які він надає.

- Services (послуги) — опис послуг зберігається у вигляді вкладених або окремих документів, кожен із яких містить назву, опис, ціну та валюту. Послуги мають зв’язок із категоріями через зовнішній ключ category_id.
- Categories (категорії) — перелік загальних категорій послуг, таких як “терапія”, “вакцинація”, “грумінг” тощо. Категорії дозволяють групувати послуги для зручної фільтрації.
- Reviews (відгуки) — зберігаються як вкладені об’єкти у кожній клініці. Включають текст коментаря, оцінку, дату, ідентифікатор автора (у разі прив’язки до облікового запису користувача).

Взаємозв’язки між об’єктами

У структурі системи застосовано вкладену модель зберігання, де об’єкти нижчого рівня логічно включені в об’єкти вищого рівня. Зв’язки реалізовано таким чином:

- Клініка → Лікарі: лікарі зберігаються всередині документа клініки, оскільки жорстко прив’язані до неї.
- Лікар → Послуги: лікарі містять перелік послуг, які вони можуть виконувати.
- Послуга → Категорія: кожна послуга посилається на відповідну категорію, що дозволяє системі реалізовувати фільтрацію.
- Клініка → Відгуки: усі відгуки користувачів зберігаються у документі клініки, що дозволяє одразу отримати повну картину.

Переваги обраної структури

- Швидкість доступу: мінімізація кількості запитів для отримання повної інформації про клініку.
- Природне моделювання логіки взаємодії: зручність розробки і роботи з API.
- Гнучкість: можливість адаптації структури до змін функціоналу без необхідності змінювати схему всієї БД.

- Масштабованість: окремі колекції (наприклад, categories) можуть використовуватись глобально без дублів у кожній клініці.

Обрана модель повністю відповідає потребам мобільного агрегатора ветеринарних клінік, де основною одиницею є клініка, що містить повний набір пов’язаних даних.

2.5 Особливості моделювання у MongoDB

Моделювання даних у документоорієнтованих базах даних, зокрема MongoDB, має принципові відмінності від класичного підходу у реляційних СУБД. Якщо у SQL базах проектування починається з нормалізації, створення численних таблиць і зв’язків між ними, то у MongoDB логіка моделювання базується на агрегації пов’язаних об’єктів у межах одного документа.

MongoDB підтримує дві основні стратегії моделювання:

вкладене зберігання (embedding);

посилання (referencing).

Вкладене зберегання

Вкладеність застосовується тоді, коли пов’язані сутності:

- завжди використовуються разом;
- мають однозначну прив’язку до об’єкта вищого рівня;
- не потребують частої незалежної модифікації.

У контексті розроблюваної системи:

- лікарі є частиною клініки → зберігаються всередині документа clinics;
- послуги лікаря → зберігаються у полі лікаря;
- відгуки → зберігаються безпосередньо в клініці як масив об’єктів.

Перевага такого підходу — максимальна швидкодія читання: усі дані, необхідні для відображення картки клініки, повертаються в одному запиті без додаткових звернень до інших колекцій.

Посилання

Посилання (reference) використовується тоді, коли:

- об’єкти мають багато зв’язків або відносяться до кількох колекцій;
- потрібно оновлювати об’єкт окремо;
- один елемент пов’язаний із багатьма батьківськими (наприклад, один лікар працює в кількох клініках — у майбутньому).

У поточному рішенні для деяких об’єктів, наприклад services або categories, може бути використано зв’язок за ідентифікатором — тобто service_id, category_id, — що дозволяє не дублювати ці об’єкти в кожному документі.

Вибір підходу в системі

У даному проекті було обрано **комбіновану модель**, яка передбачає:

- **вкладення** там, де сутності не існують окремо (наприклад, лікар належить лише одній клініці);
- **посилання** для глобальних об’єктів (наприклад, категорії послуг використовуються в багатьох клініках).

Цей підхід:

- спрощує обслуговування;
- зменшує дублювання;
- дозволяє масштабувати систему без зміни основної структури.

2.6 Процес підготовки MongoDB

Для початку роботи необхідно створити базу даних та кластер, якій потім можна поєднати з сервером. Для цього необхідно:

1. Створити кластер, зображено на рис. 5
2. Створити БД, зображено на рис. 6
3. Створити колекцію, зображено на рис. 6
4. Створити документ з даними, зображено на рис. 7

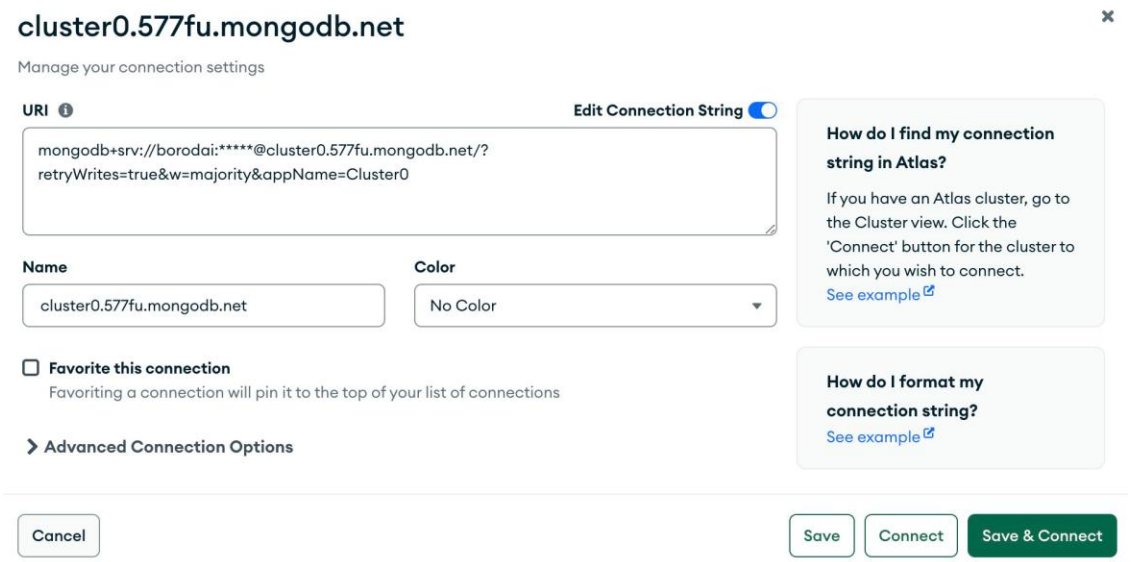

Рис. 5 Підключення до створеного кластера

Після цього етапу ми можемо підключитися до БД та створити необхідні колекції

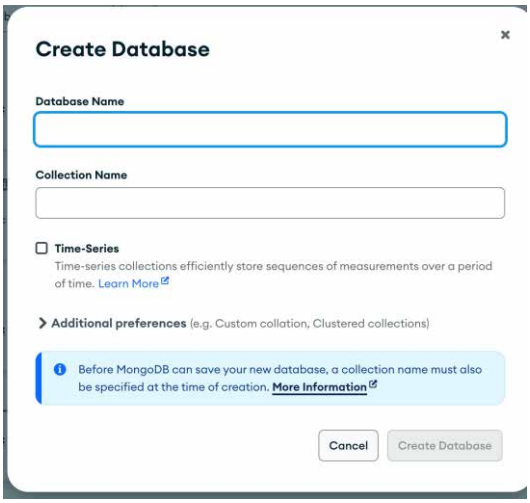

Рис. 6 Створення БД та першої колекції

На цьому етапі створена колекція по якій ми будемо групувати основні сутності такі як Категорії, Сервіси та Лікарні.

