МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Чорноморський національний університет
імені Петра Могили
Факультет комп'ютерних наук
Кафедра інженерії програмного забезпечення

ДОПУЩЕНО ДО ЗАХИСТУ

Завідувач кафедри інженерії програмного забезпечення, канд. техн. наук, доцент

_____Є. О. Давиденко

«____» _____ 2022 р.

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

ВЕБЗАСТОСУНОК УНІВЕРСИТЕТСЬКОЇ ПОЛІКЛІНІКИ

Спеціальність «Інженерія програмного забезпечення»

121 – КРБ – 408.21810804

Студент: _____ Д. Р. Бечка
підпис
«__» червня 2022 р.

Керівник: канд. техн. наук, доцент _____ Є. О. Давиденко
підпис
«__» червня 2022 р.

Консультант: канд. техн. наук, доцент _____ А. О. Алексєєва
підпис
«__» червня 2022 р.

Миколаїв – 2022

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ 4
ВСТУП 5
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ВИЯВЛЕННЯ ВИМОГ ДО СИСТЕМИ . 8
1.1 Виявлення цільової платформи системи 8
1.2 Аналіз існуючих реалізацій аналогічних систем 11
1.3 Специфікація вимог до програмного забезпечення 16
Висновки до розділу 1 21
2 МОДЕЛЮВАННЯ ФУНКІОНАЛЬНАЛЬНОЇ СКЛАДОВОЇ СИСТЕМИ 23
2.1 Моделювання сценаріїв використання системи 23
2.2 Моделювання поведінки користувача у системі 29
2.3 Діаграми діяльності 32
Висновки до розділу 2 34
3 ПРОЄКТУВАННЯ АРХІТЕКТУРИ ВЕБЗАСТОСУНКУ 36
3.1 Загальний огляд архітектури застосунку 36
3.2 Вибір технологій реалізації застосунку 39
3.2.1 Технології реалізації клієнтської частини застосунку 40
3.2.2 Технології реалізації серверної частини застосунку 42
3.2.3 Проєктування База даних 44
3.2.4 Вибір вебсервера 46
3.3 Використання сторонніх рішень 48
Висновки до розділу 3 51
4 ПРОГРАМНА РЕАЛІЗАЦІЯ ВЕБЗАСТОСУНКУ 53
4.1 Налаштування середовища розробки 53

4.2 Реалізація серверної частини вебзастосунку 56
4.2.1 Створення моделей та міграцій 56
4.2.2 Налаштування автентифікації застосунку 60
4.2.4 Реалізація API 67
4.3 Реалізація клієнтської частини вебзастосунку 72
4.3.1 Реалізація клієнтського API 73
4.3.2 Управління станом застосунку 74
4.3.3 Налаштування маршрутизації 77
4.4 Огляд користувацького інтерфейсу 79
Висновки до розділу 4 84
ВИСНОВКИ 85
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 87
ДОДАТОК А Лістинг коду для docker-compose 89
ДОДАТОК Б Лістинг коду базового класу контролерів ApiController.php 90
ДОДАТОК В Лістинг коду фабричного методу модулів ресурсів 92

ПЕРЕЛІК СКОРОЧЕНЬ

БД	–	база даних
ОС	–	операційна система
ПЗ	–	програмне забезпечення
СКБД	–	Система керування базами даних
API	–	Application Programming Interface
CSRF	–	Cross-Site Request Forgery
CSS	–	Cascading Style Sheets
HTML	–	HyperText Markup Language
JS	–	JavaScript
JSON	–	JavaScript Object Notation
JWT	–	JSON Web Tokens
SPA	–	Single Page Application
PHP	–	Hypertext Preprocessor
UML	–	Unified Modeling Language
UX	–	User Experience
XSS	–	Cross-Site Scripting

ВСТУП

«Свобода та здоров'я мають одне спільне: по-справжньому цінуєш їх лише тоді, коли їх не вистачає.»

Анрі Бек

Важко не погодитися з цими словами, особливо в тяжкі часи. Коронавірусна пандемія, яка ще недавно була у всіх на слуху. Щоденні новини про нові жертви інфекції COVID-19 викликали у людей страх, і багато людей у ЗМІ та соціальних мережах вже поспішили охрестити новий вірус «чумою XXI століття».

Дійсно, пандемія вимусила увесь світ змінити свій повсякденний ритм життя та відправитися на самоізоляцію та дотримуватися карантинних заходів. Більшості установам, таких як закладам освіти, державним та приватним закладам, банкам, промисловим підприємствам також не також не вдалось оминути від жорстких умов карантину. В результаті чого велику частка бізнесу місила закритися, а багато людей залишились без роботи.

Але пандемія несе за собою не тільки розруху. Головним позитивний трендом під час пандемії стало прискорення цифровізації. Пандемія спровокувала попит на електроніку, онлайн-освіту та інші цифрові продукти. Дійсно такі умови стали поштовхом для швидкого розвитку вебтехнологій. Більша частина існуючого бізнесу переходить в Інтернет. Електронна комерція, розважальні сервіси, онлайн-банкінг, а також, що досить важливо в теперішніх умовах – сервіси надання медичних послуг.

Важко сперечатися з тим, що в наш час сайт поліклініки, лікувально-профілактичного закладу, санаторію, аптеки або постачальника товарів медичного призначення – це один із найважливіших каналів продажу медичних послуг та супутніх товарів. Саме в Інтернеті все більше потенційних пацієнтів/замовників одержують потрібну інформацію, а на основі цієї інформації – приймають рішення про покупку.

Актуальність теми зумовлена тим, що все більш затребуваними стають прогресивні інструменти – онлайн-запис у клініку, онлайн-консультації лікарів, інтернет-магазини тощо. Ігноруючи ці можливості, компанії, що реалізують товари та послуги медичного призначення, позбавляють себе значної частини вхідного потоку клієнтів, а значить – програє у конкурентній боротьбі тим установам, які повною мірою використовують потенціал Інтернету, та не отримують суттєвого додаткового прибутку.

Створення вебзастосунку клініки, медичного центру, приватного кабінету лікаря або лікарні – це можливість:

– Залучити цільових відвідувачів – потенційних пацієнтів чи замовників товарів та послуг медичного призначення.

– Перетворити користувача, що сумнівається, який потребує медичної допомоги, але не впевнений у тому, що конкретно йому необхідно, у реального клієнта з конкретними потребами готового оплатити товари та послуги.

– Збирати базу даних клієнтів для того, щоб постійно підтримувати контакти та пропонувати послуги або товари, здатні їх зацікавити.

– Надати максимум сервісів та інформації онлайн, а значить – розвантажити адміністратора та знизити кількість помилкових звернень.

– Підвищити впізнаваність та сформувати позитивний імідж компанії.

– Збільшити обсяг продажу товарів та послуг медичного призначення і, як наслідок, підвищити прибутковість бізнесу.

Але слід розуміти, що для досягнення такої цілі, дуже важливо щоб система була правильно сконструйована, архітектура системи була придатною для розширення та працювала налагоджено.

Об’єкт кваліфікаційної роботи – процес моделювання та програмної реалізації вебзастосунку університетської поліклініки.

Предмет кваліфікаційної роботи – програмні засоби та методи створення системи за допомогою сучасних інструментів проєктування та розробки програмного забезпечення.

Мета кваліфікаційної роботи – забезпечення рішення для створення та подальшого управління системою університетської поліклініки за рахунок розробки програмного забезпечення з вебінтерфейсом.

Для досягнення поставленої мети необхідно вирішити поставлені нижче задачі:

– аналіз предметної області та методологій розробки;

– аналіз аналогічних систем та на основі отриманих результатів побудувати діаграму використання та описати у вигляді сценаріїв із використанням шаблонів;

– виявлення специфікації вимог до системи;

– проєктування архітектури системи, та огляд сучасних інструментів розробки програмного забезпечення;

– проєктування бази даних системи;

розробка, тестування та розгортання застосунку.Для реалізації поставлених задач використано мову програмування PHP (а саме використання фреймворку Laravel) – для серверної частини застосунку, та JavaScript (фреймворк Vue.js) – для клієнтської частини застосунку.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ВИЯВЛЕННЯ ВИМОГ ДО СИСТЕМИ

1.1 Виявлення цільової платформи системи

Для початку потрібно визначити мету створення системи, які проблеми має вирішувати її функціонал. Що найголовніше – зробити систему максимально зручною з точки зору користувацького досвіду.

Користувацький досвід – ключовий аспект для успіху будь-якого бізнесу. Якість досвіду користувача визначає лояльність клієнта до бізнесу, прийняття рішення про користування послугами бізнесу [1]. Користувальницький досвід зумовлює бажання взаємодіяти із системою. Досвід користувача важливий для збільшення мікро- та макроконверсій, кількості відвідувань або завантажень. Користувальницький досвід здатний привести компанію до успіху чи краху. Суттю UX можна назвати прагнення домогтися розуміння цінності бізнесу з точки зору клієнта.

Дизайн досвіду користувача лежить в основі популярності та успіху бізнесу.

Щоб продукт надавав клієнту значний і корисний досвід користувача, необхідно звернути увагу на кілька нюансів:

– Інтуїтивність вебдизайну.

– Узгодженість та його наступність щодо кнопок та інших елементів, до яких звик користувач.

– Особливості платформи.

Тільки з урахуванням перелічених факторів фахівець у галузі дизайну досвіду користувача здатний досягти позитивних результатів. Саме тому важливо на ранніх етапах проєктування системи виявити показники, на які у майбутньому буде посилатися користувацький досвід.

Нативні (native) застосунки

Це застосунки, що розроблюються для цільових операційних систем – Android, IOS, Windows Phone тощо. Ці застосунки пишуться затвердженими

мовами програмування, безпосередньо під кожну ОС, і тому органічно вбудовуються у відповідні операційні системи. Застосунки завантажуються через маркети (App Store, Google Play тощо), попередньо пройшовши верифікацію на відповідність вимогам цих маркетів [2].

Головна перевага нативних застосунків – те, що вони оптимізовані під конкретні операційні системи, отже, і працюють коректно. Також вони мають доступ до апаратної частини пристроїв, тобто можуть використовувати у своєму функціоналі камеру пристою, мікрофон, геолокацію, адресну книгу, фото, аудіо, відеофайли тощо.

Такі застосунки можуть працювати як онлайн так і офлайн, тому користувачі мають можливість користуватися додатком як їм зручно.

Зрозуміло, для написання такого продукту необхідно володіти спеціальними знаннями та вміннями для роботи в конкретному середовищі розробки (xCode для iPhone, eclipse для пристроїв на Android). Звичайно вартість таких додатків набагато вища через їхню трудомісткість і те, що під кожну платформу доводиться писати окремий додаток і іншою мовою.

Через складність, як з точки зору розробки так і з точки зору часу, який знадобився би для реалізації системи – такий тип застосунків одразу ж був відсіяний під час вибору цільової платформи.

Гібридні застосунки

Гібридні застосунки схожі за функціоналом та якістю з нативними. Це щось середнє між нативними та вебзастосунками. Подібно до нативних, вони встановлюються через офіційні магазини, мають обмежений доступ до апаратної частини пристроїв, в них можна налаштовувати push-сповіщення. А також гібридні програми вимагають менше ресурсів для розробки в порівнянні з нативними програмами.

Якість та можливості гібридних застосунків залежать від самого фреймворку, яким користувався розробник та їх якість буде пов’язана з вартістю.

Такий тип застосунків може здатися вдалим вибором для розробки системи, але він також має ряд недоліків, які вливають на остаточне рішення вибору цільової платформи ведення бізнесу:

– обмеження обсягу даних, що зберігаються в застосунку, доведеться періодично щось додатково підвантажувати;

– не має можливості отримати інформацію про тип мережного з’єднання;

– обмеження, що накладаються операційними системами;

– відсутність можливості працювати з файлами та папками;

– обмеження в реалізації користувацького інтерфейсу;

– складний процес оптимізації для різних девайсів.

Вебзастосунки

Вебзастосунок – це прикладна програма, яка зберігається на віддаленому сервері та передається через Інтернет через інтерфейс браузера. Вебсервіси за визначенням є вебзастосунками, і багато вебсайтів, хоча й не всі, містять вебзастосунки.

Вебзастосунки можуть бути розроблені для широкого спектру використання і можуть використовуватися будь-ким; від організації до окремої особи з багатьох причин. Зазвичай використовувані вебзастосунки можуть включати вебпошту, онлайн-калькулятори або магазини електронної комерції. До деяких застосунків можна отримати доступ лише через певний браузер; однак більшість із них доступні незалежно від браузера.

Такі застосунки не потрібно завантажувати, оскільки доступ до них здійснюється через мережу. Користувачі можуть отримати доступ до них через веббраузер, такий як Google Chrome, Mozilla Firefox або Safari.

Для роботи вебзастосунка потрібні вебсервер, клієнтська сторона і база даних. Вебсервери керують запитами, які надходять від клієнта. Базу даних можна використовувати для зберігання будь-якої необхідної інформації.

Вебзастосунки зазвичай мають короткі цикли розробки і можуть бути створені невеликими командами розробників [3]. Більшість із них написані на JavaScript, HTML5 та CSS. Програмування на стороні клієнта зазвичай використовує ці мови, які допомагають створювати інтерфейс програми. Програмування на стороні сервера виконується для створення сценаріїв, які використовуватиме застосунок. Такі мови, як Python, Java і PHP, зазвичай використовуються в програмуванні на стороні сервера.

Вебзастосунки мають багато різних застосувань, і разом із цим використанням приходить багато потенційних переваг. Деякі загальні переваги:

– надання доступу кільком користувачам до однієї версії програми;
– вебзастосунки не потрібно встановлювати;
– доступ до вебзастосунків можна отримати через різні платформи, такі як настільний комп’ютер, ноутбук або смартфон;
– можна отримати доступ через різні браузери.

1.2 Аналіз існуючих реалізацій аналогічних систем

Для виявлення основних вимог до розроблювальної системи проведено аналіз аналогічних систем. Аналіз існуючих реалізацій аналогічних систем проводиться для досягнення наступних цілей:

1) Визначення позиції в ніші

Практично кожен застосунок має конкурентів. Щоб зайняти лідируючі позиції, важливо розуміти, за якими ключовими запитами просуваються конкуренти, які позиції вони займають. Крім того, потрібно постійно перевіряти наявність оновлень. Це допоможе не пропустити нові функції та зміни оптимізації.

2) Підвищення рівню лояльності клієнтів

Аналізуючи близьких і далеких конкурентів (наприклад, іноземних або близьких до ніші), можна знайти ідеї для розвитку продукту. В результаті з’явиться список нових функціональних можливостей, які можна застосувати у

соєму програмному рішенні, щоб зробити його максимально цікавою для потенційних користувачів системи.

3) Визначення слабких та сильних сторін аналогічних систем

Аналіз дає розуміння, сильних сторін продукту і на що варто звернути увагу – це може стати конкурентною перевагою. Також будуть ідеї щодо покращення застосунку та впровадження нових функцій [4].

Вебзастосунок клініки Київського національного університету ім. Тараса Шевченка.

Система представляє з себе інформаційний портал, головна мета якого популяризація інформації про клініку при Київському національному університеті ім. Тараса Шевченка. На сайті зібрана основна інформація про сам медичний заклад, історію створення, час роботи та контактну інформацію, а також присутня інформація про наукову діяльність та навчання в університеті (рис. 1.1). Основна ціль таких застосунків – організувати і розділити інформаційне простір для задоволення користувачів запитів. Такі застосунки використовуються в якості маркетплейсу: розміщення рекламних публікації, просувати ідеї, популяризувати компанії, продукти та послуги.