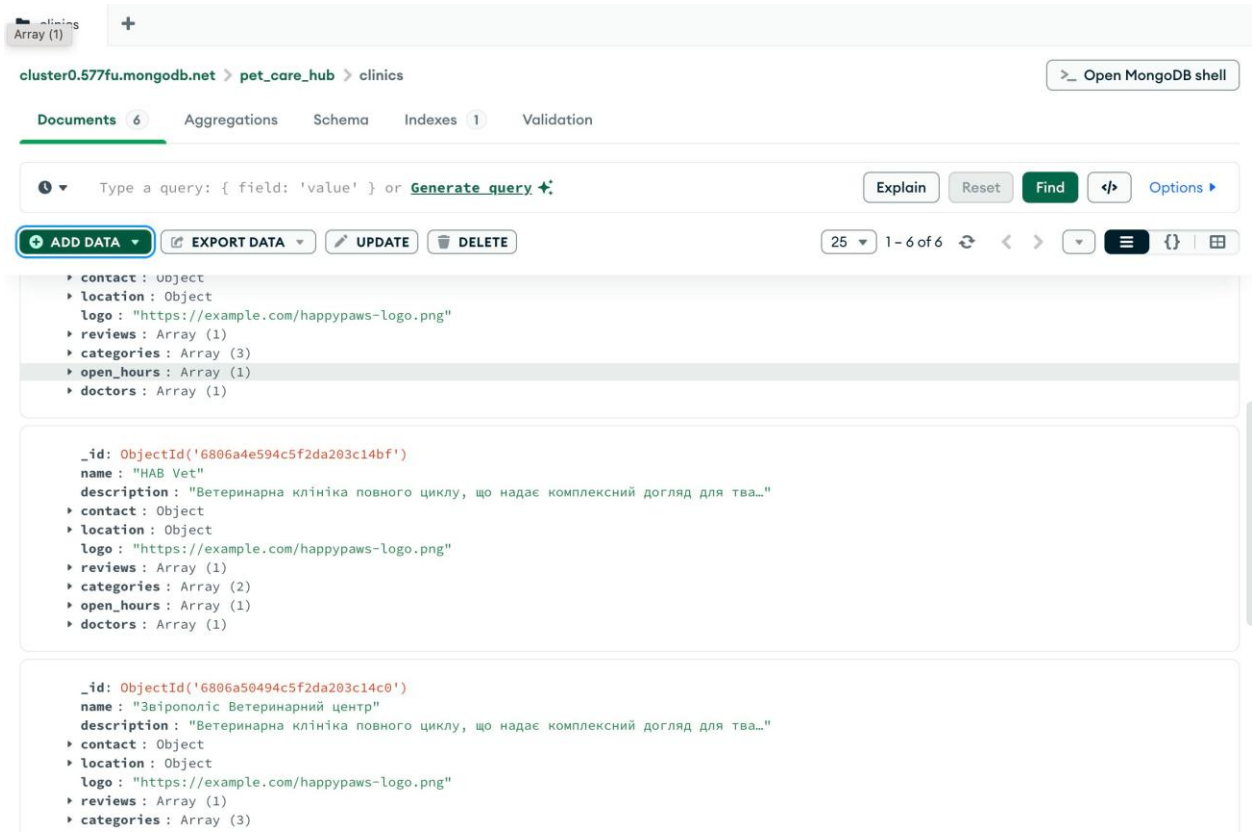

Рис. 7 Заповнена БД з декількома документами

Після заповнення маємо наступний документ з даними, на прикладі клінік

3 ПРИКЛАДНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ

3.1 Організаційна структура програмного забезпечення

Організація програмного забезпечення розроблюваної інформаційної системи побудована на принципах модульності, розподілу відповідальності та масштабованості. Такий підхід дає змогу забезпечити високу гнучкість системи, можливість подальшого розширення функціоналу, а також незалежну розробку та оновлення окремих компонентів.

У структурному плані система складається з трьох основних рівнів:

1. Клієнтський рівень (інтерфейс користувача)

Це мобільний застосунок, створений на кросплатформеній технології, який виконує функцію взаємодії з кінцевим користувачем. Користувач має змогу:

- виконувати пошук клінік за різними критеріями (локація, вид тварини, спеціалізація);
- переглядати детальну інформацію про лікарів, послуги, відгуки та графік роботи;
- застосовувати фільтри або здійснювати пошук природною мовою;
- працювати з мапою та списком одночасно.

Інтерфейс відповідає за формування запитів до серверної частини, відображення отриманих результатів та взаємодію з елементами, зручними для користувача.

2. Серверний рівень (бізнес-логіка та API)

Серверна частина системи виконує роль посередника між клієнтом і базою даних. Вона реалізує:

- прийом HTTP-запитів від мобільного клієнта;

- обробку параметрів пошуку, фільтрацію, сортування, семантичний розбір;
- генерацію відповідей у форматі JSON;
- контроль доступу до даних та валідацію введеної інформації.

Логіка сервера реалізована у вигляді RESTful API, що дозволяє легко масштабувати систему, інтегрувати її з іншими сервісами або зовнішніми модулями (наприклад, адміністративна панель для клінік).

3. Рівень зберігання даних (сховище)

База даних відповідає за надійне зберігання та організацію інформації, яка включає:

- ветеринарні клініки та їхні атрибути;
- лікарів із персональними профілями;
- послуги та їх категорії;
- графіки роботи, мови обслуговування;
- відгуки користувачів.

Особливістю організації є використання вкладених структур, що дозволяє уникнути складних зв’язків і повернути повну інформацію за один запит. Сховище не взаємодіє напряму з користувачем, що сприяє підвищенню безпеки.

Загальна логіка взаємодії

Уся взаємодія в системі побудована за принципом запит → обробка → відповідь:

1. Користувач формує запит через мобільний додаток;
2. Серверна частина аналізує параметри, звертається до бази даних;
3. Отримані дані обробляються, структуруються й повертаються клієнту.

Це дозволяє забезпечити:

- швидку відповідь при великій кількості запитів;

- розділення відповідальностей між компонентами;
- можливість замінювати або масштабувати окремі частини незалежно.

Переваги обраної організації:

- логічне відокремлення інтерфейсу, логіки та зберігання даних;
- зручність у підтримці, тестуванні та оновленні;
- ефективність при роботі з мобільними пристроями;
- готовність до масштабування в умовах зростання кількості користувачів або клінік.

3.2 Вибір інструментарію для створення прикладного програмного забезпечення

Для розробки прикладного програмного забезпечення інформаційної системи пошуку ветеринарних клінік було використано сучасний стек технологій, орієнтований на мобільну кросплатформену розробку, ефективну серверну логіку та гнучке зберігання даних.

Вибір інструментів здійснювався на основі таких критеріїв:

- підтримка мобільних платформ Android та iOS з мінімальними витратами;
- можливість створення масштабованого API з високою продуктивністю;
- зручність роботи з вкладеними структурами даних;
- підтримка відкритих стандартів і широке ком’юніті.

Клієнтська частина

Для створення мобільного інтерфейсу було обрано Flutter — фреймворк з відкритим кодом від Google, який дозволяє розробляти застосунки одразу для Android та iOS з єдиною кодовою базою. Це суттєво зменшує витрати часу на розробку та підтримку. У проекті Flutter використовується для реалізації інтерфейсу пошуку, фільтрації, перегляду даних, роботи з мапою та взаємодії з API.

Інструменти:

- Flutter SDK — платформа для компіляції коду;
- Dart — мова програмування, що використовується у Flutter;
- Google Maps SDK — для візуалізації клінік на карті;
- HTTP / Dio — для відправки запитів до API;
- Provider / Riverpod — для керування станом інтерфейсу.

Серверна частина

Серверна логіка реалізована на мові програмування Golang (Go), яка вирізняється високою продуктивністю, вбудованою підтримкою багатопоточності, зручною структурою для побудови REST API. У проекті сервер обробляє пошукові запити, фільтрує результати, працює з геоданими, реалізує логіку обробки відгуків та послуг.

Інструменти:

- Go 1.x — основна мова розробки;
- Gin — легкий веб-фреймворк для створення RESTful API;
- Mongo Driver — офіційний драйвер для взаємодії з MongoDB;
- Go Modules — для управління залежностями;
- Postman / Swagger — для тестування та документації API.

База даних

Для зберігання інформації про клініки, лікарів, послуги, години роботи, відгуки використовується MongoDB. Завдяки документоорієнтованій структурі, MongoDB дозволяє зберігати пов’язані об’єкти у вигляді вкладених елементів, що значно спрощує обробку даних у мобільному застосунку. Структура бази забезпечує ефективний доступ до повної інформації про об’єкт за один запит.

Інструменти:

- MongoDB Atlas — хмарне розгортання;

- Mongo Compass — для візуалізації структури та запитів;
- Mongosh — командна оболонка для адміністрування.

Обраний інструментарій забезпечує відповідність цільовому призначенню системи, дозволяє реалізувати високопродуктивну, зручну у використанні та технічно гнучку інформаційну систему з можливістю подальшого масштабування і вдосконалення.

3.3 Реалізація та програмування функціональних модулів

Процес реалізації програмного забезпечення передбачав створення окремих функціональних модулів для кожного з логічних компонентів системи. Завдяки попередньо розробленій архітектурі, система була реалізована за принципами модульності, ізольованості відповідальності та повторного використання коду.

Функціональні модулі розроблено як на стороні клієнта (мобільний застосунок), так і на стороні сервера (REST API), із забезпеченням повноцінної взаємодії з базою даних.

1. *Модулі клієнтської частини (Flutter-додаток)*

Клієнтська частина складається з набору UI-екранів і логічних компонентів, кожен з яких відповідає за певний аспект користувацької взаємодії:

- Екран пошуку клінік

  Реалізовано форму фільтрації з можливістю вибору типу тварини, категорії послуг, рейтингу, розташування та мови обслуговування.

- Інтерактивна мапа

  Візуалізація клінік на карті за допомогою Google Maps. Користувач може переглядати заклади поблизу та взаємодіяти з мітками.

- Сторінка клініки

Відображає повну інформацію про заклад: опис, список лікарів, години роботи, відгуки, доступні послуги. Дані отримуються одним запитом до API.

- Обробка запитів до API

  Реалізовано модулі взаємодії з сервером через HTTP-клієнт. Дані отримуються у форматі JSON і розбираються в об’єкти Dart.

2. *Модулі серверної частини (Go API)*

Серверна частина побудована як набір REST-ендпоінтів, кожен з яких відповідає за певний бізнес-процес. Основні реалізовані модулі:

- Обробка запитів клінік

  API приймає запити з параметрами фільтрації, проводить пошук у базі даних, формує список відповідних клінік та повертає дані у форматі JSON.

- Семантичний пошук за ключовими словами (NLP)

  Реалізовано базову обробку запитів у текстовому форматі з розпізнаванням ключових термінів (наприклад, “лікар-хірург для собаки”) для пошуку релевантної послуги або лікаря.