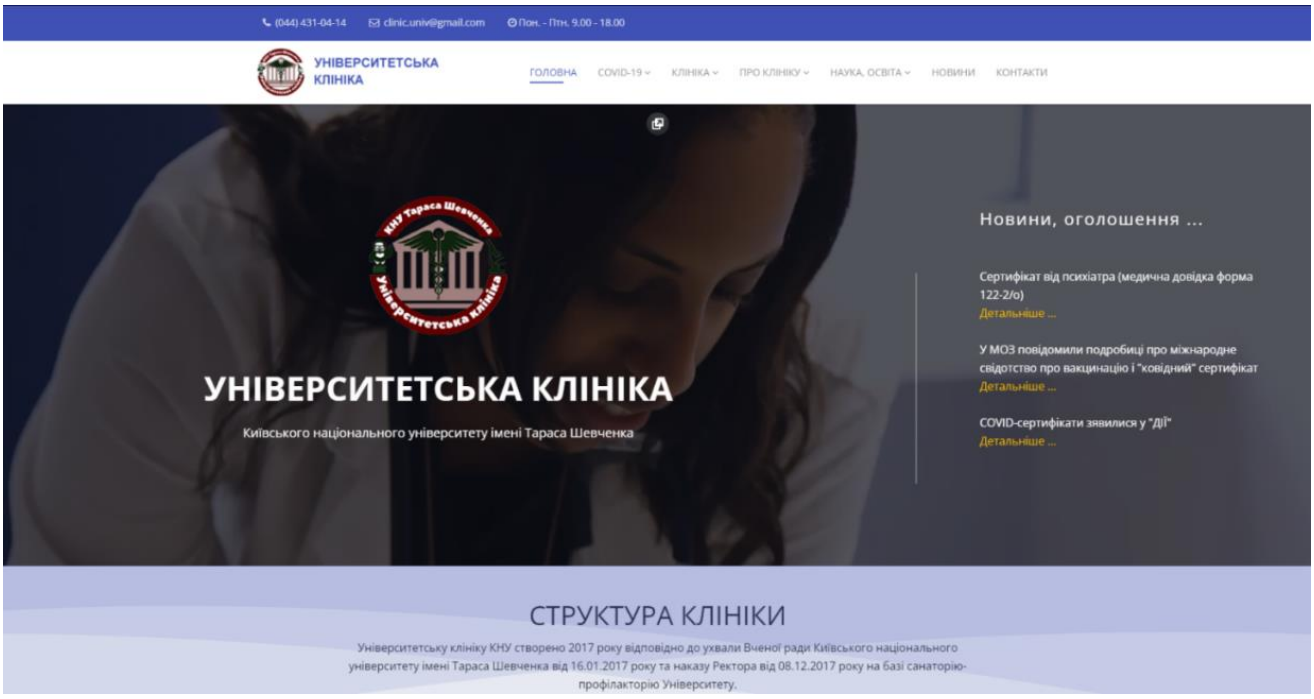

Рисунок 1.1 – Головна сторінка вебзастосунку клініки Київського національного університету ім. Тараса Шевченка

Вебзастосунок клініки Одеського національного медичного університету.

Даний вебсайт, як і в попередньому випадку має направлення інформаційного порталу. Але дана система має більш просунуті функціональні можливості (рис. 1.2). Дана система повністю базується навколо навчального закладу, тому основна інформація та функції пов’язані із науковою діяльністю та отриманням медичної освіти у закладі. Незважаючи на це, вона також має багато недоліків, на які також слід звернути при проєктуванні нової системи. Основним недоліком даного сайту є те, що за своїми функціональними можливостями, даний продукт не дотягує до повноцінного, добре спроєктованого вебзастосунку.

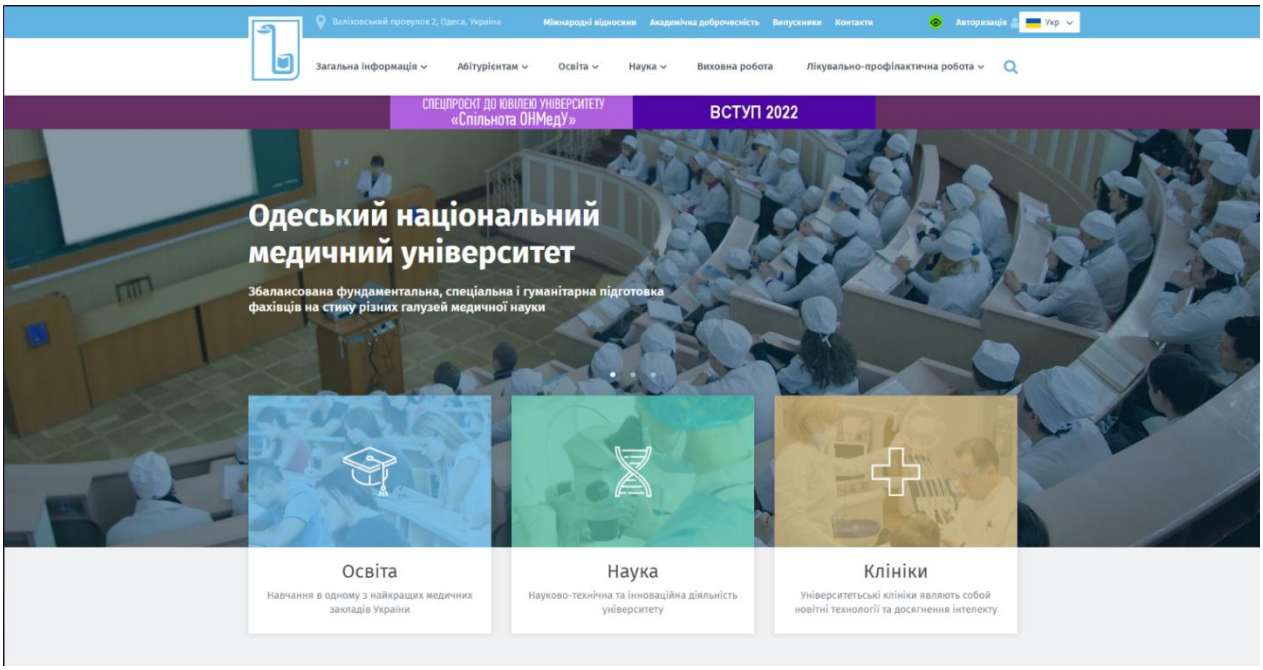

Рисунок 1.2 – Головна сторінка вебзастосунку Одеського національного медичного університету

Вебзастосунок «Добробут»

Система «Добробут» – гарний приклад добре спроєктованого вебзастосунку. Сайт містить повну інформацію про компанію та послуги, які вона

пропонує. Основна відмінність даного сайту від попередніх полягає в розширеній функціональності та можливої інтеграції з внутрішніми системами компанії. На це вказує наявність потужних інструментів, орієнтованих на бізнес (рис. 1.3). При створенні власного вебзастосунку для медичного закладу, слід приділити увагу саме цьому аналогу, адже дана система володіє основними функціональними можливостями, для системи у даній галузі.

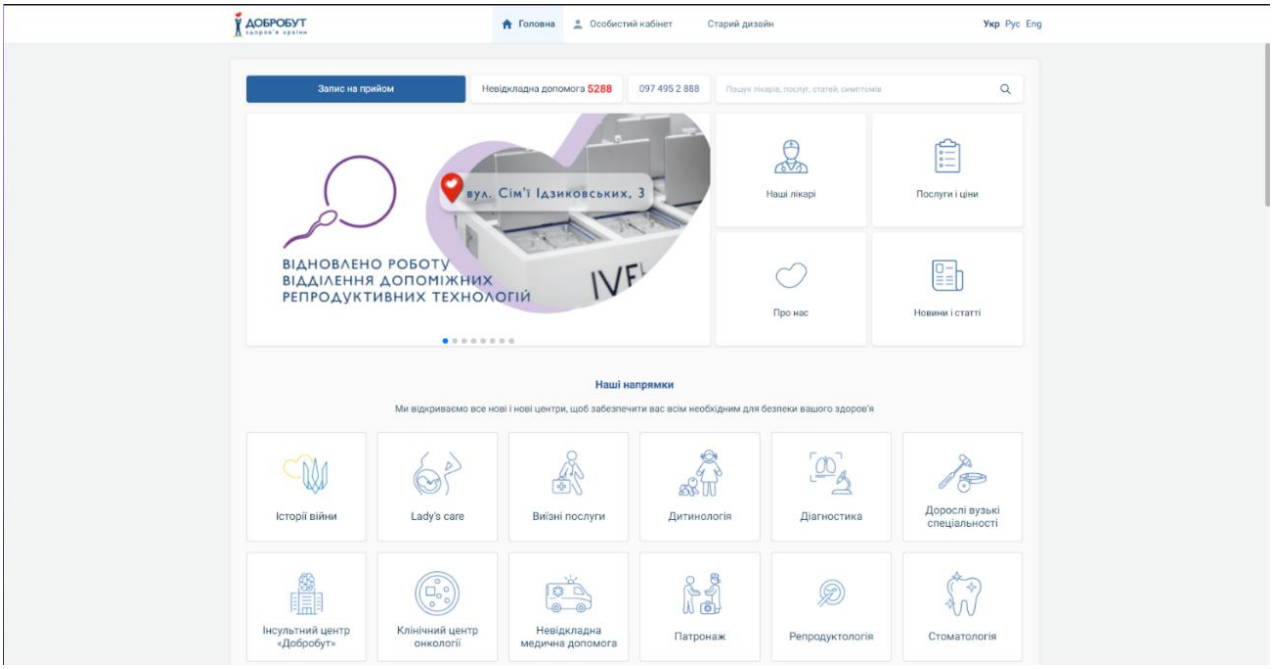

Рисунок 1.3 – Головна сторінка вебзастосунку «Добробут»

Для оглянутих систем можна виділити ряд основних переваг та недоліків (табл. 1.1).

Таблиця 1.1 – Порівняння характеристик аналогічних систем

Назва системи	**Переваги**	**Недоліки**
Вебзастосунок клініки Київського національного університету ім. Тараса Шевченка	1) швидкість відгуку сайту на дії користувача; 2) можливість створення облікового запису користувача; 3) можливість змінити мову інтерфейсу; 4) наявність пошуку по сайту;	1) відсутність функціоналу пошуку на сайті; 2) відсутність можливості записатися на прийом до лікаря; 3) відсутність можливості створення облікового запису користувача;

	5) наявність контактної інформації та зворотній зв’язок;	4) відсутність можливості залишати відгуки.

Кінець таблиці 1.1

	6) наявність блоку з новинами; 7) можливість налаштувати відображення користувацького інтерфейсу для людей з обмеженими можливостями.	
Вебзастосунок Одеського національного медичного університету	1) можливість створення облікового запису користувача; 2) можливість змінити мову інтерфейсу; 3) наявність пошуку по сайту; 4) наявність контактної інформації та зворотній зв’язок; 5) наявність блоку з новинами; 6) можливість налаштувати відображення користувацького інтерфейсу для людей з обмеженими можливостями.	1) швидкість відгуку сайту на дії користувача; 2) переповнений зайвими елементами дизайн; 3) відсутність можливість фільтрації результатів пошуку; 4) відсутність можливості залишити відгук; 5) відсутність можливості перегляду детальної інформації про лікарів; 6) відсутність можливості записатися на прийом.

Вебзастосунок «Добробут»	1) можливість створення облікового запису та авторизації; 2) наявність пошуку інформації на сайті; 3) можливість записатися на прийом до лікаря; 4) наявність контактної інформації та зворотній зв'язок; 5) наявність блоку з новинами; 6) можливість залишити відгук; 7) швидкість відгуку сайту на дії користувача; 8) можливість перегляду детальної інформації про лікарів та послуги.	1) занадто простий та монотонний дизайн; 2) переходи між сторінками занадто різкі, що погано впливає на користувацький досвід;

1.3 Специфікація вимог до програмного забезпечення

Після проведеного аналізу та виділення основних функціональних можливостей для застосунку, можна відвести підсумки у вигляді специфікації вимог до програмного забезпечення [5]. Специфікація вимог до програмного забезпечення (SRS) – це документ, який описує, що буде робити програмне забезпечення та як воно буде працювати. Він також описує функціональність, необхідну продукту для задоволення потреб усіх зацікавлених сторін (бізнесу, та користувачів).

Документ SRS усуває всю двозначність, пов'язану з неписьмовими формами презентації проекту, такими як усні дискусії. SRS полегшує розробку продукту та запобігає неправильній інтерпретації, що призводить до створення неправильного продукту і, в кінцевому підсумку, до провалу проекту. SRS гарантує, що все необхідне для створення проєкту описано в письмовій формі.

ПРИЗНАЧЕННЯ ТА МЕЖІ ПРОЄКТУ

Призначення системи (застосунку), для якої розробляється програмне забезпечення:

Надання клієнтам університетської поліклініки можливості користуватися послугами закладу, такими як пошук лікарів та запис на прийом за допомогою вебінтерфейсу розроблюваного застосунку.

ЗАГАЛЬНИЙ ОПИС

Характеристики користувачів

У системі повинні бути передбаченні наступні ролі користувачів з різними правами доступу до функцій системи:

– Користувач (User) – роль з обмеженими правами доступу до функцій системи, повинна бути присвоєна переважній більшості користувачів системи.

– Адміністратор (Administrator) – роль з усіма правами доступу у системі. Для забезпечення безпеки системи, роль повинна бути присвоєна лише одному користувачеві.

– Контент-менеджер (Content Manager) – роль з розширеними правами доступу до функцій системи. Причому для різних користувачів з цією роллю права доступу можуть відрізнятися.

Загальна структура і склад системи

Система повинна представляти з себе повноцінний вебзастосунок, який складається з основного сайту, до якого отримують доступ усі користувачі системи, та панелі адміністратора – доступної для користувачів зі спеціальними правами доступу.

Загальні обмеження

Обмеженням для функціонування застосунку полягає у наявності інтернет-з’єднання користувача.

ФУНКЦІЇ СИСТЕМИ:

– Система повинна давати можливість додавати, редагувати, видаляти та переглядати інформацію про лікарів, спеціалізації, послуги, процедури, блоги та блог-пости.

– Система повинна дозволяти здійснювати пошук даних та фільтрувати результати пошуку по полях деяких таблиць: Послуги (Код послуги, назва), Лікарі (Код лікаря, прізвище і ініціали, спеціалізація, рейтинг), Блог-пости (Назва, частина контенту, автор).

– Система повинна надавати можливість налаштовувати навігацію сайту.

– Система повинна давати можливість управляти правами доступу для різних користувачів.

– Система повинна давати користувачу змогу потратити до особистого кабінету після проходження авторизації.

– Система повинна давати користувачу можливість записатися на прийом до лікаря, шляхом вказання особистої інформації (ПІБ, номер телефону або електронна пошта) з можливістю обрати доступні дату, час запису, лікаря та процедуру.

– Користувач повинен мати змогу залишати відгуки та оцінювати якість роботи окремих лікарів.

– Адміністратор повинен мати можливість модерувати відгуки користувачів та оцінки користувачів.

– Система повинна обчислювати загальний рейтинг лікарів основуючись на оцінках користувачів.

ВИМОГИ ДО ТЕХНІЧНОГО ЗАБЕЗПЕЧЕННЯ

Система розробляється для функціонування на пристроях з графічним дисплеєм, та можливістю отримати доступ до інтернету із браузеру.

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Архітектура програмної системи

Програмна система повинна реалізовувати клієнт-серверну архітектуру, яка повинна містити наступні складові частини:

– сервер, який надає інформацію або інші послуги програмам, які звертаються до нього;

– набір клієнтів, які використовують сервіси, що надаються серверами;

– мережа, яка забезпечує взаємодію між клієнтами та сервером.

Системне програмне забезпечення

Система має бути розгорнута на обладнанні із встановленою операційною системою сімейства GNU/Linux.

Мережне програмне забезпечення

Клієнтська частина застосунку повинна відкриватися за допомогою браузеру (Safari, Mozilla Firefox, Google Chrome, Opera, Edge).

Програмне забезпечення ведення інформаційної бази

У якості системи управління базами даних повинна виступати PostgreSQL версії 12.5 і вище.

Мова і технологія розробки ПЗ

Серверна частина застосунку повинна бути реалізована на мові програмування PHP, з використанням фреймворку Laravel.

Клієнтська частина застосунку повинна бути написана на JS та використовувати фреймворк Vue.js.

ВИМОГИ ДО ЗОВНІШНІХ ІНТЕРФЕЙСІВ

Інтерфейс користувача

Інтерфейс користувача системи повинен відповідати наступним вимогам:

– Графічні інтерфейси мають бути захищені від несанкціонованих та нетипових дій користувача. Ця вимога відноситься в першу чергу до вебсайту користувача.

– Графічні інтерфейси повинні виводити повідомлення про помилку у разі некоректних або помилкових дій системи.

– Графічні інтерфейси повинні виводити повідомлення про помилку у разі некоректних дій користувача.

– Вважається неприйнятним, якщо будь-якими діями у вебінтерфейсі користувач міг викликати недієздатність системи частково або повністю для інших користувачів системи.

Програмний інтерфейс

Клієнтська частина системи повинна взаємодіяти із серверною частиною за допомогою інтерфейсу Rest API.

Жодних сторонніх програмних інтерфейсів в системі використовуватися не повинно.

Комунікаційний протокол

Система повинна використовувати протокол HTTPS для передачі даних між клієнтом та сервером.

ВЛАСТИВОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Доступність

Система повинна підтримувати цілодобовий режим роботи.

Переносимість

Система повинна працювати на будь-яких девайсах, на яких встановлений браузер з підтримуваною версією. Вебзастосунок повинен бути кросбраузерним – мати однаковий користувацький інтерфейс та мати однакову поведінку не залежно від браузера в якому він відкривається.