- Вивід профілю клініки

  Сервер повертає повний документ клініки з усіма вкладеними об’єктами (лікарі, послуги, графік, відгуки) за її ідентифікатором.

- Обробка геолокаційного пошуку

  Реалізовано пошук найближчих клінік за координатами користувача з фільтрацією по радіусу.

3. *Логіка взаємодії з базою даних*

Кожен серверний модуль взаємодіє з базою даних через окремий репозиторій, який реалізує логіку доступу до MongoDB. Для більшості запитів

використовуються агреговані фільтри, що дозволяє повертати необхідну інформацію без потреби в додаткових обробках на клієнті.

Схема взаємодії виглядає наступним чином:

клієнт → API → репозиторій → MongoDB → репозиторій → API → клієнт.

Таке структурування забезпечує чіткий поділ відповідальності між модулями, спрощує тестування й підтримку коду, а також дозволяє гнучко розширювати функціонал у майбутньому.

4 ВПРОВАДЖЕННЯ, ТЕСТУВАННЯ ТА ОЦІНКА ІНФОРМАЦІЙНОЇ СИСТЕМИ

4.1 Апаратні та програмні вимоги

Перед впровадженням інформаційної системи важливо визначити, за яких умов вона може функціонувати стабільно, ефективно та без збоїв. Вимоги стосуються трьох основних компонентів: мобільного клієнта, серверної частини та бази даних. Усі компоненти розгортаються на різних пристроях і мають власні потреби в ресурсах.

4.1.1 Вимоги до клієнтської частини (мобільного застосунку)

Мобільний застосунок створений з використанням фреймворку Flutter і підтримує Android та iOS. Для комфортної роботи системи рекомендується наступна конфігурація пристрою:

Апаратні вимоги:

- Смартфон або планшет із процесором ARM або ARM64;
- Оперативна пам’ять: не менше 2 ГБ;
- Вільне місце на пристрої: від 100 МБ;
- Доступ до інтернету (Wi-Fi або мобільний зв’язок).

Програмні вимоги:

- Операційна система Android 5.0+ або iOS 12.0+;
- Доступ до Google Play / App Store (у разі публічного розповсюдження);
- Підтримка GPS (для використання пошуку за геолокацією);
- Дозволи: доступ до місцезнаходження, інтернету.

4.1.2 Вимоги до серверної частини (API та логіка)

Серверний застосунок реалізовано на мові Go. Для його роботи необхідно мати фізичний або віртуальний сервер із такими характеристиками:

Апаратні вимоги:

- CPU: не менше 2 ядер;
- RAM: не менше 4 ГБ;
- Доступ до мережі (інтернет або локальна).

Програмні вимоги:

- Операційна система: Linux (Ubuntu 20.04 або новіша) або Windows Server;
- Go 1.20 або вище;
- Git (для клонування репозиторію);
- Наявність SSL-сертифіката (для захищеного з’єднання).

Компоненти:

- Серверна логіка розгортається як окремий HTTP-сервіс;
- Всі маршрути API організовані за принципами REST;
- Дані обробляються у форматі JSON.

4.1.3 Вимоги до бази даних MongoDB

MongoDB використовується як основне сховище даних для всієї системи. Можна використовувати як хмарне розгортання (MongoDB Atlas), так і локальне розміщення.

Апаратні вимоги:

- Дисковий простір: від 20 ГБ;
- RAM: від 2 ГБ (мінімум), 4 ГБ — рекомендовано;
- Підключення до мережі (в разі віддаленого доступу).

Програмні вимоги:

- MongoDB версії 6.0 або новішої;
- Mongo Compass або mongosh (для адміністрування);
- Забезпечення резервного копіювання (опціонально — через cron).

4.1.4 Вимоги до середовища розробки

Для локальної розробки застосунку використовуються такі інструменти, зазначені у таблиці 3:

Таблиця 3

Компонент	Інструмент
Мова для сервера	Go 1.20+
Менеджер залежностей	Go Modules
IDE	Visual Studio Code або GoLand
Мова інтерфейсу	Dart 3.x
Flutter SDK	Flutter 3.10 або новіший
Android Emulator	Android Studio / VS Code
MongoDB GUI	MongoDB Compass

4.2 Установка та запуск системи

Розгортання інформаційної системи включає запуск усіх її основних компонентів: мобільного клієнта, серверної частини та бази даних. Для локального запуску передбачається використання персонального комп’ютера або ноутбука як середовища розробки й тестування. У промисловому середовищі кожен компонент може бути розгорнутий окремо — наприклад, сервер на хмарному VPS, база даних — у MongoDB Atlas, а клієнт — у вигляді готового пакету на смартфоні користувача.

4.2.1 Підготовка середовища

Перед запуском системи необхідно встановити наступні компоненти:

- Flutter SDK (версія 3.10 або новіша): для збірки та запуску мобільного застосунку;
- Dart SDK (входить у Flutter): мова програмування для клієнтської логіки;

- Go (Golang) (версія 1.20 або новіша): для запуску серверної частини;
- MongoDB (локально або через MongoDB Atlas): для зберігання даних;
- Postman або аналог: для тестування API;
- MongoDB Compass: для перегляду структури бази даних (опціонально);
- Android Emulator або фізичний смартфон для тестування клієнта.

4.2.2 Установка та запуск мобільного клієнта

1. Завантажити й встановити Flutter SDK з офіційного сайту: https://flutter.dev.
2. Додати Flutter до системного PATH.
3. Перевірити наявність пристрою за допомогою команди:
   flutter doctor
4. Клонувати або відкрити проєкт мобільного застосунку:
5. git clone https://github.com/kkomilfo/pet_care_hub
   cd vet-clinics-app
6. Встановити всі залежності:
   flutter pub get
7. Запустити застосунок на емульованому або підключеному фізичному пристрої:
   flutter run

Примітка: Для повноцінної роботи додатку необхідно, щоб API-сервер працював і був доступний за вказаною у змінних середовища адресою.

4.2.3 Установка та запуск серверної частини

1. Перейти до каталогу з серверним застосунком:
   cd server
2. Створити .env або .env.local файл зі змінними середовища:
   Приклад:

PORT=8080

DB_URI=mongodb://localhost:27017

DB_NAME=vet_clinics

3. Запустити сервер:

   go run main.go

   або (якщо використовується збірка):

   go build

   ./server

4. Перевірити доступність API, наприклад:

   http://localhost:8080/api/clinics

4.2.4 Налаштування MongoDB

MongoDB може бути встановлена локально або використана через хмарну платформу MongoDB Atlas. У обох випадках необхідно:

- Створити базу даних із назвою pet_care_hub;
- Створити колекції: clinics, categories, reviews (у разі потреби);
- Забезпечити відповідний доступ до URI для підключення з сервера.

Приклад URI для локальної бази:

mongodb://localhost:27017

Приклад URI для MongoDB Atlas:

mongodb+srv://user:password@cluster.mongodb.net/vet_clinics?retryWrites=true&w=majority

4.2.5 Налаштування клієнта для взаємодії з сервером

У коді Flutter-застосунку зазвичай використовується змінна або файл конфігурації, у якому зберігається базовий URI сервера. Необхідно змінити значення на фактичну адресу API, наприклад:

const String apiUrl = "http://localhost:8080/api"; // або IP сервера

У разі використання емулятора Android Studio з localhost, потрібно використовувати:

http://10.0.2.2:8080

4.2.6. Перевірка взаємодії компонентів

1. Запустити сервер і перевірити його відповідь через браузер або Postman.
2. Запустити мобільний застосунок.
3. Перейти на екран пошуку клінік і перевірити, чи завантажуються дані з API.
4. Ввести тестовий відгук, перевірити його появу в базі через Compass або запит.

   Якщо все працює — система готова до локального або хмарного розгортання.

4.3 Діаграма розгортання

Для наочного представлення фізичного розміщення компонентів інформаційної системи використано діаграму розгортання (deployment diagram), яка демонструє зв’язки між клієнтськими пристроями, сервером API та базою даних.

Ця діаграма належить до стандартів моделювання UML і дозволяє візуально відобразити, які програмні модулі розміщуються на яких апаратних вузлах, а також, як між ними здійснюється взаємодія.

На рисунку 8 представлено розгортання системи в реальному середовищі.