Продуктивність

Час від надходження запиту обслуговування клієнта до віддачі контенту клієнту сервісом (тобто. загальний час обробки запиту на серверної частини) має становити не більше 5 секунд при нормальному навантаженні обладнання.

Надійність

– Система повинна зберігати працездатність у разі відмови або виходу з ладу вебсерверів.

– Необхідно забезпечити збереження всієї накопиченої інформації на момент відмови або виходу з експлуатації одного з дискових накопичувачів.

– У разі повного відключення електроенергії система повинна обробити та після відновлення електроенергії запустити повторно на виконання всі незавершені операції.

– Система має забезпечувати працездатність у разі збою одного зі своїх модулів.

Безпека

Користувач системи має доступ до інформації, доступної його ролі та прав доступу до різних частин системи. Дані клієнтів та користувачів повинні бути захищені від несанкціонованих запитів до БД. Паролі користувачів повинні зберігатися в БД у зашифрованому вигляді за допомогою криптографічного алгоритму Bcrypt.

ІНШІ ВИМОГИ

Система не має додаткових вимог, окрім описаних у даному документі.

Висновки до розділу 1

У першому розділі було проаналізовано предметну область та визначено цільову платформу для розроблювальної системи. Найбільш підходящою платформою було визнано веб.

На основі проведеного аналізу аналогічних систем на цільовій платформі можна зробити висновок, що проаналізовані системи мають багато недоліків, серед яких можна виділити обмежений функціонал та застарілий дизайн. Переглянуті системи в основному представляють з себе більше інформаційні портали аніж повноцінні вебзастосунки. Цей факт є показником того, що на даний момент ніша автоматизованих систем для медичних установ не переповнена, тому розробка системи яка може зайняти місце в цій ніші є доцільним рішенням.

Із результатів аналізу програмних рішень інформаційних систем медичних закладів можна сформулювати завдання, що стоїть при розробці даної системи: потрібно розробити інформаційну систему електронної реєстратури, що забезпечує реалізацію всіх основних стадій пошуку лікарів та послуг медичного

закладу, запису пацієнта на прийом до лікаря та отримання талону на прийом і вигідно виділяє вебзастосунок серед аналогічних систем.

Результатом проведеної у розділі роботи стало виявлення специфікації вимоги до розроблюваного програмного забезпечення.

2 МОДЕЛЮВАННЯ ФУНКІОНАЛЬНАЛЬНОЇ СКЛАДОВОЇ СИСТЕМИ

Після проведення аналізу ринку та складання специфікації вимог слідує наступна стадія розробки програмного забезпечення – моделювання теоретичної основи продукту. Сучасні засоби програмування дозволяють частково об’єднати етапи проєктування та програмної реалізації, будучи заснованими на об’єктно-орієнтованому підході, але повноцінне планування потребує більш ретельного та скрупульозного моделювання.

Системне моделювання вносить додаткову формальність у процеси аналізу та проєктування. У процесі розробки системи часто використовуються схеми і малюнки, які допомагають наочно відобразити деякі аспекти розробки. Системне моделювання формалізує це наочне уявлення як з допомогою діаграм, виконаних з використанням стандартних нотацій (синтаксису), а й забезпечує середовище (засоби) розуміння і обговорення ідей, що з процесом розробки [6].

Моделювання системи має такі переваги:

– заохочує використання певної термінології, однозначність якої підтримується в рамках розробки всієї системи;

– дозволяє за допомогою діаграм отримати наочне представлення системних специфікацій та архітектури системи;

– дозволяє підтвердити достовірність деяких аспектів поведінки системи за допомогою динамічних моделей;

– дозволяє постійно удосконалювати систему за допомогою уточнення архітектури, підтримуючи генерацію тестів та вихідного коду.

2.1 Моделювання сценаріїв використання системи

Варіант використання фіксує угоду між учасниками системи щодо її поведінки. Варіант використання описує поведінку системи при її відповідях на запит одного з учасників, що називається основною дійовою особою, у різних умовах. Кожен варіант використання має опис. Основний варіант використання

представляє стандартний потік подій у системі, а альтернативні шляхи описують варіації поведінки.

Не існує стандартизованого формату сценарію використання, тому кожна організація стикається з визначенням стандартів, які слід включити. Часто варіанти використання документуються за допомогою шаблону документа варіантів використання, попередньо визначеного організацією, що полегшує читання варіантів використання та надає стандартизовану інформацію для кожного варіанту використання в моделі.

Сценарій повинен легко читатися. Тому вам слід уникати сценаріїв з більш ніж дев'ятьма кроками, і завжди повинен використовуватися наголос на тому хто та що робить [7].

Основуючись на специфікації вимог до системи, можна виділити основні сценарії (табл. 2.1 – 2.5). У даному розділі наведені деякі з основних сценаріїв використання розроблюваної системи. Причому найбільш розповсюджені сценарії, такі як реєстрація або авторизація, не розглядаються.

Таблиця 2.1 – Сценарій редагування даних про лікаря

Діючі особи	Адміністратор, Система
Мета	Відредагувати інформацію про лікаря
Передумова	Адміністратор авторизований в панелі адміністратора та має відповідні права доступу для редагування даних про лікаря
Успішний сценарій: 1) Адміністратор переходить на сторінку із списком лікарів; 2) Система відображає список лікарів; 3) Адміністратор обирає запис, який потрібно відредагувати; 4) Адміністратор переходить на сторінку редагування даних лікаря; 5) Адміністратор вносить зміни та відправляє форму; 6) Система змінює дані у базі даних відповідно до змін внесених Адміністратором;	

Кінець таблиці 2.1

Результат	Адміністратор змінив данні про лікаря
Розширення:	
5а	Форма не пройшла валідацію Результат: поля форми підсвічуються червоним кольором, система видає відповідне повідомлення про помилку

Сценарій редагування даних про лікаря (табл. 2.1) також можна застосувати і для інших моделей системи, які доступні для редагування в панелі адміністратора. Даний сценарій також може бути використаний при створені нових записів відповідних моделей. Тому для уникнення повторювань дані сценарії використання упущені.

Таблиця 2.2 – Сценарій пошуку лікарів користувачем

Діючі особи	Користувач, Система
Мета	Знайти лікарів за певними ім’ям або категорією
Успішний сценарій: 1) Користувач вводить повну або часткову назву категорії або ім’я лікаря у поле для пошуку; 2) Система здійснює пошук по лікарям у базі даних які відповідають запиту; 3) Система повертає користувачеві інформацію про лікарів, що задовольняють відправлений запит.	
Результат	Користувач отримав сторінку з інформацією про лікарів.
Розширення:	
3а	Система не знайшла жодного збігу за запитом користувача серед усіх записів про лікарів у базі даних Результат: користувач отримує повідомлення про відсутність знайдених результатів пошуку, що задовольняють умовам.

Таблиця 2.3 – Сценарій додавання відгуку користувачем

Діючі особи	Користувач
Мета	Залишити відгук про певного лікаря.
Передумова	Користувач авторизований та знаходиться на сторінці відображення інформації про лікаря.
Пост-умова	Відгук відправляється на модерацію.
Успішний сценарій: 1) Користувач відкриває форму додавання відгуку 2) Користувач заповнює контактну інформацію; 3) Користувач пише текст відгука; 4) Користувач ставить оцінку лікарю та зберігає відгук;	
Результат	Користувач залишив відгук про лікаря.

Таблиця 2.4 – Сценарій модерації відгуків користувачів

Діючі особи	Контент-менеджер, Користувач, Система
Мета	Прийняти відправлений користувачем відгук.
Передумова	Контент-менеджер авторизований в панелі адміністратора.
Успішний сценарій: 1) Контент-менеджер переходить на сторінку із відгуками користувачів; 2) Система відображає дані, надіслані користувачем; 3) Контент-менеджер перевіряє на відгук дані, після чого підтверджує заявку; 4) Контент-менеджер зберігає зміни; 5) Система змінює дані у базі даних відповідно до прийнятого рішення Контент-менеджером; 6) Користувач отримує повідомлення про успішне опублікування відгуку.	
Результат	Контент-менеджер обробив заяву залишення відгуку.

Кінець таблиці 2.4

Розширення:	
3а	Контент-менеджер відхиляє заявку та вказує причину. Результат: користувачу на вказану пошту відправляється повідомлення з поясненням.

Таблиця 2.5 – Сценарій запису користувача на прийом

Діючі особи	Користувач
Мета	Записатися на прийом до певного лікаря.
Передумова	Користувач знаходиться на сторінці із відображення інформації про лікаря або на сторінці запису на прийом.
Успішний сценарій: 1) Користувач переходить на сторінку лікаря, або на сторінку запису; 2) Користувач обирає доступний час, послугу та лікаря; 3) Користувач підтверджує інформацію про запис.	
Результат	Користувач записався на прийом до лікаря.

Для наглядності сценарії використання можна доповнити діаграмою прецедентів. Діаграма прецедентів – це тип поведінкової діаграми UML, часто використовується для аналізу систем [7]. Вони дозволяють візуалізувати різні типи ролей у системі і те, як ці ролі взаємодіють із системою. Прецеденти є цінним інструментом для розуміння функціональних вимог до системи. Перший варіант прецедентів має складатися на ранній стадії виконання проєкту. Докладніші версії прецедентів можуть з’являтися безпосередньо перед реалізацією цього прецеденту.

Діаграма варіантів використання включає 2 основних елементи:

1) Актор (учасник): набір логічно пов’язаних ролей, які виконуються під час взаємодії з прецедентами або сутностями (системою, підсистемою чи класом).

Учасником може бути особа, роль людини в системі чи іншій системі, підсистемі чи класі, що представляє щось поза сутністю.

2) Випадок використання (прецедент): опис окремого аспекту поведінки системи з точки зору користувача. Прецедент показує не те, «як» досягається певний результат, а лише «який».

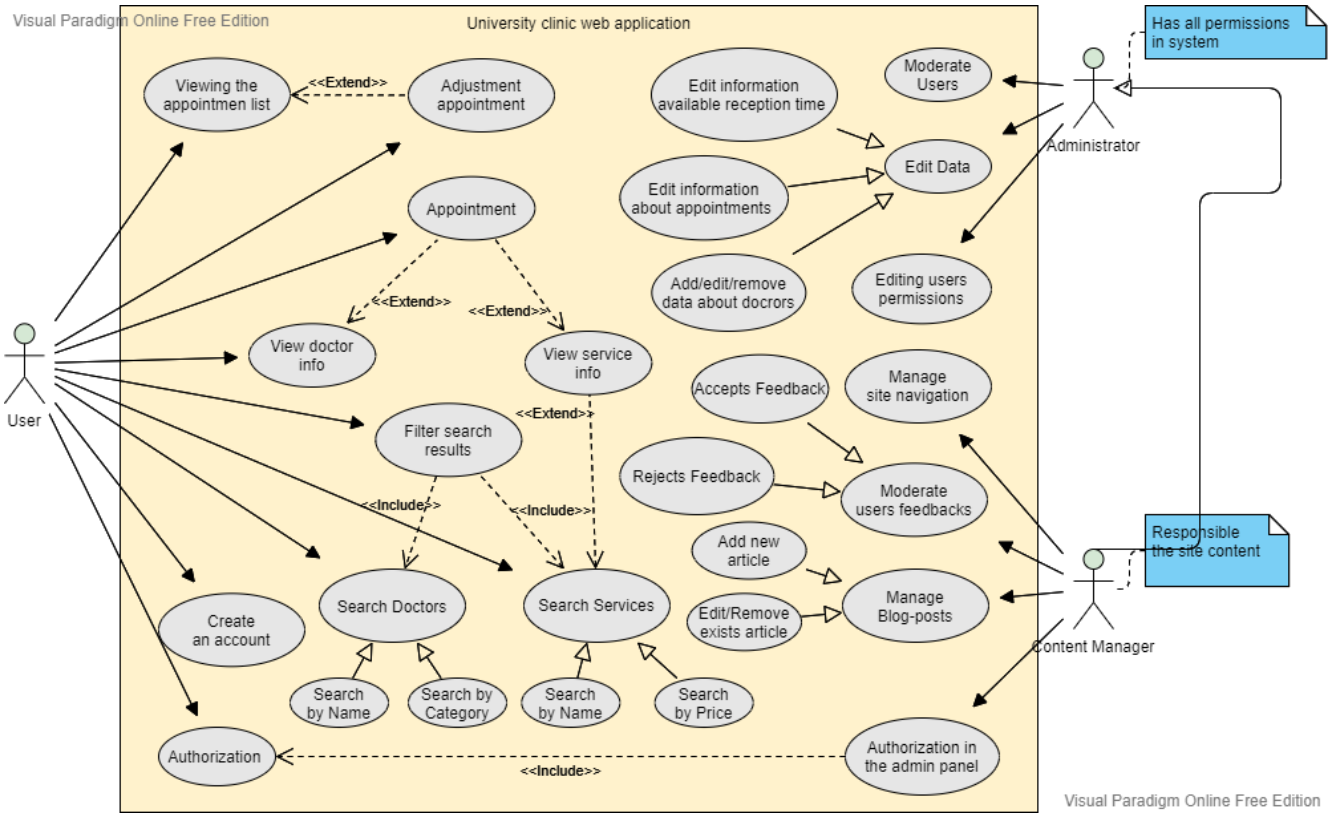

Рисунок 2.1 – Діаграма прецедентів системи

На рис. 2.1, згідно із специфікацією вимог до системи, представлені основні актори розроблюваної системи:

– адміністратор (Administrator) – має повний доступ до всіх функцій системи, включаючи основний сайт та панель адміністратора;

– контент-менеджер (Content Manager) – має доступ до функцій системи, які відповідають за наповнення сайту контентом, модерація блогу, коментарів користувачів, редагування навігації сайту.

– користувач (User) – має можливість створити обліковий запис, переглядати базову інформацію на сайті, пошук лікарів та послуг з можливістю

фільтрування результатів пошуку, можливість залишати відгуки та записатися на прийом до лікаря.

2.2 Моделювання поведінки користувача у системі

Діаграми послідовностей використовуються для уточнення діаграм прецедентів більш детального опису логіки сценаріїв використання. Це чудовий засіб документування проєкту з погляду сценаріїв використання [8].

Діаграми послідовностей зазвичай містять об’єкти, які взаємодіють у рамках сценарію, повідомлення, якими вони обмінюються, і результати, пов’язані з повідомленнями, що повертаються. Втім, результати, що часто повертаються, позначають лише в тому випадку, якщо це не очевидно з контексту.

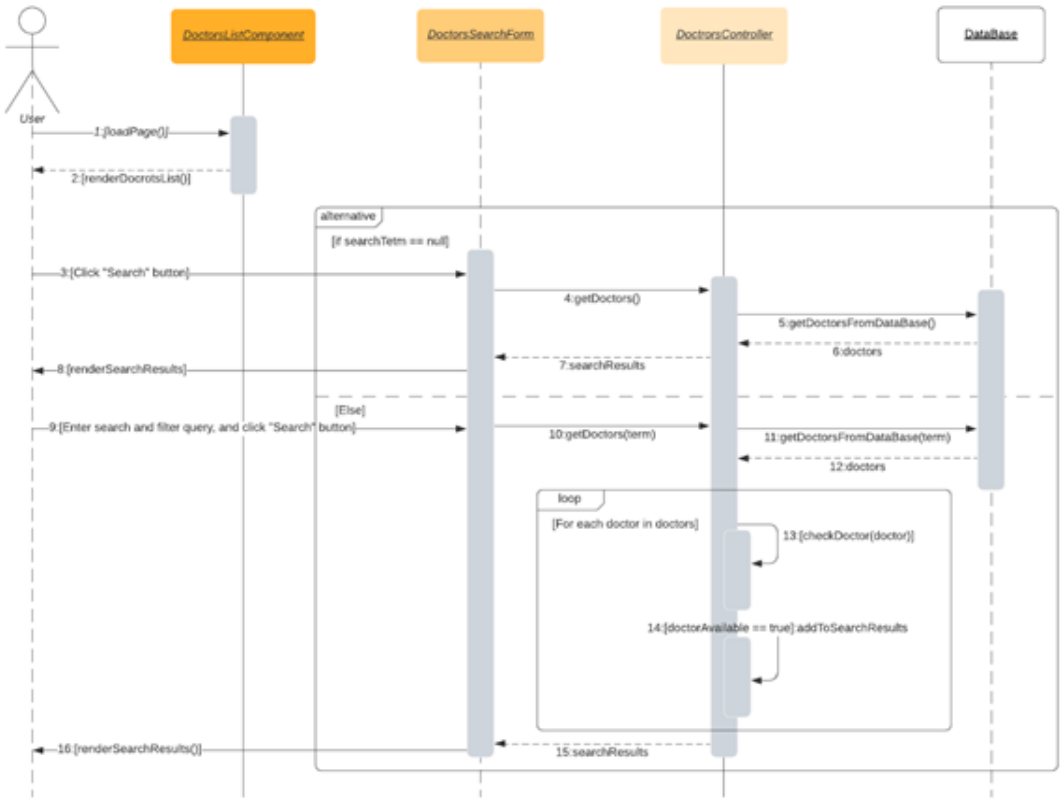

Рисунок 2.2 – Діаграма послідовності «Пошук та фільтрація лікарів»

Діаграма послідовності «Пошук та фільтрація лікарів» демонструє етапи пошуку лікарів користувачем (рис. 2.2). При виконанні даного варіанту використання користувач переходить до сторінки пошуку, після чого відбувається

завантаження сторінки (1), при завантаженні на сторінці відображається форма пошуку з опціями фільтрації (2), користувач може виконати пошук лікаря ввівши ім’я лікаря – повне або частково. У випадку, якщо користувач відправляє порожню форму пошуку натиснувши кнопку «Пошук» (3), відбувається звернення до контролеру на отримання інформації про лікарів без параметрів (4), контролер в свою чергу відправляє запит до бази даних на отримання інформації про всіх лікарів (5), після чого з бази даних повертаються всі записи про лікарів (6). Після цього результати пошуку повертаються до компоненту форми пошуку (7), які в свою чергу відображає отримані результати на сторінці (8). У випадку, якщо користувач заповнив форму пошуку перед відправкою (9), відбуваються ті ж етапи, що і при відправці порожньої форми, але в параметрами пошуку (10 - 12). Після того, як результати пошуку були отримані з бази даних (12), у контролері відбувається фільтрація результатів за додатковими параметрами (13). Записи, які відповідають параметрам фільтрів потрапляють до списку результатів (14). Після чого контролер повертає відфільтрований список результатів та відбувається відображення списку результатів (15 - 16).

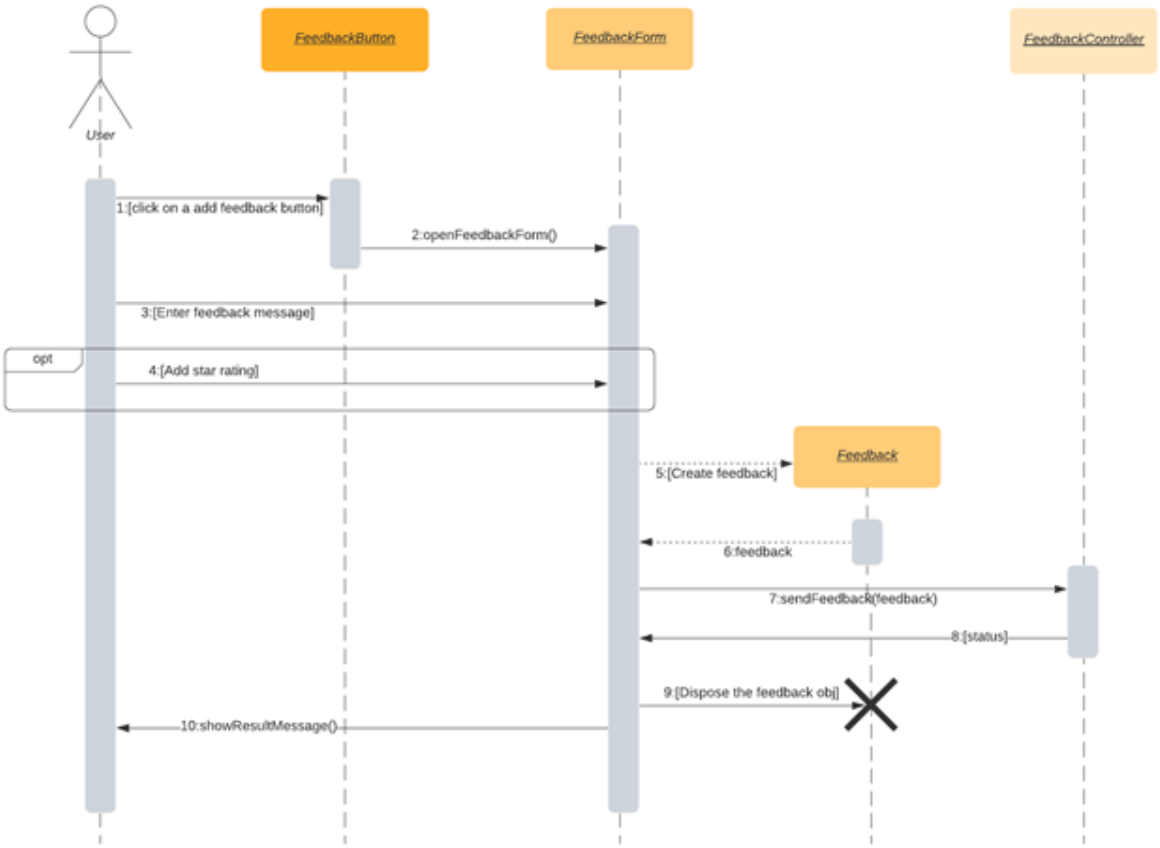

Рисунок 2.3 – Діаграма послідовності «Додавання відгуку користувачем»

На діаграмі послідовності «Додавання відгуку користувачем» зображені етапи додавання відгуку користувачем (рис. 3.3). Для того, щоб запустити даний сценарій, користувач натискає кнопку «Залишити відгук» (1). Після цього буде відображена форма для відгуку (2), користувач вводить повідомлення (3), та за бажанням ставить оцінку у вигляді рейтингу (4). Після відправки форми, створюється об’єкт feedback (5 - 6). Далі новостворений об’єкт відправляється до бази даних (7), після чого база даних повертає статус виконання запиту (8). Після отримання відповіді від бази даних, об’єкт feedback видаляється (9), а користувач отримує повідомлення про статус збереження відгуку (10).

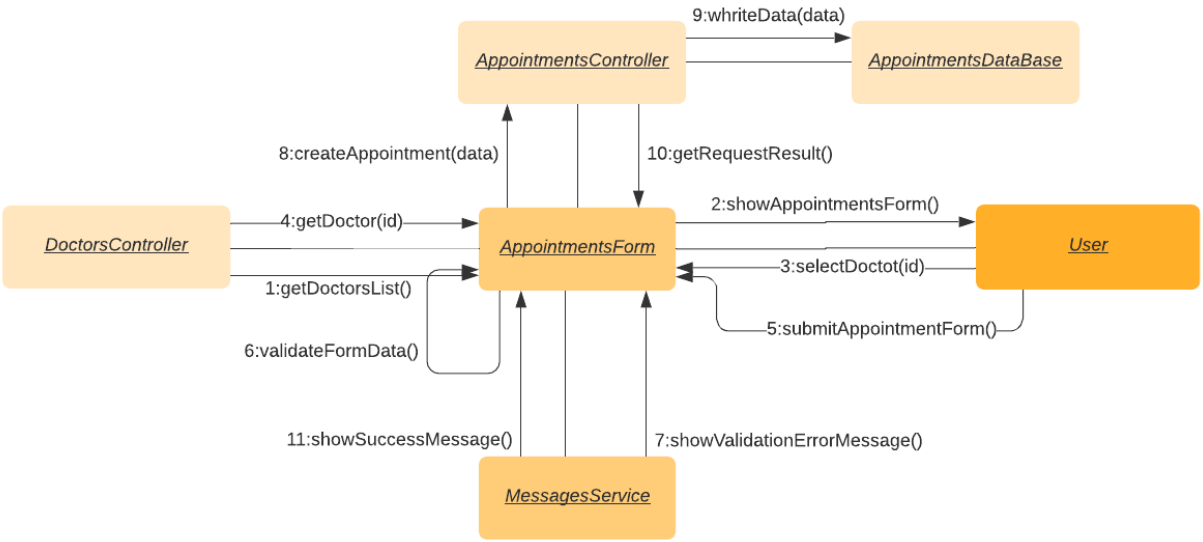

Рисунок 2.4 – Діаграма кооперацій «Запис на прийом»

Діаграма кооперацій «Запис на прийом» демонструє етапи запису користувача на прийом до лікаря (рис. 3.5). Даний сценарій починається з отримання даних про список лікарів (1), після чого відбувається відображення форми для користувача (2). Користувач обирає лікаря для запису (3 - 4) та заповнює основну інформацію в формі, після чого відправляє форму (5). Компонент форми проводить валідацію заповнених даних (6), в разі якщо данні не відповідають умовам валідації, користувач отримує відповідне повідомлення (7), у протилежному випадку данні відправляються до контролеру (8), який в свою чергу додає запис до бази даних (9). Результат виконання повертається формі (10) і користувач отримує відповідне повідомлення.

2.3 Діаграми діяльності

Діаграми діяльності використовуються, щоб проілюструвати потік контролю в системі та посилатися на кроки, пов’язані з виконанням варіанту використання. Діаграма діяльності фокусується на стані потоку та послідовності, в якій це відбувається.

Даний тип діаграм зображує потік управління від початкової точки до кінцевої точки, показуючи різні шляхи прийняття рішень, які існують під час виконання дії. Таким чином за допомогою діаграми діяльності можна зобразити як послідовну, так і одночасну обробку діяльності. Вони використовуються в моделюванні бізнесу та процесів, де їх основне використання полягає в зображенні динамічних аспектів системи.

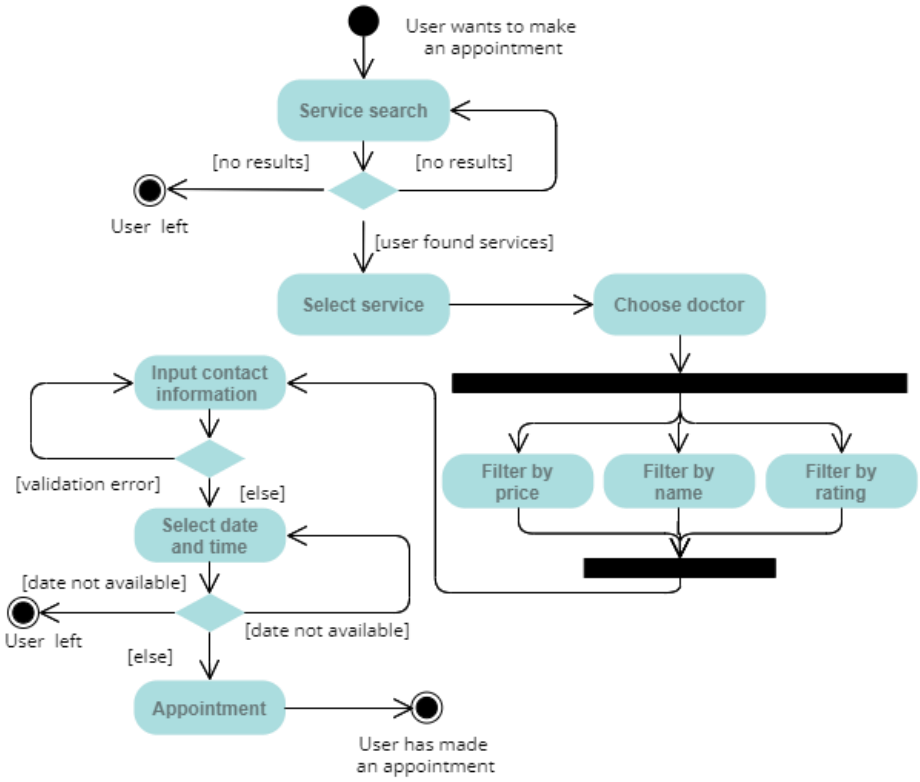

Рисунок 2.5 – Діаграма діяльності – «Запис на прийом»

На рис. 2.5 зображена модель поведінки користувача під час запису на прийом. На ній відображені основні кроки, які повинен виконати користувач для того, щоб записатися на прийом до лікаря, включаючи деякі провальні сценарії, під час яких користувач може не досягти своєї цілі. Основні етапи, які користувач повинен пройти, це вибір послуги, вибір лікаря, заповнення контактної інформації та вибір дати та часу прийому.

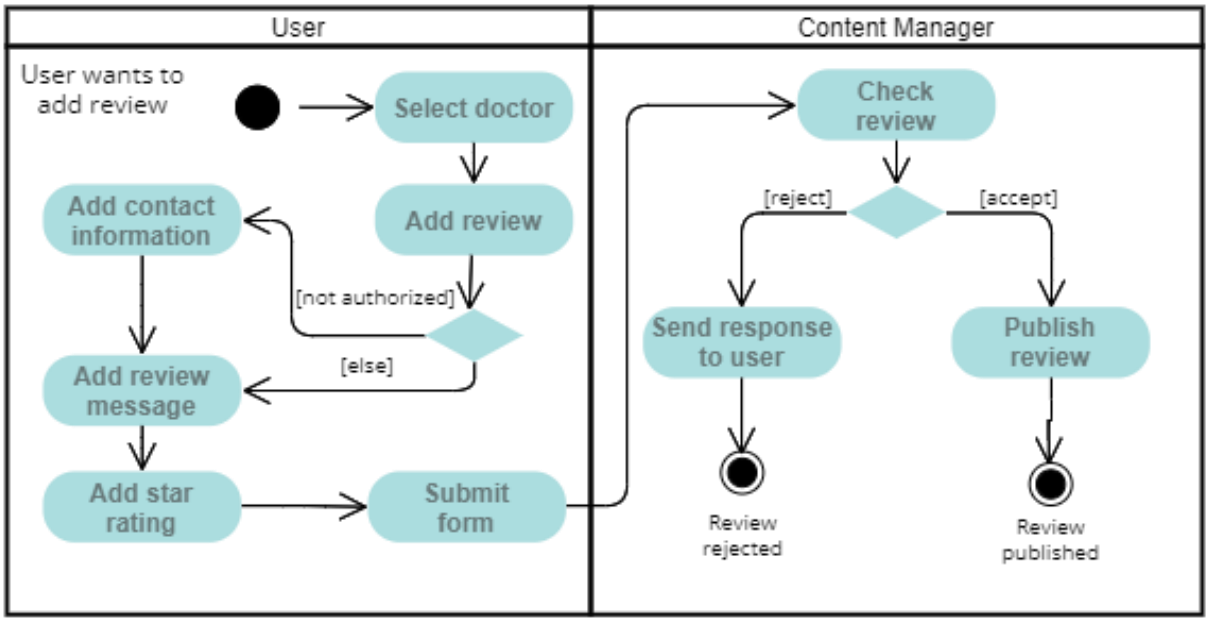

Рисунок 2.6 – Діаграма діяльності «Додавання відгуку користувачем»

Рис. 2.6 демонструє процес додавання відгуку про лікаря користувачем. Дана модель виділяється тим, що у ній для досягнення поставленої цілі приймають участь два актори – користувач, який залишає відгук, та контент-менеджер, чия задача у даному сценарії – перевірити відгук, та вирішити прийняти його чи відхилити.

Представлені діаграми використовується для моделювання діяльності користувачів у системі, яка є не чим іншим, як вимогами бізнесу. Діаграма більше впливає на розуміння бізнесу, ніж на деталі впровадження. Діаграми діяльності іноді розглядають як блок-схеми. Хоча діаграми виглядають як блок-схема, це не так. Діаграма показує різні потоки, такі як паралельні, розгалужені, одночасні та поодинокі [9].

Висновки до розділу 2

У другому розділі були розроблені проєктні рішення для розроблюваної системи, відповідно до специфікації вимог до програмного забезпечення виявленої у попередньому розділі. Проєктні рішення представлені у вигляді вербального опису функцій системи, який представлений у вигляді сценаріїв

використання та графічних моделей, серед яких: діаграми прецедентів, послідовності, концепцій та діяльності.

Моделі можуть застосовуватися для різних цілей та покривати найрізноманітніші аспекти розробки системи. Так, наприклад, одна модель може описувати загальну структуру взаємодії всередині всієї організації, а інша – відображати лише одну конкретну функціональну вимогу до цієї системи.

Найчастіше, моделі представляють собою певний візуальний ряд, у якому відображення інформації використовуються взаємозалежні діаграми. Нові методи, такі як об’єктно-орієнтовані методи моделювання, розширюють концепцію моделювання, проте більшість підходів, що використовуються в них – базуються на відомих і перевірених часом принципах.

Можна зробити висновок, що використання моделей системи дозволяє вільно спілкуватися різним організаціям між собою, використовуючи стандартні нотації. Моделювання дозволяє системному інженеру більшою мірою виявити свої творчі здібності.