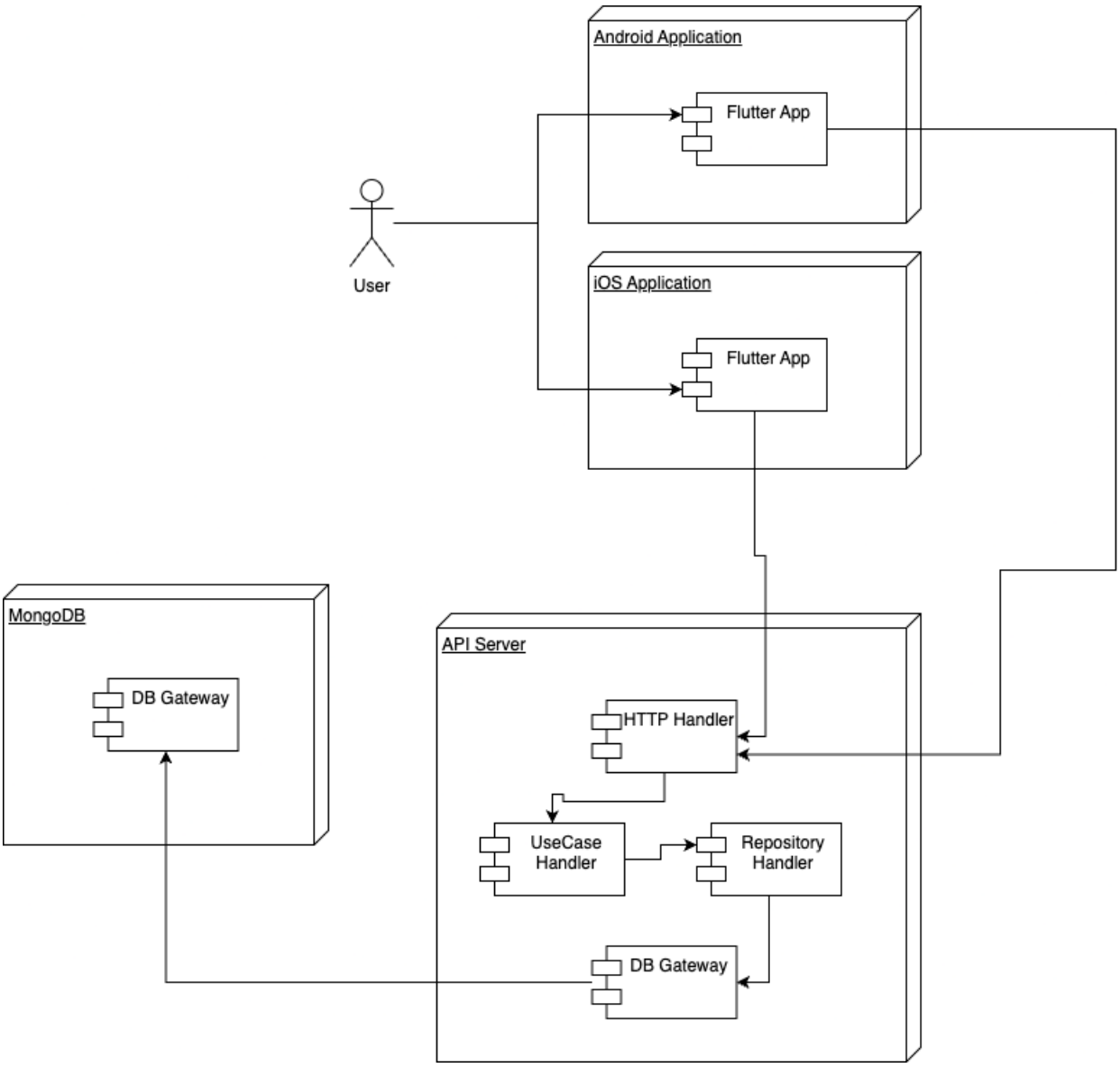

Рис. 8 Діаграма розгортання

Опис компонентів діаграми:

- Користувач (Client) — фізичний пристрій користувача (смартфон або планшет), на якому встановлений мобільний застосунок, створений з використанням Flutter. Клієнтська частина виконує роль інтерфейсу між користувачем та всією логікою системи.
- Flutter Application — мобільний застосунок, який надсилає HTTP-запити до API, приймає відповіді у форматі JSON, відображає інформацію про

клініки, лікарів, послуги та відгуки. Запускається на Android/iOS-пристроях.

- API Server (Go) — серверна частина, розміщена на окремому сервері або у хмарному середовищі. Вона відповідає за обробку логіки пошуку, валідацію запитів, підключення до бази даних і формування відповідей.
- База даних MongoDB — окрема система зберігання, яка може бути реалізована як локально (через встановлений інстанс MongoDB), так і у хмарі (через MongoDB Atlas). Зберігає всі дані про клініки, лікарів, послуги, категорії, години роботи, відгуки та користувачів.
- Комунікація між компонентами відбувається за допомогою:
    - Протоколу HTTPS між клієнтом і сервером;
    - Прямого підключення до бази даних зі сторони API за MongoDB URI.

Особливості реалізації:

- Вся логіка сервера і бази повністю ізольована від клієнта, що забезпечує підвищену безпеку системи;
- Мобільний застосунок не звертається до бази даних напряму — лише через API;
- Сервер реалізує архітектурний підхід “UseCase-Repository”, де кожна частина відповідає за окрему логіку: отримання даних, взаємодію з БД, обробку помилок;
- Така структура дозволяє легко масштабувати систему — розміщуючи базу, API та мобільні клієнти на різних вузлах або в хмарі.

4.4 Взаємодія з користувачем та сценарії використання системи

Ключовою особливістю інформаційної системи є її орієнтованість на кінцевого користувача — власника тварини, який шукає ветеринарну клініку,

послугу або конкретного лікаря. Важливо, щоб інтерфейс був інтуїтивно зрозумілим, а логіка взаємодії — послідовною, простою та швидкою.

Ролі користувачів у системі

У поточній версії реалізовано основна роль:

- Гостьовий користувач (власник тварини) — може переглядати інформацію про клініки, фільтрувати за критеріями, читати або залишати відгуки;

Алгоритм типового використання системи

Користувач запускає мобільний застосунок → переходить на головний екран, де має змогу взаємодіяти з системою за різними сценаріями приклад яких наведено у рисунках 9 – 14.