3 ПРОЄКТУВАННЯ АРХІТЕКТУРИ ВЕБЗАСТОСУНКУ

3.1 Загальний огляд архітектури застосунку

Коли мова заходить за вибір архітектури для вебзастосунку, найбільш популярним є клієнт-серверна архітектура. Клієнт-серверна архітектура – це обчислювальна модель, в якій сервер розміщує, доставляє та керує більшістю ресурсів і послуг, які запитує клієнт. Вона також відома як модель мережевих обчислень або мережа клієнт-сервер, оскільки всі запити та послуги доставляються через мережу. Архітектура може мати інші системи, підключені через мережу, де ресурси розподіляються між різними комп’ютерами.

Для роботи даної архітектури потрібні три компоненти [10]:

– робочі станції або, як їх називають – клієнтські комп’ютери. Робочі станції працюють як підпорядковані серверам і надсилають їм запити на доступ до спільних файлів і баз даних. Сервер запитує інформацію від робочої станції і виконує кілька функцій як центральне сховище файлів, програм, баз даних і політик керування. Робочі станції керуються політиками, визначеними сервером;

– сервери – визначаються як пристрої швидкої обробки, які діють як централізовані сховищі мережевих файлів, програм, баз даних і політик. Сервери мають величезний простір для зберігання і надійну пам’ять для обробки кількох запитів, які надходять одночасно з різних робочих станцій. Сервери можуть виконувати багато ролей, таких як поштовий сервер, сервер баз даних, файловий сервер і контролер домену одночасно;

– мережеві пристрої – це середовище, яке з’єднує робочі станції та сервери в архітектурі клієнт-сервер. Багато мережевих пристроїв використовуються для виконання різних операцій у мережі. Наприклад, концентратор використовується для підключення сервера до різних робочих станцій. Повторювачі використовуються для ефективної передачі даних між двома пристроями. Мости використовуються для ізоляції сегментації мережі.

Архітектури клієнт-сервер має різні типи:

а) однорівнева архітектура.

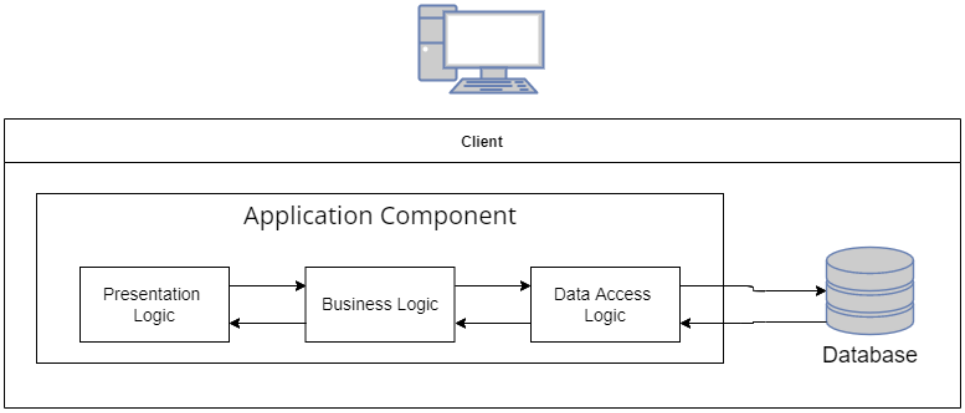

Рисунок 3.1 – Однорівнева архітектура

У даному типі (рис. 3.1), архітектура містить усі види налаштувань, таких як налаштування конфігурації та маркетингова логіка, на одному пристрої. Хоча різноманітність послуг, які пропонує однорівнева архітектура, робить її одним із надійних джерел, працювати з такою архітектурою важко. В першу чергу це пов'язано з розбіжністю даних. Часто це призводить до повторення роботи. Однорівнева архітектура складається з кількох шарів, таких як рівень презентації, бізнес-рівень і рівень даних, які об'єднані за допомогою унікального програмного пакета. Дані, присутні на цьому рівні, зазвичай зберігаються в локальних системах або на спільному диску.

б) дворівнева архітектура

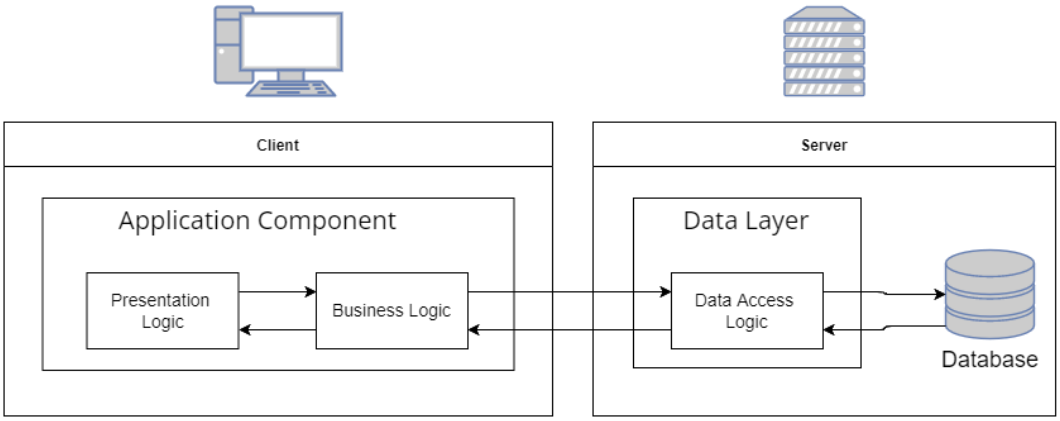

Рисунок 3.2 – Дворівнева архітектура

У даному типі архітектури (рис. 3.2), інтерфейс користувача зберігається на стороні клієнта, а база даних — на сервері, тоді як логіка бази даних і бізнес-логіка підтримується або на стороні клієнта, або на стороні сервера.

Дворівнева архітектура швидше в порівнянні з однорівневою архітектурою за рахунок того, що дворівнева архітектура не має посередника між клієнтом і сервером. Він часто використовується, щоб уникнути плутанини між клієнтами.

в) 3-рівнева архітектура

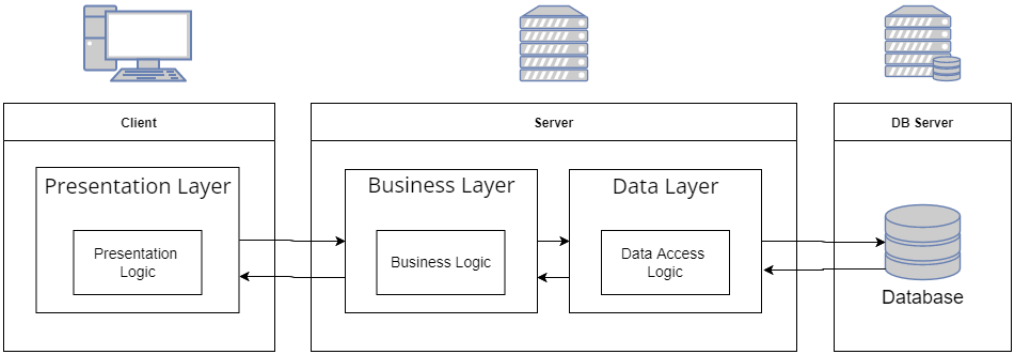

Рисунок 3.3 – Трирівнева архітектура

На відміну від дворівневої архітектури, яка не має посередника, у 3-рівневій архітектурі між клієнтом і сервером знаходиться проміжне програмне забезпечення (рис. 3.3). Якщо клієнт надсилає запит на отримання конкретної інформації від сервера, запит спочатку отримає проміжне програмне забезпечення. Потім він буде відправлений на сервер для подальших дій. Такий же шаблон буде дотримуватись, коли сервер надсилає відповідь клієнту. Структура трирівневої архітектури поділяється на три основні рівні: рівень представлення, рівень програми та рівень бази даних [11].

Усі три шари контролюються на різних кінцях. У той час як рівень представлення керується на пристрої клієнта, проміжне програмне забезпечення та сервер обробляють рівень застосунку і рівень бази даних відповідно. Завдяки наявності третього рівня, який забезпечує контроль даних, трирівнева архітектура є більш безпечною, має невидиму структуру бази даних та забезпечує цілісність даних.

При розробці вебзастосунку університетської поліклініки найкращим варіантом є використання саме трирівневневої архітектури, оскільки даний тип архітектури має ряд явних переваг у порівнянні з іншими типами:

– розвантаження сервера бази даних за рахунок виконання частини операцій, перенесених на сервер застосунку;

– зменшення розміру клієнтських програм за рахунок розвантаження їх від зайвого коду;

– спрощення налаштування клієнтів – при зміні загального коду сервера програм автоматично змінюється поведінка програм клієнтів, тобто. забезпечується єдина поведінка всіх клієнтів.

До того ж слід враховувати перспективу розширення масштабу проєкту, та збільшення кількості клієнтів застосунку, тому вибір трирівневої організації системи стає дедалі очевиднішим.

3.2 Вибір технологій реалізації застосунку

Вибір правильного стеку технологій для розробки вебзастосунку є одним з найважливіших кроків у всьому життєвому циклі проєкту. Правильний стек технологій гарантує, що продукт буде якісним і відповідатиме очікуванням клієнтів. Вибір правильного стеку технологій для веброзробки відповідно до потреб системи перед запуском запобігає величезній кількості всіх майбутніх проблем, пов’язаних із виправленням або оновленнями. Це, безумовно, може заощадити досить значну суму бюджету та часу.

Для обраної архітектури реалізації застосунку характерне розділення частин застосунку на шари, серед яких можна виділити:

– фронтенд – також відомий як клієнтська сторона, оскільки користувачі бачать і взаємодіють з цією частиною застосунку. Для вебзастосунку ця взаємодія здійснюється у веббраузері і можлива завдяки низці інструментів програмування. Вебзастосунки, орієнтовані на клієнта, зазвичай створюються за допомогою комбінації JavaScript, HTML і CSS;

– бекенд – працює поза сценою і не видимий для користувачів, це двигун, який управляє вашим додатком і реалізує його логіку. Вебсервер, який є частиною бекенда, приймає запити від браузера, обробляє ці запити відповідно до певної логіки, звертається до бази даних, якщо потрібно, і відправляє назад відповідний вміст. Бекенд складається з бази даних, застосунку написаному на мові серверного програмування та сервера;

– база даних – це організована колекція інформації. Бази даних зазвичай включають агрегації записів даних або файлів. Наприклад, при розробці застосунку для медичного закладу цими записами або файлами будуть дані про лікарів, послуги, записи на прийом, а також дані про клієнтів;

– вебсервер – комп’ютерна програма, яка розповсюджує вебсторінки в міру їх реквізиції. Основною метою вебсервера є зберігання, обробка та доставка вебсторінок користувачам. Ця взаємодія здійснюється за допомогою протоколу передачі гіпертексту (HTTP).

3.2.1 Технології реалізації клієнтської частини застосунку

На сьогоднішній день кількість технологій та підходів реалізації клієнтської частини застосунку стрімко зростає. Усі ці технології та підходи спрямовані на те, щоб зробити процес розробки максимально простим та швидким.

Але окрім значного розвитку технологій, також зростають і вимоги користувачів до програмного забезпечення. Основна частина цих вимог пов’язана з вирішенням ряду проблем, які погано впливають на користувацький досвід. Серед таких проблем можна виділити: швидкість роботи вебзастосунку, швидкість переходів між сторінками, кросбраузерність та немало важливий аспект – динамічний контент вебзастосуку. Популярним підходом для вирішення усіх перелічених проблем – є використання сучасних фрейворків та бібліотек реалізованих на JS, а також реалізація клієнтської частини у вигляді односторінкового вебзастосунку (SPA).

Основний принцип побудови архітектури SPA – увесь застосунок повинен працювати на одній сторінці. Динаміка контенту зумовлена тим, що при взаємодії користувача із системою, компоненти завантажуються за допомогою асинхронних запитів JS, з використанням таких технології як AJAX [12].

Оскільки застосунок повинен виконувати велику кількість ітерацій рендерингу компонентів, доцільно використовувати у такій архітектурі спеціалізований фреймфорк. Найкращим вибором у такому випадку є Vue.js.

Серед особливостей даного фреймворку можна виділити основні [13]:

– Vue.js це HTML-first фреймворк. Це означає, що Vue.js має багато характеристик подібних до Angular, а це, завдяки використанню різних компонентів, допомагає оптимізації HTML-блоків;

– можна використовувати як для створення односторінкових застосунків, так і для більш складних вебінтерфейсів. Важливо, що невеликі інтерактивні елементи можна легко інтегрувати до існуючої інфраструктури без наслідків;

– розмір фреймворку дорівнює приблизно 20КБ, при цьому він зберігає свою швидкість і гнучкість, що дозволяє досягти кращої продуктивності в порівнянні з іншими фреймворками;

– має велику екосистему, включаючи інструменти для управління станом та маршрутизацією застосунку.

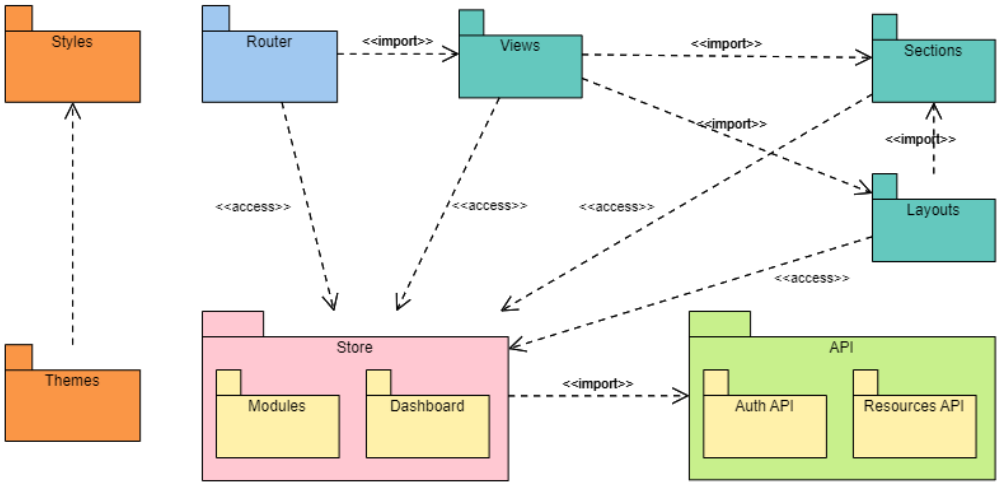

Рисунок 3.4 – Діаграма пакетів SPA

На діаграмі пакетів (рис. 3.4), відображаються основні пакети для розроблюваного вебзастосунку. На діаграмі враховані особливості архітектури SPA. Структура включає використання роутінгу, управління станом, використання API, а також декомпозицію шаблонів для можливості повторного використання окремих компонентів.

3.2.2 Технології реалізації серверної частини застосунку

Оскільки клієнтська частина застосунку використовує архітектуру SPA, на серверній частині немає необхідності реалізовувати будь-який функціонал, який відповідає за відображення. Натомість серверна частина застосунку повинна надавати публічний інтерфейс для надання можливості комунікації між клієнтом та сервером. Такий інтерфейс носить назву RESTful API.

RESTful API – це архітектурний стиль інтерфейсу, який використовує HTTP-запити для доступу та виконання різних операцій над даними. Ці дані можна використовувати для типів запитів GET, PUT, POST і DELETE, що стосується операцій читання, оновлення, створення та видалення ресурсів.

Для реалізації RESTful API, вебсервіс повинен дотримуватися наступних архітектурних обмежень [14]:

– ресурси мають бути однозначно ідентифіковані за допомогою однієї URL-адреси, і лише за допомогою базових методів мережевого протоколу, таких як DELETE, PUT і GET з HTTP, можна маніпулювати ресурсом;

– повинно бути чітке розмежування між клієнтом і сервером. Проблеми з інтерфейсом користувача та збором запитів – це домен клієнта. Доступ до даних, керування робочим навантаженням і безпека є доменом сервера. Це нещільне з’єднання клієнта і сервера дозволяє розвивати та вдосконалювати кожен незалежно від іншого;

– усі операції мають відбуватися без збереження стану, і будь-яке необхідне керування станом має здійснюватися на стороні клієнті, а не на сервері;

– усі ресурси повинні мати можливість кешування, якщо явно не вказано, що кешування неможливе;

– у більшості випадків сервер надсилає назад статичні представлення ресурсів у форматі XML або JSON. Однак у разі необхідності сервери можуть надіслати клієнтові виконуваний код.

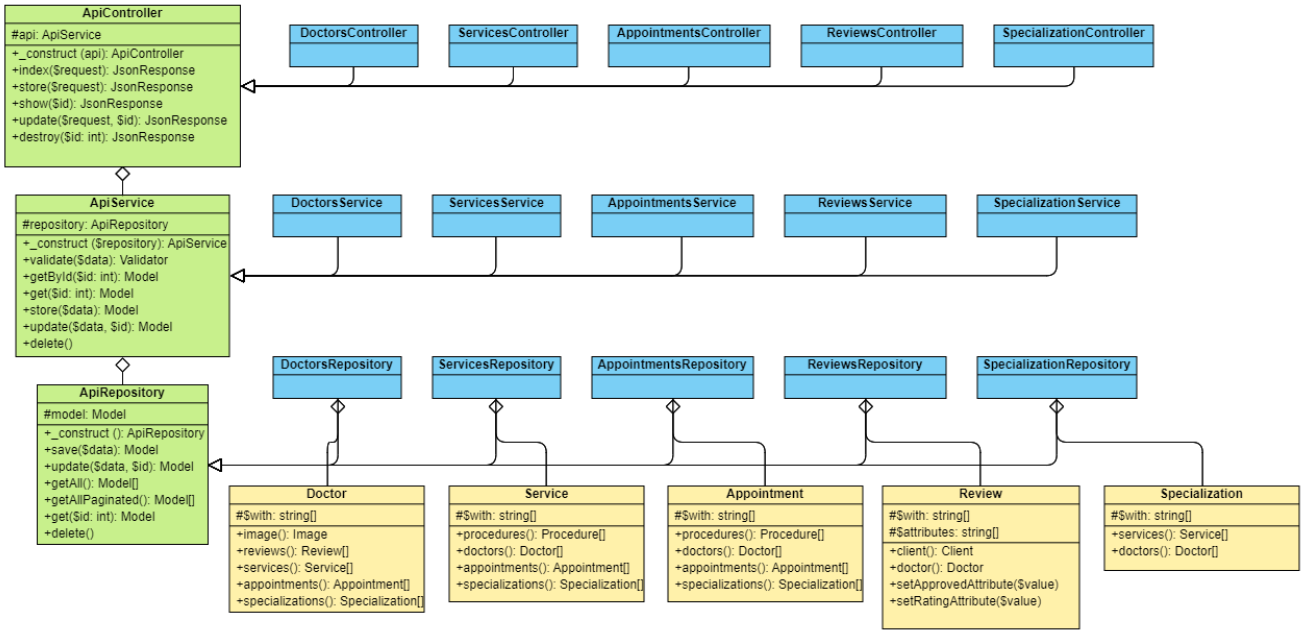

Рисунок 3.5 – Діаграма класів застосунку

На діаграмі класів (рис. 3.5) наведена базова структура класів для вебзастосунку університетської поліклініки. На діаграмі наведена лише частина основних класів. Окрім деяких класі вебзастосунку, на діаграмі також відсутні будь-які класи фреймворку.

Загалом система складається з двох основних частин: моделі та контролери. Моделі – це представлення даних у об’єктно-орієнтованому форматі. На діаграмі відображені деякі з основних моделей – Doctor, Service, Appointment, Review та Specialization. Інші класи моделей мають схожу структуру.

Що стосується контролерів, дана частина розділена на три шари:

– контролери – класи які відповідають комунікацію клієнта та сервера. Класи цієї групи приймають запити від клієнта, в залежності від типу запиту, та віддають результат для конкретного ресурсу. Усі класи контролери наслідуються від класу ApiController;

– репозиторії – класи, за допомогою яких відбувається доступ до даних. Усі операції читання та запису відбуваються у класах цієї групи. Батьківський клас ApiRepository.

– сервіси – класи проміжного шару між контролерами та репозиторіями. Дана група класів відповідає за валідацію, перехоплення та обробку помилок. Базовий клас – ApiService.

Усі три шари працюють разом. Контролер приймає запит від клієнта, та передає управління у сервіс. Тут відбувається валідація запиту, валідація успішна, керування передається до репозиторію де і відбувається доступ до даних. Після цього у зворотному порядку результат повертається клієнту.

3.2.3 Проєктування База даних

При проєктуванні бази даних важливим кроком є вибір системи керування цією базою даних. Система керування базами даних або СКБД – це тип програмного забезпечення, яке взаємодіє з самою базою даних, шляхом використання інтерфейсу користувача для отримання даних та їх аналізу. СКБД також містить ключові інструменти для керування базою даних. Усього існує два типи СКБД: реляційні та нереляційні, які також називають SQL та NoSQL відповідно [15].

Загалом існує велика кількість різноманітних СКБД. При виборі типу бази даних потрібно враховувати конкретні потреби для розроблюваного застосунку. Також потрібно враховувати характеристики СКБД із стеком обраних технологій для уникнення проблем із сумісністю. Для вебзастосунку університетської поліклініки не має досить жорстких вимог до бази даних. Що стосується стеку обраних – Laravel підтримує чотири системи баз даних: MySQL, PostgreSQL,

SQLite та SQL Server. Опираючись на цей факт можна зробити порівняльний аналіз доступних СКБД та обрати найбільш оптимальний варіант. Оскільки SQLite досить проста СКБД, її можна відразу відкинути для уникнення проблем, які можуть виникнути при зростанні розміру системи у майбутньому.

Таблиця 3.1 – Порівняння реляційних СКБД

Характеристика	**MySQL**	**SQL Server**	**PostgreSQL**
Переваги	1) безкоштовне встановлення; 2) простий синтаксис і помірна складність; 3) підтримка хмарних баз даних.	1) різноманіття версій; 2) end-to-end рішення для бізнес-даних; 3) багата документація та допомога громади; 4) підтримка хмарних баз даних.	1) масштабованість; 2) підтримка спеціальних типів даних; 3) інтеграція сторонніх інструментів; 4) повністю безкоштовна.
Недоліки	1) проблеми при масштабуванні; 2) подвійна ліцензія; 3) обмежена відповідність стандартам SQL.	1) висока вартість; 2) нечіткі та плаваючі умови ліцензії; 3) складний процес налаштування.	1) невідповідна документація; 2) відсутність інструментів звітності та аудиту.

Враховуючи переваги та недоліки доступних СКБД (табл. 3.1), очевидним лідером є PostgreSQL. До перелічених переваг також можна відзначити той факт, що PostgreSQL є не просто реляційна, а об’єктно-реляційна СКБД. Фундаментальна характеристика об’єктно-реляційної бази даних – це підтримка спеціальних об’єктів та їх поведінки, включаючи типи даних, функції, операції, домени та індекси. Завдяки складним запитам і широкому вибору користувальницьких інтерфейсів, виконаних за допомогою попередньо визначених функцій, PostgreSQL ідеально підходить для аналізу і зберігання даних.

Після виявлення цільової СКБД слідує наступний етап – моделювання бази даних. Згідно до можливостей, які пропонує PostgreSQL можна створити фізичну модель бази даних для розроблюваного вебзастосунку.

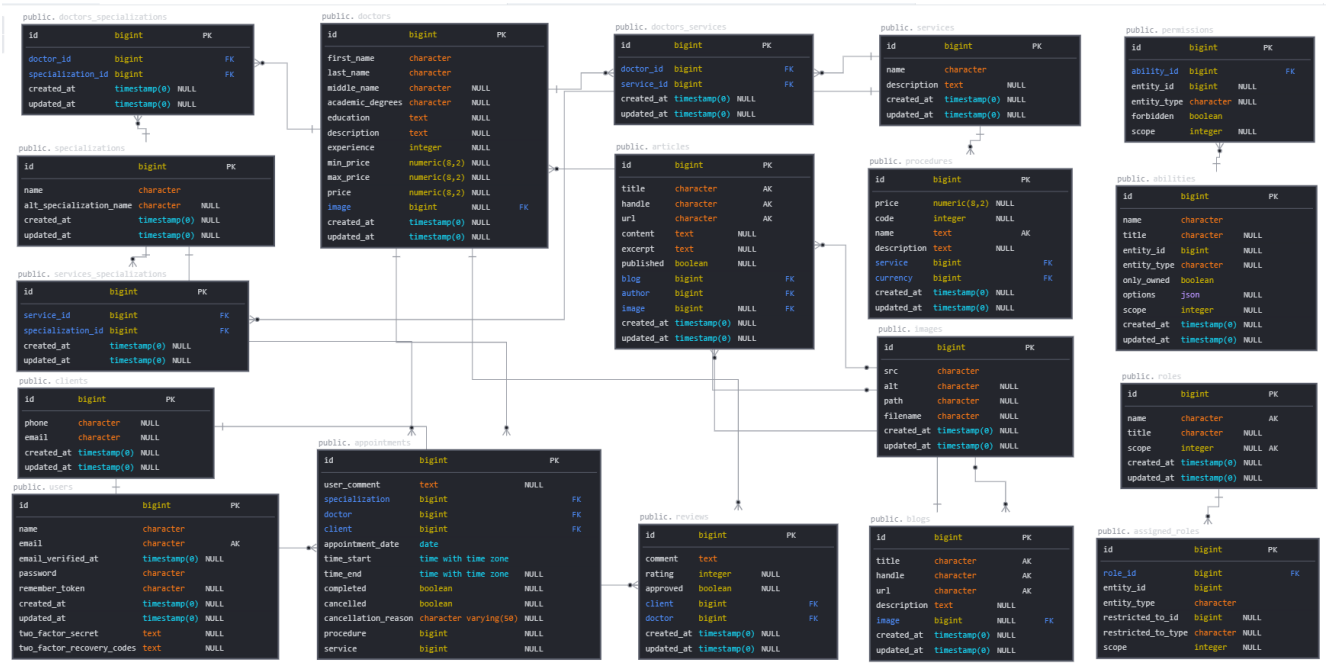

Рисунок 3.6 – Фізична модель бази даних

На фізичній моделі (рис. 3.6) зображені усі сутності та їх атрибути, а також зв’язки між ними у контексті розроблюваного програмного забезпечення.

3.2.4 Вибір вебсервера

Вибір вебсервера досить важлива частина проєктування вебзастосунку, тому даному етапу також потрібно приділити увагу. Існує кілька різних категорій, на які варто звернути увагу під час вибору сервера, наприклад, підтримка операційної системи (ОС), безпека, документація та продуктивність. Наразі існує багато різних типів серверного програмного забезпечення, але серед них можна виділити два найбільш популярних – Nginx та Apache. Це пояснюється тим, що обидва вони пропонують високу продуктивність для багатьох різних конфігурацій серверів і підходять для певних програм краще, ніж інші.

Nginx – це рішення з відкритим вихідним кодом, яке багато користувачів обирають за його стабільність і масштабованість. Це частково пов'язано з його архітектурою. Насправді, частиною мети початкового випуску Nginx було можливість обробляти 10 000 підключень одночасно. Це те, що було необхідно ще в 2004 році через швидко розширювану мережу в той час [16].

Основними перевагами Nginx є те, що даний вебсервер продуктивний, чудово працює зі статичними файлами та виконує функції балансувальника навантаження та «зворотного проксі». Усе це стосується часу роботи, швидкості та безпеки.

Apache – це спроба розробити та підтримувати HTTP-сервер з відкритим вихідним кодом для сучасних операційних систем, включаючи UNIX та Windows. Основною метою Apache є запровадження безпечного, ефективного та розширюваного сервера, який надає послуги HTTP у синхронізації з поточними стандартами HTTP [17].

Apache постачається, як попередньо встановлене програмне забезпечення у всіх дистрибутивах Linux, тому воно досить популярне для цієї ОС. Однак, незважаючи на те, що він використовує архітектуру, відмінну від Nginx, він все ще пропонує потужність, масштабованість і структуровану документацію.

Таблиця 3.2 – Порівняння вебсерверів Nginx та Apache

Параметри порівняння	**Nginx**	**Apache**
Загальна характеристика	Вебсервер з відкритим кодом, високоефективний та має можливості зворотного проксі.	Сервер з відкритим вихідним кодом, який працює через HTTP.
Продуктивність	Може одночасно обробляти запити та завантаження.	Працює повільно, коли на сервер надходить кілька запитів на завантаження, і його

		продуктивність знижується.
Пам’ять	Дуже ефективний і займає значно менше місця в пам’яті.	Займає і споживає більше пам’яті.

Кінець таблиці 3.2

Можливості потоку	Має можливість обробляти декілька з’єднань в одному потоку одночасно.	Використовує один потік як єдиний послідовний потік даних для одного з’єднання.
Платформа	Працює на всіх системах Unix, але не підтримує всі платформи Windows.	Добре працює на всіх платформах Windows і системах Unix.

З порівняльної таблиці (табл 3.1) можна побачити, що Nginx має досить великий список переваг над Apache. До того ж Nginx дуже добре працює із статичним контентом, а оскільки клієнтська сторона представлена у вигляді SPA з великою кількістю статичних файлів стилів та скриптів на JS, то Nginx є однозначним лідером під час вибору вебсерверу.

3.3 Використання сторонніх рішень

Для досить великих застосунків нормальною практикою вважається використання сторонніх засобів та готових реалізацій деяких функцій. До таких компонентів відносяться різного роду бібліотеки та плагіни. У даному розділі виділені основні сторонні рішення, які використовуються під час розробки вебзастосунку.

Laravel Sanctum

Laravel Sanctum надає просту систему аутентифікації для SPA (односторінкових застосунків), мобільних застосунків і простих API на основі JWT. Sanctum дозволяє користувачам застосунку генерувати кілька токенів API для свого облікового запису. Цим токенам можуть бути надані можливості/області, які визначають, які дії токенам дозволено виконувати [18].

Даний пакет пропонує простий спосіб аутентифікації односторінкових застосунків (SPA), які повинні взаємодіяти з API розроблених на Laravel. Sanctum використовує вбудовані в Laravel послуги аутентифікації сеансів на основі файлів cookie. Зазвичай для цього Sanctum використовує захист вебаутентифікації Laravel. Це забезпечує захист від CSRF, аутентифікації сеансу, а також захищає від витоку облікових даних через XSS. Загалом автентифікація відбувається за допомогою файлів cookie лише тоді, коли вхідний запит надходить із інтерфейсу SPA. Коли Sanctum перевіряє вхідний запит HTTP, він спочатку перевірить наявність файлу cookie аутентифікації, і у разі його відсутності, Sanctum перевірить заголовок авторизації на наявність дійсного токена API.

Laravel Fortify

Laravel Fortify – це базова реалізація функцій авторизації користувачів для Laravel. Fortify реєструє маршрути та контролери, необхідні для реалізації всіх функцій автентифікації Laravel, включаючи вхід, реєстрацію, скидання пароля, перевірку електронної пошти тощо [19].

На перший погляд може здатися, що даний пакет виконує ті ж самі функції, що і Sanctum, але ці два пакети вирішують дві різні, але пов’язані проблеми. Laravel Sanctum займається лише керуванням токенами API та автентифікацією існуючих користувачів за допомогою файлів cookie або токенів сеансу. Sanctum не надає жодних маршрутів, які обробляють реєстрацію, авторизацію користувачів тощо.

Bouncer

Bouncer – це пакет для проєктів розроблених на мові PHP, який надає зручну можливість керувати ролями та правами доступу користувачів у системі. Даний пакет досить гнучкий, і дозволяє працювати з різними моделями Eloquent.

Vuex

Vuex – це шаблон керування станом (State) та бібліотека для застосунків Vue. Він служить централізованим сховищем для всіх компонентів програми, з правилами, які гарантують, що стан можна змінювати лише передбачуваним чином.

Vuex допомагає впоратися зі спільним управлінням станом з вартістю додаткових концепцій і шаблонів. Це компроміс між короткостроковою та довгостроковою продуктивністю. Це основна ідея Vuex, натхненна Flux, Redux та The Elm Architecture. На відміну від інших шаблонів, Vuex також є реалізацією бібліотеки, розробленою спеціально для Vue, щоб скористатися перевагами його детальної системи реактивності для ефективних оновлень.

Vue Router

Однією із основних проблем при розробці односторінкових користувацьких вебінтерфейсів є маршрутизація та роутінг. Як і у випадком із керуванням станом застосунку в екосистемі Vue є модуль для вирішення даної проблеми. Vue Router є офіційним маршрутизатором для Vue.js. Він глибоко інтегрується з ядром Vue, щоб зробити створення односторінкових застосунків за допомогою Vue легким.

Tailwind CSS

Tailwind CSS – CSS-фреймворк для швидкого створення користувацьких інтерфейсів. На відміну від інших бібліотек та фреймворків на основі CSS, таких як Bootstrap, Bulma і Foundation, Tailwind CSS не має теми за замовчуванням або вбудованих компонентів інтерфейсу. Натомість він поставляється з попередньо розробленими віджетами, які можна використовувати для створення сайту з нуля.