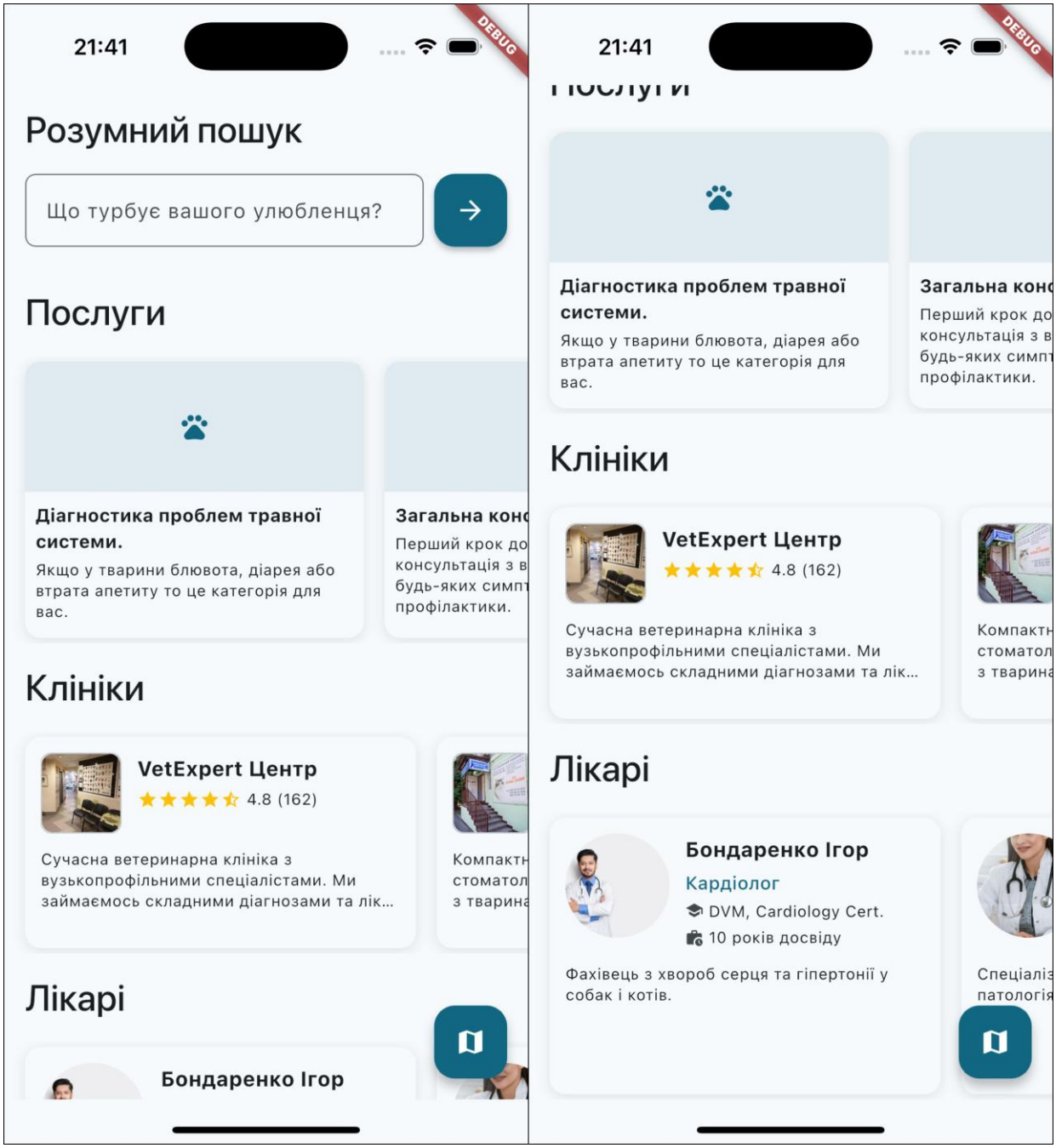

Рис. 9 Головний екран додатку з інформацію про категорії, лікарні та лікарі

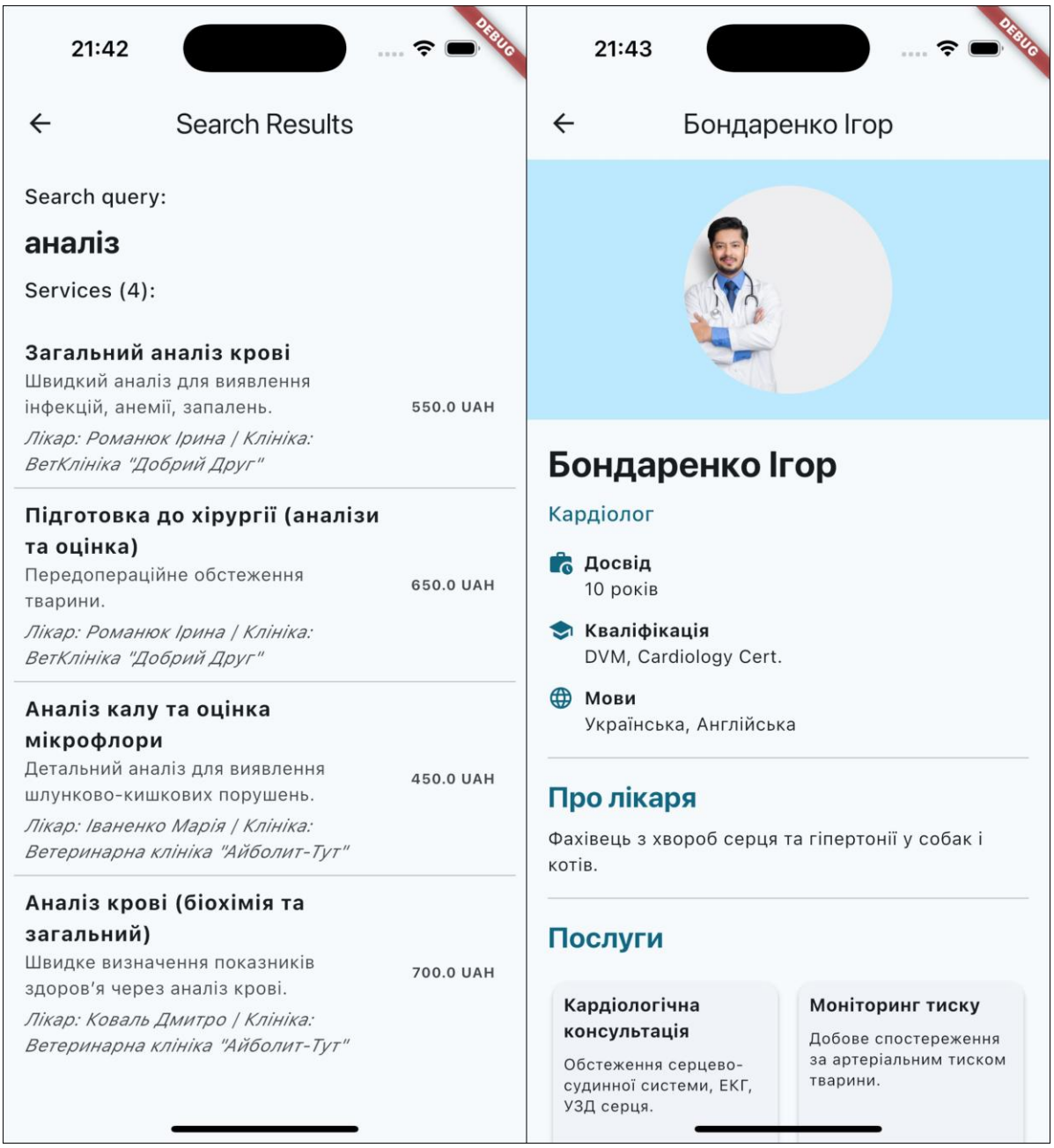

Рис. 10 Використання NLP пошуку, де за запитом надається послуги лікарів, і можна одразу перейти на його сторінку

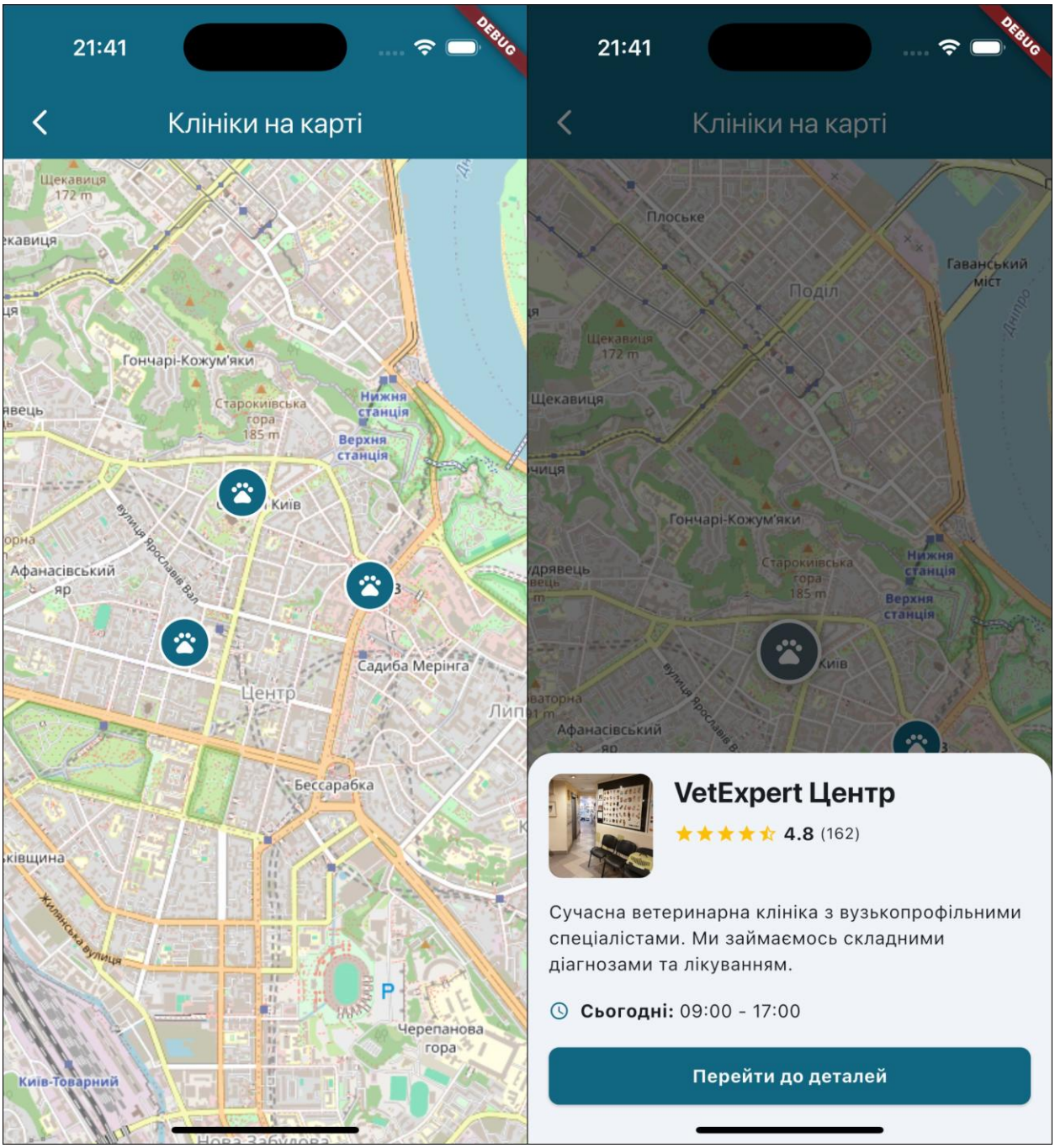

Рис. 11 Взаємодія користувача з мапою, де він зможе побачити клініки поруч нього, і детальні інформацію про неї