Amazon S3

Amazon Simple Storage Service (S3) — це рішення хмарного сховища, яке надається Amazon Web Services (AWS), підрозділом гіганта електронної комерції з розподілених обчислень і вебінфраструктури [20].

Використовуючи надійну та масштабовану інфраструктуру та архітектуру зберігання об’єктів на основі ключів, Amazon S3 добре підходить для розміщення величезної кількості структурованих і неструктурованих даних у формі озер даних.

Amazon S3 надає об’єктне сховище, яке створено для зберігання та відновлення будь-якої кількості інформації чи даних з будь-якого місця через Інтернет. Він забезпечує це сховище через інтерфейс вебсервісів.

Даний сервіс чудово підходить для зберігання медіа контенту, такого як зображення або відео.

Висновки до розділу 3

У третьому розділі було спроєктовано архітектуру розроблюваного застосунку. Виявлено найбільш підходящий тип розроблюваної архітектури системи, виділені основні складові частини для реалізації системи, а також підібрані інструменти та програмні засоби для розробки системи на різних етапах.

Для розроблюваної системи обрано трирівневу архітектуру клієнт-сервер. Даний тип архітектури передбачає розділення представлення, бізнес-логіки та даних на різні шари, що дозволяє зменшити навантаження на систему, а також підвищити рівень безпеки та продуктивності системи.

Для кожного рівня архітектури системи було виявлено набір інструментів та технологій, що використовуються при розробці застосунку.

Також у даному розділі було виявлено набір сторонніх інструментів та готових рішень, які використовуються для спрощення та пришвидшення етапу розробки вебзастосунку на різних рівнях архітектури.

Проєктування архітектури системи є одним із найголовніших етапів у розробці програмного забезпечення. Її використання дозволяє значно спростити

розробку, чітко розуміти, що вийде в результаті, і як працюватимуть всі функції. Архітектура проєкту дозволяє робити якісне програмне забезпечення і надалі зменшує витрати на його утримання та обслуговування.

4 ПРОГРАМНА РЕАЛІЗАЦІЯ ВЕБЗАСТОСУНКУ

4.1 Налаштування середовища розробки

Перед початком програмної реалізації застосунку, потрібно налаштувати середовище розробки. Деякі деталі, такі як встановлення ПЗ для розробки та встановлення залежностей для застосунку у даному розділі упущено, оскільки дані налаштування є досить базовими і не потребують додаткового опису.

Як було виділено у попередньому розділі, трирівнева клієнт-серверна архітектура припускає, що різні шари можуть фізично розміщуватися на різних комп’ютерах. Оскільки при розробці програмного забезпечення використовувати декілька пристрої у більшості випадків не є можливим, таку поведінку можна емулювати за допомогою Docker.

Docker – це відкрита платформа для розробки, доставки та експлуатації програм. За допомогою docker може відокремити застосунок від інфраструктури і звертатися з інфраструктурою як керованим застосунком. У своєму ядрі docker дозволяє запускати практично будь-який застосунок, безпечно ізольований у контейнері [21]. Безпечна ізоляція дозволяє запускати на одному хості багато контейнерів одночасно.

Docker допомагає упакувати застосунок у контейнер, и цим самим надає можливість емулювати поведінку застосунку різних пристроях. Але для розроблюваної системи цього не достатньо, оскільки потрібно імітувати поведінку декількох систем. Для цього можна використати ще один інструмент – docker-compose. Docker Compose – це інструмент, який був розроблений, щоб допомогти визначати багатоконтейнерні застосунки та спільно використовувати їх. За допомогою Compose можна створити файл конфігурації YAML, щоб визначати служби системи, і за допомогою однієї команди все запустити або зупинити.

Для конфігурації застосунку, в корні проєкту створюється файл docker-compose.yml, вміст даного файлу наведено у (додаток А). У файлі docker-compose визначаються три служби:

– **db** – це визначення служби витягує образ postgres з Docker і визначає нові змінні середовища, у тому числі базу даних Laravel для застосунку а також ряд змінних для доступу до бази даних;

– **nginx** – це визначення служби використовує образ nginx:alpine з Docker та відкриває порти 8000 та 80;

– **app** – це визначення служби містить основний застосунок і запускає особистий образ Docker, pmbssu-polyclinic, код якого наведено далі.

Docker дозволяє задавати середовище всередині окремих контейнерів за допомогою файлу Dockerfile. Файл Dockerfile дозволяє створювати особисті образи. які можна використовувати для встановлення потрібного програмного забезпечення програми та зміни налаштувань відповідно до вимог. Таким чином образ для основного застосунку буде мати наступний вигляд:

```
FROM php:7.4-fpm

ARG user
ARG uid

RUN apt-get update && apt-get install -y \
    git \
    curl \
    libpng-dev \
    libonig-dev \
    libxml2-dev \
    zip \
    unzip

RUN apt-get install -y libpq-dev \
    && docker-php-ext-configure pgsql -with-pgsql=/usr/local/pgsql \
    && docker-php-ext-install pdo pdo_pgsql pgsql

RUN apt-get clean && rm -rf /var/lib/apt/lists/*

RUN docker-php-ext-install mbstring exif pcntl bcmath gd

COPY --from=composer:latest /usr/bin/composer /usr/bin/composer

RUN useradd -G www-data,root -u $uid -d /home/$user $user
RUN mkdir -p /home/$user/.composer && \
    chown -R $user:$user /home/$user
```

```
WORKDIR /var/www

USER $user
```

Спочатку Dockerfile створює образ поверх образу php:7.4-fpm Docker. Це образ на базі із встановленим екземпляром PHP FastCGI PHP-FPM. Також цей файл встановлює необхідні пакети для Laravel: pdo, pdo_pgsql, pgsql із composer.

Директива RUN визначає команди для оновлення, встановлення та налаштування параметрів усередині контейнера, включаючи виділеного користувача та групу з ім’ям www. Інструкція WORKDIR визначає каталог /var/www як робочий каталог програми.

Наступним кроком є налаштування серверу. Для налаштування Nginx потрібно створити файл app.conf із конфігурацією служб у папці ~/docker-compose/nginx/pmbssu-policlinic.conf. Лістинг конфігурації наведено нижче:

```
# disable any limits to avoid HTTP 413 for large image uploads
client_max_body_size 0;

server {
    listen 80;
    index index.php index.html;
    error_log  /var/log/nginx/error.log;
    access_log /var/log/nginx/access.log;
    root /var/www/public;
    client_max_body_size 0;
    location ~ \.php$ {
        try_files $uri =404;
        fastcgi_split_path_info ^(.+\.php)(/.+)$;
        fastcgi_pass app:9000;
        fastcgi_index index.php;
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        fastcgi_param PATH_INFO $fastcgi_path_info;
    }
    location / {
        try_files $uri $uri/ /index.php?$query_string;
        gzip_static on;
    }
}
```

Серверний блок визначає конфігурацію вебсервера Nginx за допомогою наступних директив:

– **listen** – визначає порт, що прослуховується для вхідних запитів;

– **error_log** та **access_log** – визначають файли для запису журналів.

– **root** – визначає шлях до кореневої папки, формуючи повний шлях для будь-якого запитаного файлу в локальній файловій системі.

У блоці розташування php директива fastcgi_pass вказує, що служба app прослуховує сокет TCP на порту 9000. З нею сервер PHP-FPM виконує прослуховування через мережу, а не через сокет Unix. Хоча сокет Unix дає невелику перевагу у швидкості в порівнянні з сокетом TCP, у нього немає мережевого протоколу і він пропускає мережевий стек. У разі розташування хостів в одній системі використання сокету Unix може мати сенс, але якщо служби працюють на різних хостах, сокет TCP дає перевагу, дозволяючи підключатися до розподілених служб. Оскільки контейнери app та nginx працюють на різних хостах, у нашій конфігурації ефективніше використовувати сокет TCP.

Що стосується контейнеру db – даний контейнер використовує офіційний образ postgres із Docker Hub. Для бази даних додаткових налаштувань не потрібно, за виключенням налаштування деяких змінних середовища у файлі .env, що використовуються для доступу до бази:

```
DB_CONNECTION=pgsql
DB_HOST=pmbssu-polyclinic-db
DB_PORT=5432
DB_DATABASE=pmbssu_polyclinic
DB_USERNAME=pmbssu_admin
DB_PASSWORD=admin
```

Тепер налаштування середовища завершено і можна приступати до подальшої розробки.

4.2 Реалізація серверної частини вебзастосунку

4.2.1 Створення моделей та міграцій

Моделі та міграції у Laravel це основні сутності для взаємодії з базою даних. Міграції – це сутності для маніпулювання процесом створення та видалення таблиць у базі даних. Вони дозволяють керувати процесом створення та внесення змін у схему таблиць бази даних шляхом написання програмного коду на PHP.

Таким чином не залежно від типу бази даних, створення таблиць для застосунку будуть виглядати майже однаково для всіх.

Моделі, на відміну від міграцій, використовуються для взаємодії з цією таблицею. Окрім отримання записів із таблиці бази даних, моделі також дозволяють вставляти, оновлювати та видаляти записи з таблиці.

Для створення переважної більшості сутностей у Laravel використовується інтерфейс командного рядка Artisan. З його допомогою створити міграцію можна за допомогою однієї команди – *make:migration*. Таким чином створення міграції для сутності «Запис на прийом» буде мати наступний вигляд (рис. 4.1):

```
PS D:\Projects\PMBSSU-Polyclinic\pmbssu-polyclinic> php artisan make:migration create_appointment_table
Created Migration: 2022_06_17_202447_create_appointment_table
```

Рисунок 4.1 – Створення міграції для таблиці «appointments» за допомогою Artisan

Після виконання даної команди створюється клас міграції, який наслідується від базового класу усіх міграцій Laravel – Migration. Клас міграції містить два методи: *up* і *down*. Метод *up* використовується для додавання нових таблиць, стовпців або індексів до бази даних, тоді як метод *down* обертає операції, що виконуються методом *up*. Для сутності «Запис на прийом» лістинг коду методу *up* наведено нижче.

```
public function up()
    {
        Schema::create('appointments', function (Blueprint $table) {
            $table->id();
            $table->text('user_comment')->nullable();
            $table->date('appointment_date');
            $table->time('time_start', $precision = 0);
            $table->time('time_end', $precision = 0)->nullable();
            $table->boolean('completed')
                ->nullable()
                ->default(false);
            $table->boolean('cancelled')
                ->nullable()
                ->default(false);
            $table->string('cancellation_reason', 255)->nullable();

            $table->foreignId('client')
                ->constrained('clients', 'id')
                ->cascadeOnDelete();

            $table->bigInteger('doctor')->nullable();
            $table->foreign('doctor')
```

```
                ->references('id')
                ->on('doctors')
                ->nullOnDelete();

            $table->bigInteger('specialization')->nullable();
            $table->foreign('specialization')
                ->references('id')
                ->on('specializations')
                ->nullOnDelete();

            $table->bigInteger('procedure')->nullable();
            $table->foreign('procedure')
                ->references('id')
                ->on('procedures')
                ->nullOnDelete();

            $table->bigInteger('service')->nullable();
            $table->foreign('service')
                ->references('id')
                ->on('services')
                ->nullOnDelete();

            $table->timestamps();
        });
    }
```

Метод *up* демонструє процес створення таблиці «appointments». У ньому додаються усі поля таблиці, а також індекси, первинний та зовнішні ключі, які відповідають спроєктованій у попередньому розділі фізичній моделі бази даних. Для виконання даного методу міграції використовується команда artisan *migrate*. Для того, щоб вказати конкретну міграцію, потрібно також передати ключ *--path* та у якості значення вказати шлях до міграції (рис 4.2).

```
PS D:\Projects\PMBSSU-Polyclinic\pmbssu-polyclinic>  php artisan migrate --path=/database/migrations/2022_06_17_202447_create_appointment_table.php
Migrating: 2022_06_17_202447_create_appointment_table
Migrated:  2022_06_17_202447_create_appointment_table (398.26ms)
```

Рисунок 4.2 – Виконання міграції

Метод *down* навпаки видаляє таблицю, в тому випадку, якщо таблиця існує у базі даних:

```
public function down()
{
    Schema::dropIfExists('appointments');
}
```

Даний метод виконується при відкаті міграції. Відкат відбувається схожим чином, для цього використовується команда *migrate:rollback* (рис. 4.3).

```
PS D:\Projects\PMBSSU-Polyclinic\pmbssu-polyclinic>  php artisan migrate:rollback --path=/database/migrations/2022_06_17_202447_create_appointment_table.php
Rolling back: 2022_06_17_202447_create_appointment_table
Rolled back:  2022_06_17_202447_create_appointment_table (216.06ms)
```

Рисунок 4.3 – Відкат міграції за допомогою команди migrate:rollback

Схожим чином до створення міграції, створюються і класи для моделей. За допомогою Artisan це можна зробити однією командою – *make:model* (рис. 4.4).

```
PS D:\Projects\PMBSSU-Polyclinic\pmbssu-polyclinic> php artisan make:model Appointment
```

Рисунок 4.4 – Створення моделі сутності «Запис на прийом» за допомогою Artisan

Моделі в Laravel подібно до міграцій також мають спільний батьківський клас – Model. У більшості випадків для класів моделей для повноцінного функціонування окрім вказання у якості базового класу Model додаткових налаштувань не потрібно. Але для більш розширених варіантів використання слід додати додаткові поля та методи. Таким чином клас моделі сутності «Запис на прийом» матиме наступний вигляд:

```php
<?php

namespace App\Models;

class Appointment extends Model
{
    use HasFactory;

    protected $with = ['client', 'procedure', 'doctor', 'specialization'];

    protected $attributes = [
        'completed' => false
    ];

    public function setCompletedAttribute($value)
    {
        $this->attributes['completed'] = is_null($value) ? false : $value;
    }

    public function client(): BelongsTo
    {
        return $this->belongsTo(Client::class, 'client');
    }

    public function procedure(): BelongsTo
    {
        return $this->belongsTo(Procedure::class, 'procedure');
    }

    public function doctor(): BelongsTo
    {
        return $this->belongsTo(Doctor::class, 'doctor');
```

```
    }

    public function specialization(): BelongsTo
    {
        return $this->belongsTo(Specialization::class, 'specialization');
    }
}
```

Поле *$with* у класі Appointment використовується для зазначення зв'язків, які потрібно завантажувати при доступі до моделі. Це означає, що при звернені до моделі замість вказаних ідентифікаторів будуть до моделі будуть приєднані інші моделі, які відповідають вказаним зв'язкам.

Захищене поле *$attributes* використовується для зазначення значень за замовчуванням для вказаних атрибутів моделі.

Метод *setCompletedAttribute* відіграє роль сетеру для атрибуту completed. З його допомогою можна додати додаткову перевірку при записі значення поля completed.

Методи *client*, *procedure*, *doctor* та *specialization* додані для вказання відношень між моделлю Appointment та іншими моделями, відповідно до схеми таблиці. В даному випадку метод *belongsTo* вказує на відношення з моделями Client, Procedure, Doctor та Specialization.

Наприклад модель Doctor має зв'язок із моделлю Appointment один до багатьох. Таким чином визначення відношення у моделі Doctor буде мати наступний вигляд:

```
public function appointments(): HasMany
{
    return $this->hasMany(Appointment::class);
}
```

Загалом більшість міграцій та моделей мають більш менш схожу структуру, тому вони не будуть детально розглядатися.

4.2.2 Налаштування автентифікації застосунку

Для створення базового функціоналу автентифікації для вебзастосунку використовуються два пакети Laravel – Sunctum та Fortify.

Для використання Sanctum для автентифікації SPA, слід додати проміжне програмне забезпечення Sanctum до групи проміжного програмного забезпечення API у файлі app/Http/Kernel.php застосунку:

```
'api' => [
    \Laravel\Sanctum\Http\Middleware\EnsureFrontendRequestsAreStateful::class,
],
```

Наступним кроком потрібно додати можливість автентифікації користувачів SPA. Для цієї функції Sanctum використовує вбудовані в Laravel послуги аутентифікації сеансів на основі файлів cookie. Такий підхід до аутентифікації забезпечує переваги захисту CSRF, аутентифікації сеансу, а також захищає від витоку облікових даних аутентифікації через XSS.