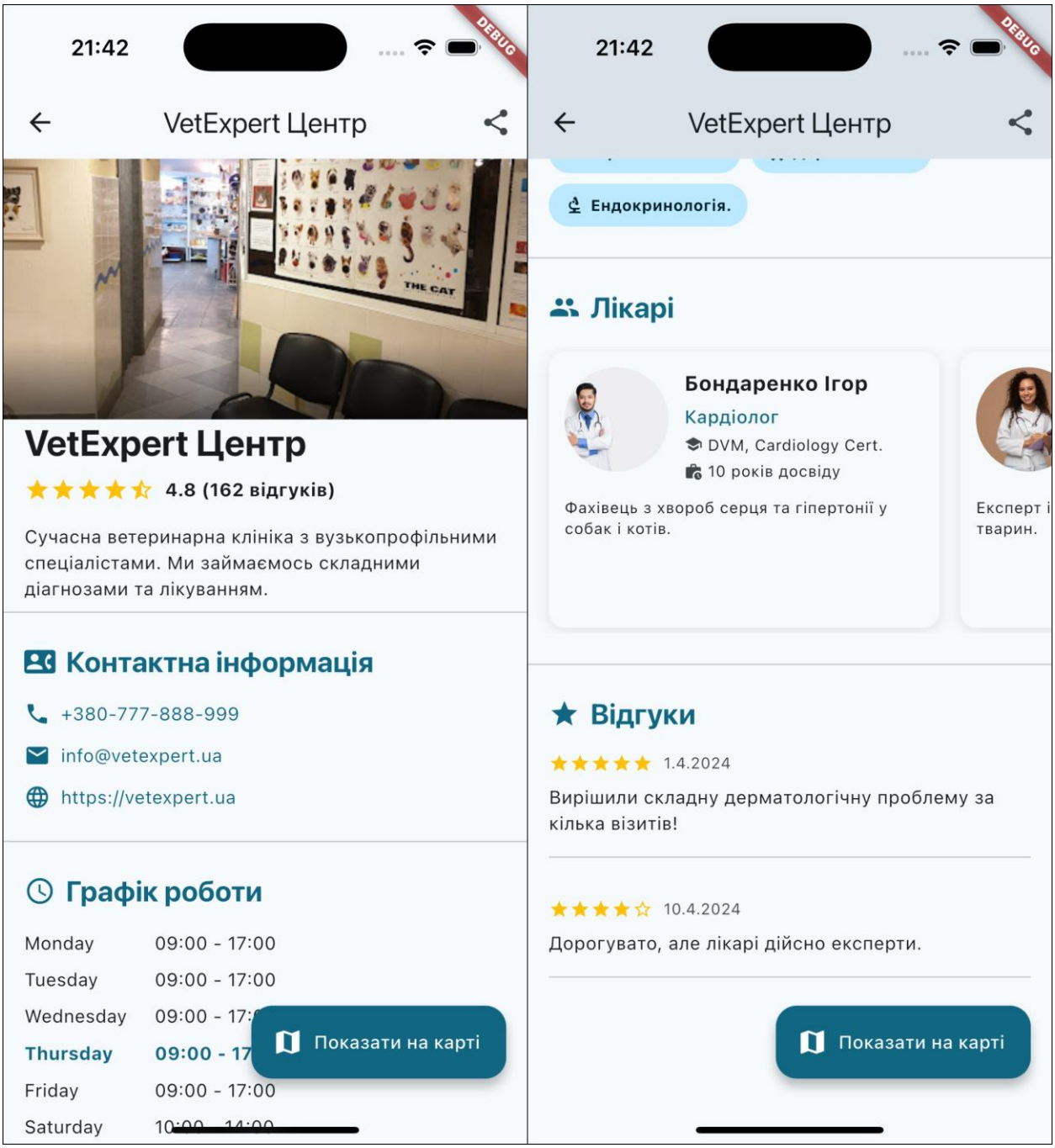

Рис. 12 Детальній екран клініки

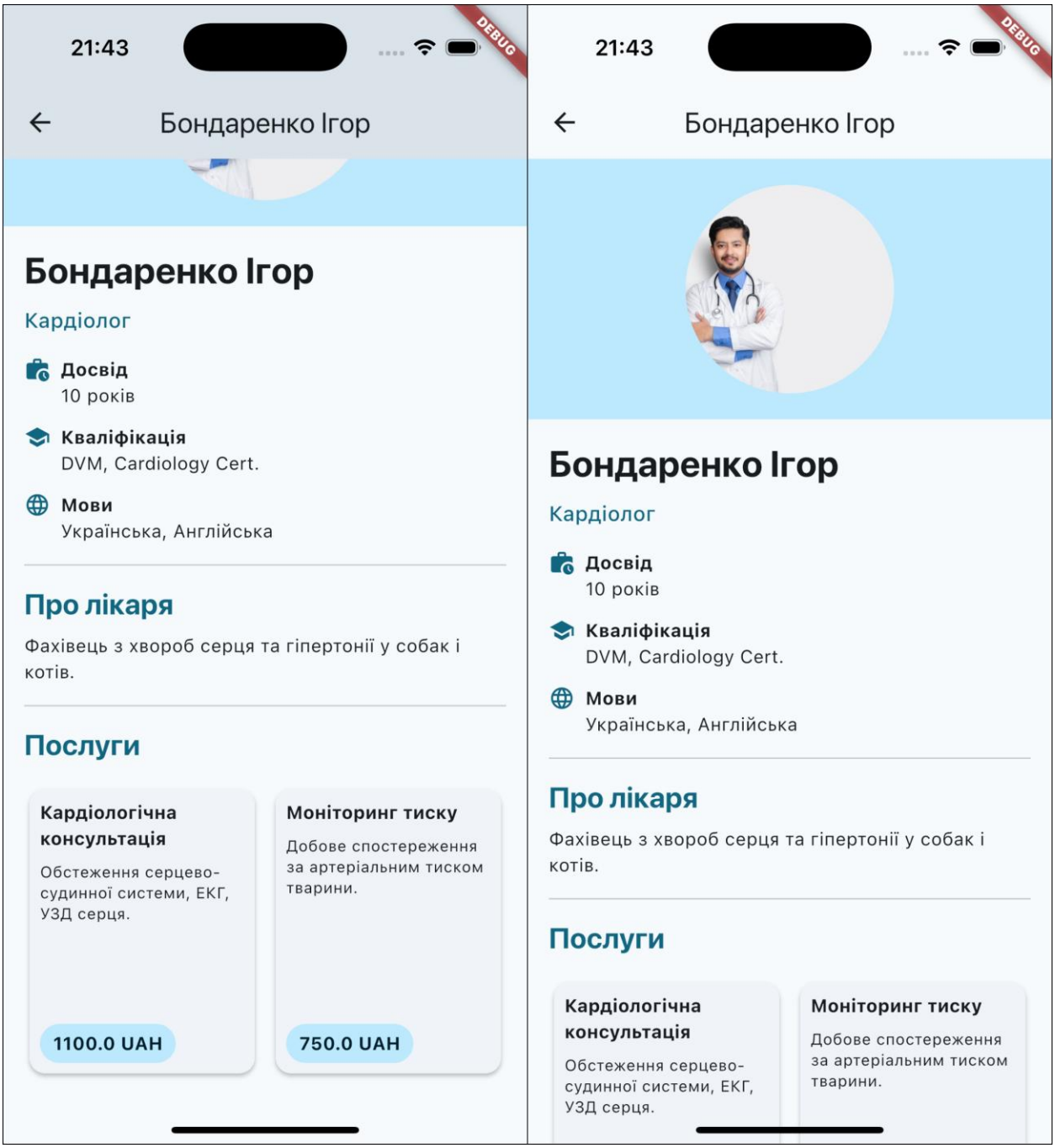

Рис. 13 Детальній екран лікаря

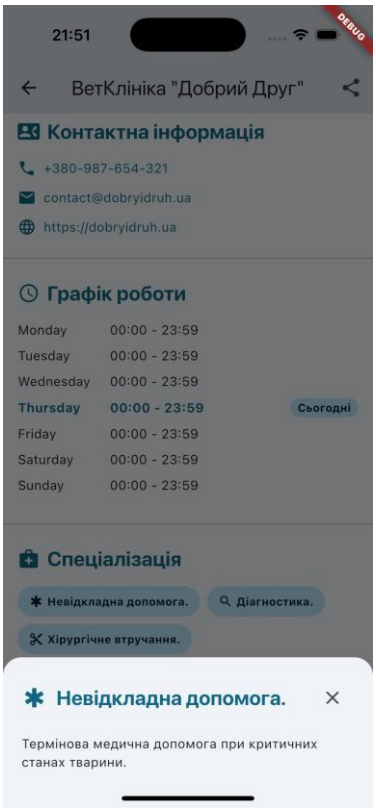

Рис. 14 Додаткова дія перегляду інформацію о категоріях яку несе клініка

4.5. Тестування системи

Тестування є важливою частиною життєвого циклу продукту, що дозволяє перевірити правильністьреалізації, стабільність роботи, а також відповідність вимогам, визначеним на етапах моделювання та проєктування. У рамках цієї роботи було проведено ручне функціональне тестування як клієнтської, так і серверної частини системи.

Тестування проводилось у локальному середовищі розробника з використанням таких інструментів:

- Postman — для перевірки REST-запитів до серверного API;
- MongoDB Compass — для перегляду змін у базі даних;
- Android Emulator і iOS Simulator — для тестування мобільного клієнта;
- Flutter DevTools — для відлагодження клієнтської частини;

Тестування охоплювало базові функції системи: пошук клінік, перегляд інформації, фільтрацію, додавання відгуків, роботу з геолокацією.

Сценарне тестування

Крім тестування окремих запитів, було проведено перевірку цілісних сценаріїв взаємодії користувача із системою:

- Сценарій: Пошук клініки за параметрами

  Користувач обирає вид тварини, категорію послуг і мову → запускається пошук → відображаються клініки, що відповідають критеріям.

- Сценарій: Перегляд клініки

  Користувач переглядає опис, список лікарів, послуги, години роботи → відкриває відгуки → переходить назад до мапи.

4.6 Оцінка ефективності роботи системи

Після розгортання та тестування інформаційної системи пошуку ветеринарних клінік було проведено оцінювання її ефективності. Воно охоплює кілька основних критеріїв: продуктивність, надійність, масштабованість, зручність використання та гнучкість до змін.

Продуктивність серверної частини

Серверна частина системи реалізована на мові Go, що дозволило досягти високої швидкодії завдяки вбудованій підтримці багатопоточності, легкості виконання та мінімальному споживанню ресурсів.

Під час тестування було отримано такі показники часу відповіді, надано у вигляді таблиці 4:

Таблиця 4

Запит	Середній час відповіді
Отримання списку клінік	200–250 мс
Пошук за геолокацією	300–420 мс
Отримання детального профілю	190–240 мс

Такі результати свідчать про стабільну роботу API-сервера в умовах локального навантаження. При хмарному розгортанні за допомогою CDN або балансування очікується ще вища продуктивність.

Продуктивність бази даних

MongoDB забезпечила швидкий доступ до вкладених об’єктів завдяки гнучкій схемі даних. Операції читання виконуються миттєво завдяки локальному індексуванню полів, таких як category, location, rating.

Переваги:

- Повернення повного документа клініки в одному запиті без JOIN;
- Висока ефективність читання за умов вкладеності;
- Підтримка геопросторових запитів ($geoWithin, $near).

Середній час обробки одного запиту до MongoDB (локально): 35–65 мс.

Масштабованість

Система проектувалась як масштабована завдяки поділу на незалежні модулі:

- Клієнт може працювати автономно з будь-яким сервером через конфігурацію URI;
- API-сервер легко запускається в кількох інстансах у хмарі;
- MongoDB може бути розгорнута в кластері зі шардінгом та реплікацією;
- Інтерфейс не залежить від логіки — це дає змогу незалежно оновлювати UI без змін у бекенді.

У майбутньому можливе горизонтальне масштабування з використанням балансування навантаження на API (наприклад, через Nginx або Kubernetes).

Зручність та UX – ефективність

Інтерфейс мобільного додатку створено на Flutter з урахуванням принципів адаптивного дизайну:

- логічна навігація;
- контрастні кольори для зручності читання;
- підтримка світлої і темної теми;
- відображення карт і списків у зручному вигляді.

Результати внутрішнього UX-тестування:

- середній час до першого пошуку — менше 10 секунд;
- кількість дій для отримання профілю клініки — 2–3;
- задоволеність інтерфейсом (за 5-бальною шкалою): 4.8.

Гнучкість та адаптивність системи

Завдяки використанню NoSQL-структури даних MongoDB, система дозволяє:

- Додавати нові поля в документи без міграцій;
- Адаптувати схему без порушення цілісності;
- Реагувати на зміну вимог без потреби зміни логіки на клієнті.

Це особливо важливо при масштабуванні проекту: наприклад, додавання нових категорій послуг або підтримки нових мов не потребує зміни структури таблиць, як у реляційних БД.

4.7 Безпека та обмеження доступу

Інформаційна система, що реалізується, передбачає обробку даних, які не є критично конфіденційними, однак для забезпечення стабільності, цілісності

та безпечного використання системи необхідно враховувати базові принципи інформаційної безпеки. На поточному етапі реалізовано базовий рівень безпеки, з можливістю подальшого розширення до повноцінної моделі авторизації та автентифікації користувачів.

Обмеження прямого доступу до бази даних

MongoDB розгорнута у захищеному середовищі через MongoDB Atlas і не має прямого публічного доступу з клієнтського боку. Всі запити до бази здійснюються виключно через API, що реалізує серверна частина.

Цей підхід дозволяє:

- контролювати формат і структуру запитів;
- запобігати несанкціонованому запису або видаленню даних;
- централізовано обробляти помилки і доступи.

Контроль запитів до API

На стороні API реалізовано наступні елементи захисту:

- Перевірка типу даних при надсиланні POST/PUT-запитів;
- Обробка помилкових або неповних запитів з поверненням кодів помилок (400, 404, 500);
- Фільтрація запитів по параметрах (наприклад, запити з некоректними координатами не обробляються).

Захист від небажаних дій

Для уникнення перевантаження системи на рівні сервера можливе додавання:

- Обмеження кількості запитів (rate limiting);
- Аудит логів запитів — для виявлення підозрілої активності.

Ці заходи можуть бути реалізовані з використанням сторонніх рішень або middleware-фільтрів у фреймворку Go через Gin middleware.

ВИСНОВКИ

У ході виконання бакалаврської кваліфікаційної роботи було розроблено інформаційну систему пошуку ветеринарних клінік, що дозволяє користувачам знаходити клініки, ознайомлюватися з послугами, читати відгуки, переглядати графік роботи лікарів та здійснювати пошук за фільтрами або геолокацією. Розроблена система орієнтована на мобільну платформу, що відповідає сучасним вимогам до зручності та доступності.

У першому розділі проаналізовано предметну область та існуючі рішення, виявлено основні недоліки аналогів, що дозволило обґрунтувати доцільність створення нової системи. Було сформульовано вимоги до функціональності, розроблено UML-діаграми для моделювання основних бізнес-процесів, сценаріїв взаємодії користувача з системою та логіки її роботи.

У другому розділі здійснено вибір засобів зберігання даних. Після аналізу реляційних та нереляційних баз даних обґрунтовано використання MongoDB як оптимального рішення для документоорієнтованого підходу. Побудовано структуру основних колекцій, описано взаємозв’язки між об’єктами, а також представлено фрагмент схеми збереження даних у реальному середовищі.

У третьому розділі описано архітектуру інформаційної системи. Було реалізовано три незалежні рівні: мобільний клієнт на Flutter, серверну частину на мові програмування Go з REST API та базу даних MongoDB. Детально розглянуто реалізацію ключових функціональних модулів, а також логіку взаємодії між компонентами.

У четвертому розділі проведено впровадження та тестування системи в локальному середовищі. Оцінено апаратні та програмні вимоги,

продемонстровано процес інсталяції та запуску клієнта, сервера й бази даних. Сформульовано типові сценарії використання та наведено результати тестування, які засвідчили стабільну роботу системи. Проведено аналіз ефективності, продуктивності, гнучкості до змін та безпеки. Усі заявлені функції реалізовано, а система показала готовність до реального використання.

Таким чином, поставлену мету — створення інформаційної системи пошуку ветеринарних клінік з мобільним інтерфейсом, адаптивною архітектурою та гнучкою базою даних — досягнуто. Розроблена система може бути використана як у навчальних, так і в реальних практичних цілях. У подальшому її можливо доповнити модулями авторизації, онлайн-запису на прийом та кабінетами клінік, що дозволить значно розширити сферу її застосування.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Документація по створенню зрозумілих інтерфейсів : [Електронний ресурс] – Режим доступу: https://developer.apple.com/design/human-interface-guidelines
2. M. L. Dyndyn (2024). “Application of information technologies in veterinary medicine.” Scientific Messenger of LNU of Veterinary Medicine and Biotechnology. https://www.researchgate.net/publication/389840862_Application_of_information_technologies_in_veterinary_medicine
3. Документація Mongo DB по GEO пошуку: [Електронний ресурс] – Режим доступу: https://www.mongodb.com/docs/manual/geospatial-queries/
4. Petrenko, I. V., & Tkachenko, L. M. (2021). Aktualni aspekty rozvytku telemedytsyny u veterynarii Naukovyi visnyk veterynarnoi medytsyny, 56(2), 78–83 (in Ukrainian).
5. Документація по Go: [Електронний ресурс] – Режим доступу: https://go.dev/doc/
6. Документація по Flutter: [Електронний ресурс] – Режим доступу: https://docs.flutter.dev/
7. Unified Modeling Language (UML) Resource Page: [Електронний ресурс] – Режим доступу до ресурсу: http://www.uml.org/.
8. Introduction to NoSQL: [Електронний ресурс] – Режим доступу до ресурсу: https://www.geeksforgeeks.org/introduction-to-nosql/
9. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. - Addison-Wesley, 1995. - 395 p.

10. Why you should use a Go backend in Flutter: [Електронний ресурс] – Режим доступу до ресурсу: https://blog.logrocket.com/why-use-go-backend-flutter/
11. Go Driver Quick Start: [Електронний ресурс] – Режим доступу до ресурсу: https://www.mongodb.com/docs/drivers/go/current/quick-start/
12. Як будувати UML-діаграми. Розбираємо три найпопулярніші варіанти: [Електронний ресурс] – Режим доступу до ресурсу: https://dou.ua/forums/topic/40575/
13. Google Developers. Flutter Architecture Overview. [Електронний ресурс] – Режим доступу: https://docs.flutter.dev/resources/architectural-overview
14. Fowler, M. (2002). Patterns of Enterprise Application Architecture. Addison-Wesley. – 560 с.
15. How to Build a Rest API in Golang (Detailed Tutorial): [Електронний ресурс] – Режим доступу до ресурсу: https://roadmap.sh/golang/rest-api

ДОДАТОК А

Моделі Бази Даних

Сторінок 4

Київ-2025

```
type Clinic struct {
  ID          primitive.ObjectID   `bson:"_id" json:"id"`
  Name        string               `bson:"name" json:"name"`
  Description string               `bson:"description" json:"description"`
  Contact     Contact              `bson:"contact" json:"contact"`
  Location    GeoLocation          `bson:"location" json:"location"`
  Logo        string               `bson:"logo" json:"logo"`
  Reviews     []Review             `bson:"reviews" json:"reviews"`
  Categories  []string             `bson:"categories" json:"categories"`
  OpenHours   []OpenHours          `bson:"open_hours" json:"open_hours"`
  Doctors     []Doctor             `bson:"doctors" json:"doctors"`
}
type Contact struct {
  Phone   string `bson:"phone" json:"phone"`
  Email   string `bson:"email" json:"email"`
  Website string `bson:"website" json:"website"`
}
type GeoLocation struct {
  Type        string    `bson:"type" json:"type"`
  Coordinates []float64 `bson:"coordinates" json:"coordinates"`
}
type Review struct {
  ID      primitive.ObjectID `bson:"id" json:"id"`
  Rating  int                `bson:"rating" json:"rating"`
  Comment string             `bson:"comment" json:"comment"`
  Date    time.Time          `bson:"date" json:"date"`
}
```

```
type OpenHours struct {
  Day      string `bson:"day" json:"day"`
  Open     string `bson:"open" json:"open"`
  Close    string `bson:"close" json:"close"`
  IsClosed bool   `bson:"is_closed" json:"is_closed"`
}
type Doctor struct {
  ID             primitive.ObjectID `bson:"_id" json:"id"`
  Name           string             `bson:"name" json:"name"`
  Services       []ServiceRef       `bson:"services" json:"services"`
  Qualifications string             `bson:"qualifications" json:"qualifications"`
  Specialty      string             `bson:"specialty" json:"specialty"`
  Experience     int                `bson:"experience_years" json:"experience_years"`
  Description    string             `bson:"description" json:"description"`
  Image          string             `bson:"image" json:"image"`
  Languages      []string           `bson:"languages" json:"languages"`
}
type ServiceRef struct {
  ServiceID primitive.ObjectID `bson:"service_id" json:"service_id"`
  Price     float64            `bson:"price" json:"price"`
  Currency  string             `bson:"currency" json:"currency"`
}
type Category struct {
  ID          string `bson:"_id" json:"id"`
  Name        string `bson:"name" json:"name"`
  Description string `bson:"description" json:"description"`
}
```

```
type Service struct {
  ID          primitive.ObjectID `bson:"_id" json:"id"`
  Name        string             `bson:"name" json:"name"`
  Description string             `bson:"description" json:"description"`
  CategoryID  string             `bson:"category_id" json:"category_id"`
}
```

ДОДАТОК Б

Код програми

Сторінок 7

Київ-2025

```
package db
import("context"
  "fmt"
  "log"
  "os"
  "time"
  "go.mongodb.org/mongo-driver/mongo"
  "go.mongodb.org/mongo-driver/mongo/options")
var (DB * mongo.Client DatabaseName string)
func ConnectDB() {
  mongoURI: = os.Getenv("MONGO_URI") ctx,
  cancel: = context.WithTimeout(context.Background(), 10 * time.Second)
  defer cancel()
  clientOpts: = options.Client().ApplyURI(mongoURI) client,
  err: = mongo.Connect(ctx, clientOpts)
  if err != nil {
    log.Fatalf("MongoDB connection error: %v", err)
  }
  err = client.Ping(ctx, nil) if err != nil {
    log.Fatalf("MongoDB ping error: %v", err)
  }
  fmt.Println("Connected to MongoDB!") DB = client
}
func GetCollection(collectionName string) * mongo.Collection {
  if DB == nil {
    ConnectDB()
  }
```

```
  if DatabaseName == "" {
    DatabaseName = os.Getenv("DATABASE_NAME")
  }
  return DB.Database(DatabaseName).Collection(collectionName)
}
```

```
import 'package:flutter/material.dart';
import 'package:pet_care_hub/data/mock_categories.dart';
import 'package:pet_care_hub/data/mock_clinics.dart';
import 'package:pet_care_hub/data/mock_doctors.dart';
import 'package:pet_care_hub/models/category.dart';
import 'package:pet_care_hub/models/clinic.dart';
import 'package:pet_care_hub/models/doctor.dart';
import 'package:pet_care_hub/pages/doctor/doctor_details_page.dart';
import 'package:pet_care_hub/pages/search/search_page.dart';
import 'package:pet_care_hub/pages/search/search_results_page.dart';
import 'package:pet_care_hub/pages/home/widgets/clinics_widget.dart';
import 'package:pet_care_hub/pages/home/widgets/doctors_widget.dart' as
doctors;
import 'package:pet_care_hub/pages/home/widgets/search_widget.dart';
import 'package:pet_care_hub/pages/home/widgets/services_widget.dart';
import 'package:pet_care_hub/pages/clinic/clinic_details_page.dart';
import 'package:pet_care_hub/pages/map/clinics_map_page.dart';

class MyHomePage extends StatefulWidget {
  const MyHomePage({super.key, required this.title});
```

```
  final String title;

  @override
  State<MyHomePage> createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  void _handleSearch(String query) {
    if (query.isNotEmpty) {
      Navigator.push(
        context,
        MaterialPageRoute(
          builder: (context) => SearchResultsPage(searchQuery: query),
        ),
      );
    }
  }

  void _handleServiceTap(Category service) {
    Navigator.push(
      context,
      MaterialPageRoute(
        builder: (context) => SearchPage(initialCategoryId: service.id),
      ),
    );
  }
```

```
void _handleClinicTap(Clinic clinic) {
  Navigator.push(
    context,
    MaterialPageRoute(
      builder: (context) => ClinicDetailsPage(clinic: clinic),
    ),
  );
}

void _handleDoctorTap(Doctor doctor) {
  Navigator.push(
    context,
    MaterialPageRoute(
      builder: (context) => DoctorDetailsPage(doctor: doctor),
    ),
  );
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    body: SafeArea(
      child: SingleChildScrollView(
        child: Column(
          children: [
            SearchWidget(onSearch: _handleSearch),
```

```
            ServicesWidget(
              services: mockCategories,
              onServiceTap: _handleServiceTap,
            ),
            ClinicsWidget(
              clinics: mockClinics,
              onClinicTap: _handleClinicTap,
            ),
            doctors.DoctorsWidget(
              doctors: mockDoctors,
              onDoctorTap: _handleDoctorTap,
            ),
          ],
        ),
      ),
    ),
    floatingActionButton: FloatingActionButton(
      onPressed: _openMap,
      tooltip: 'Карта',
      heroTag: 'homeMapButton',
      backgroundColor: Theme.of(context).colorScheme.primary,
      foregroundColor: Theme.of(context).colorScheme.onPrimary,
      child: const Icon(Icons.map),
    ),
  );
}
```

```
  void _openMap() {
    Navigator.push(
      context,
      MaterialPageRoute(
        builder: (context) => ClinicsMapPage(clinics: mockClinics),
      ),
    );
  }
}

ntrolled: true,
      shape: const RoundedRectangleBorder(
        borderRadius: BorderRadius.vertical(top: Radius.circular(20)),
      ),
      builder: (context) => _buildClinicPreview(context, clinic),
    ).then((_) {
      // Clear selection when bottom sheet is closed
      if (mounted) {
        setState(() {
          _selectedClinic = null;
        });
      }
    });
  }

  Widget _buildClinicPreview(BuildContext context, Clinic clinic) {
    // Find today's opening hours
```

```
final today = DateTime.now().weekday;
final daysMap = {
  1: 'Monday',
  2: 'Tuesday',
  3: 'Wednesday',
  4: 'Thursday',
  5: 'Friday',
  6: 'Saturday',
  7: 'Sunday',
};
final todayHours = clinic.openHours.firstWhere(
  (hours) => hours.day == daysMap[today],
  orElse: () => OpenHours(
    day: daysMap[today] ?? 'Unknown',
    open: '00:00',
    close: '00:00',
    isClosed: true,
  ),
);

return Container(
  padding: const EdgeInsets.all(16),
  height: 300,
  child: Column(
    crossAxisAlignment: CrossAxisAlignment.start,
    children: [
      Row(
```

```
          crossAxisAlignment: CrossAxisAlignment.start,
          children: [
            ClipRRect(
              borderRadius: BorderRadius.circular(10),
              child: Image.network(
                clinic.imageUrl,
                width: 80,
                height: 80,
                fit: BoxFit.cover,
                errorBuilder: (context, error, stackTrace) {
                  return Container(
                    width: 80,
                    height: 80,
                    color: Theme.of(context).colorScheme.primary.withValues(alpha:
0.1),
                    child: Icon(
                      Icons.local_hospital,
                      size: 40,
                      color: Theme.of(context).colorScheme.primary,
                    ),
                  );
                },
              ),
            ),
            const SizedBox(width: 16),
            Expanded(
              child: Column(
```

```
crossAxisAlignment: CrossAxisAlignment.start,
children: [
  Text(
    clinic.name,
    style: Theme.of(context).textTheme.titleLarge?.copyWith(
      fontWeight: FontWeight.bold,
    ),
    maxLines: 2,
    overflow: TextOverflow.ellipsis,
  ),
  const SizedBox(height: 8),
  Row(
    children: [
      for (int i = 0; i < 5; i++)
        Icon(
          i < clinic.rating.floor()
              ? Icons.star
              : i < clinic.rating
                  ? Icons.star_half
                  : Icons.star_border,
          color: Colors.amber,
          size: 16,
        ),
      const SizedBox(width: 4),
      Text(
        '${clinic.rating}',
        style: Theme.of(context).textTheme.bodyMedium?.copyWith(
```

```
                fontWeight: FontWeight.bold,
              ),
            ),
            Text(
              ' (${clinic.reviewCount})',
              style: Theme.of(context).textTheme.bodySmall,
            ),
          ],
        ),
      ],
    ),
  ),
),
const SizedBox(height: 16),
Text(
  clinic.description,
  style: Theme.of(context).textTheme.bodyMedium,
  maxLines: 3,
  overflow: TextOverflow.ellipsis,
),
const SizedBox(height: 16)
```