По-перше, потрібно вказати, з яких доменів SPA надсилатиме запити. Ці домени вказуються за допомогою параметра конфігурації з визначенням стану у файлі конфігурації sanctum:

```
'stateful' => explode(',', env('SANCTUM_STATEFUL_DOMAINS', sprintf(
    '%s%s',
    'localhost,localhost:3000,localhost:8000,127.0.0.1,127.0.0.1:8000,::1',
    env('APP_URL') ? ','.parse_url(env('APP_URL'), PHP_URL_HOST) : ''
))),
```

Далі потрібно переконатися, що конфігурація CORS застосунку повертає заголовок Access-Control-Allow-Credentials зі значенням True. Цього можна досягти, встановивши параметр supports_credentials у файлі конфігурації config/cors.php значення true:

```
'supports_credentials' => true,
```

На цьому налаштування Sanctum завершуються. Наступним кроком потрібно налаштувати Fortify. У першу чергу потрібно увімкнути функції які будуть використовуватись у застосунку. Файл конфігурації fortify містить масив конфігурації функцій. Цей масив визначає, які серверні маршрути та функції Fortify надаватиме за замовчуванням. Для даного застосунку будуть додані всі доступні функції – реєстрація, скидання паролю, підтвердження електронної пошти, оновлення паролю та даних користувача, а також двофакторну автентифікацію:

```
'features' => [
    Features::registration(),
    Features::resetPasswords(),
    Features::emailVerification(),
    Features::updateProfileInformation(),
    Features::updatePasswords(),
    Features::twoFactorAuthentication([
        'confirmPassword' => true,
    ]),
],
```

Для базового використання, даних налаштувань достатньо. Після цього можна протестувати функції автентифікації. Оскільки на даному етапі, ще не має можливості використовувати користувацький інтерфейс, автентифікацію можна протестувати за допомогою Postman.

Спершу потрібно додати pre-request скрипт, який буде відправляти запит на отримання XSRF-токену для автентифікації через файли cookies (рис. 4.5).

Params Authorization Headers (11) Body Pre-request Script Tests Settings

```
1  pm.sendRequest({
2      url: 'http://localhost:8000/sanctum/csrf-cookie',
3      method: 'GET',
4      header: {
5          'Accept': 'application/json'
6      }
7  }, function (error, response, { cookies }) {
8      if (!error) {
9          pm.environment.set('xsrf-token', cookies.get('XSRF-TOKEN'))
10     }
11 })
12
13
```

Рисунок 4.5 – Додавання скрипту попереднього запиту отримання XSRF-токену

Отриманий токен потрібно передавати у заголовку X-XSRF-TOKEN для подальших запитів (рис. 4.6).

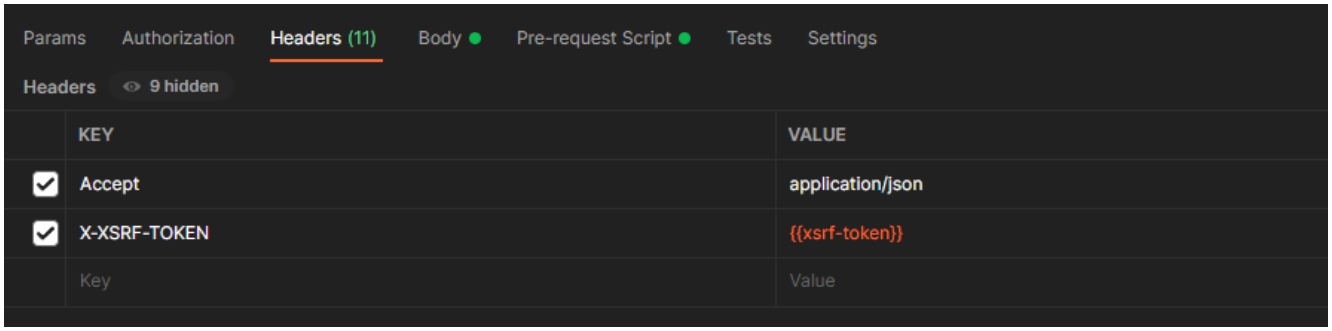

Рисунок 4.6 – Заголовки для запитів автентифікації

Тепер можна перевірити коректність роботи API автентифікації. Спершу потрібно зареєструвати тестового користувача (рис. 4.7).

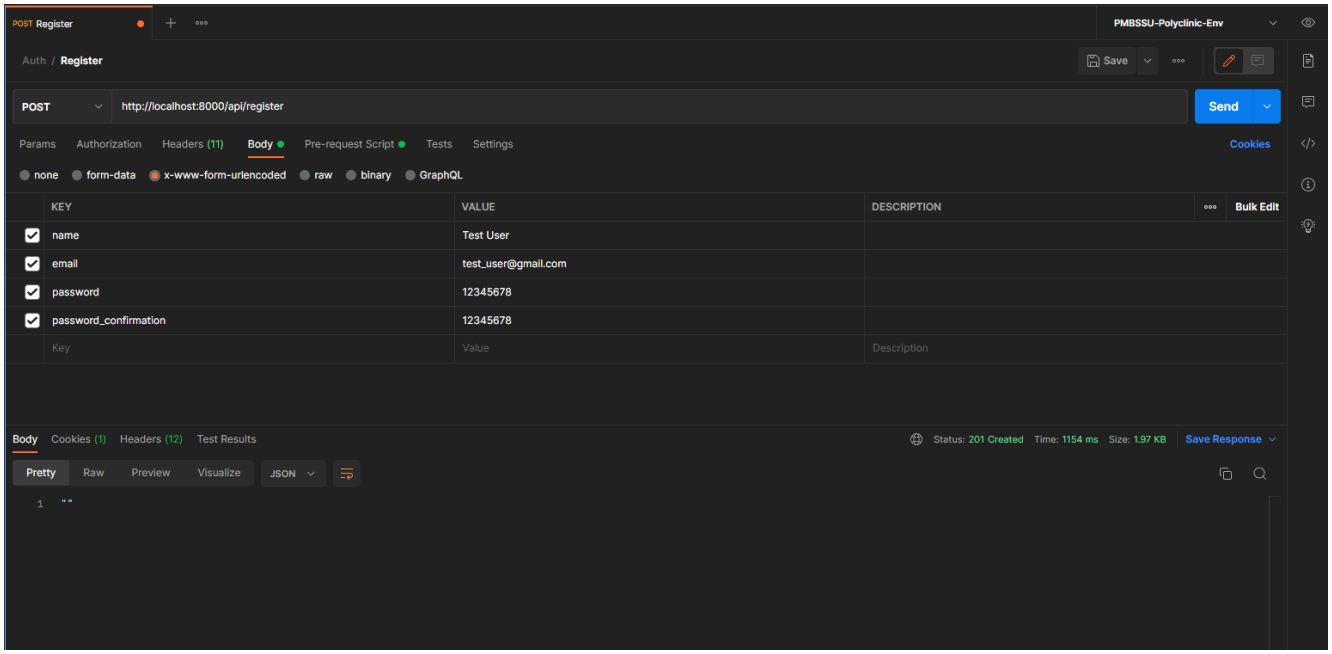

Рисунок 4.7 – Тестовий запит на створення користувача

Запит на реєстрацію користувача виконався зі статусом 201, що підтверджує факт створення нового облікового запису. Тепер потрібно авторизуватися під створеним користувачем (рис. 4.8).

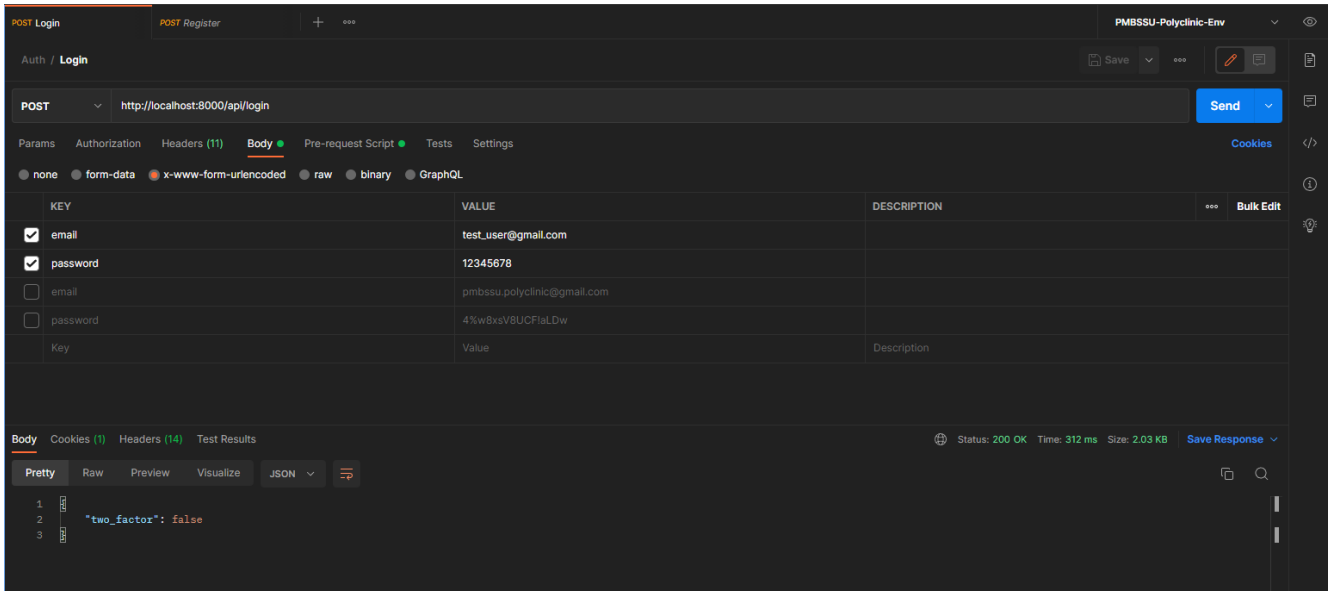

Рисунок 4.8 – Тестовий запит на авторизацію користувача.

Обидва запити виконані успішно. Результати підтверджують коректність роботи функцій автентифікації застосунку.

4.2.3 Додавання прав доступу та ролей користувачів

Для керування правами доступу та ролями користувачів у системі використовується пакет Bouncer. Bouncer – це пакет, який не залежить від фреймворків, та дозоляє керувати правами за допомогою моделей Eloquent.

Єдиним налаштуванням для даного пакету є додавання спеціального трейту HasRolesAndAbilities до моделі User:

```
class User extends Model
{
    use HasRolesAndAbilities;
}
```

Права доступу потрібно додати до всіх основних сутностей у системі. У кожної сутності повинні бути права на різні CRUD операції. Для того щоб не описувати ці права для кожної сутності можна створити метод *generateAbility* для генерації цих прав:

```
private function generateAbility(string $ability_target): array
{
    return [
        'read' => Bouncer::ability()->firstOrCreate([
```

```
            'name' => "read_$ability_target",
            'title' => "Read $ability_target",
        ]),
        'create' => Bouncer::ability()->firstOrCreate([
            'name' => "create_$ability_target",
            'title' => "Create $ability_target",
        ]),
        'update' => Bouncer::ability()->firstOrCreate([
            'name' => "update_$ability_target",
            'title' => "Update $ability_target",
        ]),
        'delete' => Bouncer::ability()->firstOrCreate([
            'name' => "delete_$ability_target",
            'title' => "Delete $ability_target",
        ])
    ];
}
```

У методі *generateAbility* для генерації дозволів використовується метод *ability* фасаду Bouncer. Тепер за допомогою методу *generateAbility* можна створити основні права доступу для основних сутностей:

```
$appointments    = $this->generateAbility('appointments');
$articles        = $this->generateAbility('articles');
$authors         = $this->generateAbility('authors');
$blogs           = $this->generateAbility('blogs');
$doctors         = $this->generateAbility('doctors');
$images          = $this->generateAbility('images');
$procedures      = $this->generateAbility('procedures');
$reviews         = $this->generateAbility('reviews');
$services        = $this->generateAbility('services');
$specializations = $this->generateAbility('specializations');
$users           = $this->generateAbility('users');
```

Додатково потрібно створити право доступу для панелі адміністратора. Якщо користувач має даний дозвіл, то він зможе потрапити до панелі адміністратора:

```
$can_access_to_dashboard = Bouncer::ability()->firstOrCreate([
    'name' => "access_to_dashboard",
    'title' => "Access to dashboard",
]);
```

Наступним кроком потрібно створити ролі, та застосувати до них відповідні дозволи. Для ролі «Адміністратор» код виглядатиме наступним чином:

```
$admin = Bouncer::role()->firstOrCreate([
    'name' => 'admin',
    'title' => 'Administrator',
]);

Bouncer::allow($admin)->to($can_access_to_dashboard);
Bouncer::allow($admin)->to(array_values($appointments));
Bouncer::allow($admin)->to(array_values($articles));
Bouncer::allow($admin)->to(array_values($authors));
```

```
Bouncer::allow($admin)->to(array_values($blogs));
Bouncer::allow($admin)->to(array_values($doctors));
Bouncer::allow($admin)->to(array_values($images));
Bouncer::allow($admin)->to(array_values($procedures));
Bouncer::allow($admin)->to(array_values($reviews));
Bouncer::allow($admin)->to(array_values($services));
Bouncer::allow($admin)->to(array_values($specializations));
```

Для створення нової ролі використовується метод фасаду Bouncer – *role*, а для застосування дозволів до ролі – метод *allow*. Схожим чином створюється роль «Контент-Менеджер».

```
$manager = Bouncer::role()->firstOrCreate([
    'name' => 'manager',
    'title' => 'Manager',
]);

Bouncer::allow($manager)->to($can_access_to_dashboard);
Bouncer::allow($manager)->to(array_values($articles));
Bouncer::allow($manager)->to(array_values($authors));
Bouncer::allow($manager)->to(array_values($blogs));
Bouncer::allow($manager)->to(array_values($images));
Bouncer::allow($manager)->to(array_values($reviews));
```

Заключним етапом налаштування ролей системи є створення користувача із роллю «Адміністратор». Для цього буде використано клас типу seeder. Класи даного типу використовуються для генерації записів у базі даних програмним способом, використовуючи для цього лише одну команду – *db:seed*. В більшості випадків такі класи використовуються для внесення у базу даних тестових даних, але це чудовий спосіб створення обов’язкових записів, які повинні бути у базі даних застосунку. Клас seeder за замовчуванням містить лише один метод: *run*. Цей метод викликається, коли виконується команда *db:seed*:

```
class AdminUserSeeder extends Seeder
{
    public function run()
    {
        $user = User::create([
            'name' => 'Admin',
            'email' => 'pmbssu.polyclinic@gmail.com',
            'email_verified_at' => now(),
            'password' => Hash::make('4%w8xsV8UCF!aLDw'),
            'remember_token' => Str::random(10)
        ]);

        Bouncer::assign('admin')->to($user);
    }
}
```

Після запуску команди *db:seed* у базі даних створюється обліковий запис користувача з роллю адміністратор (рис. 4.9).

	1
id	1
name	Admin
email	pmbssu.polyclinic@gmail.com
email_verified_at	2022-05-11 18:46:04
password	$2y$10$yZsOdHEXxziO79bsl2UNzOC6LTuRtMjCoCY8x874c7cIVRfi6m8j2
remember_token	OAIMYZ9THrj6MVKHMlCV9FaELDR8Xah5wgyUod4peyHvjqYePDExo8kxKUZU
created_at	2022-05-11 18:46:04
updated_at	2022-05-11 18:46:04
two_factor_secret	<null>
two_factor_recovery_codes	<null>

Рисунок 4.9 – Обліковий запис адміністратора у базі даних

Для звичайних користувачів буде використовуватися роль за замовчуванням, без жодних прав доступу. Незважаючи на це звичайні користувачі зможуть переглядати контент сайту, а також записуватися на прийом, але такі користувачі не зможуть зашкодити цілісності системи, наприклад спробувавши відредагувати або видали дані про якусь сутність за допомогою API. Тому використання ролей гарантує безпечне використання системи.

4.2.4 Реалізація API

Для реалізації RESTfull API у Laravel потрібно розробити контролери та маршрути. Контролери можуть групувати пов’язану логіку обробки запитів в один клас. Наприклад, клас UserController може обробляти всі вхідні запити, пов’язані з користувачами, включаючи показ, створення, оновлення та видалення користувачів. Для API доречно використовувати контролери ресурсів. Даний тип контролерів відрізняється тим, що він запроваджує основні маршрути для CRUD операцій. Усі контролери у системі наслідуються від базового класу ApiController (додаток Б). Даний клас реалізує узагальнену для усіх ресурсів поведінку. Таким чином при створенні нового контролеру для реалізації базового функціоналу потрібно лише передати до конструктору базового класу сервіс для того чи

іншого ресурсу. Наприклад контролер AppointmentController буде мати наступний вигляд:

```
class AppointmentController extends ApiController
{
    public function __construct(AppointmentService $apiService)
    {
        parent::__construct($apiService);
    }
}
```

До конструктору передається екземпляр класу AppointmentService. Даний клас представляє з себе проміжний шар між контролером, який приймає запити та репозиторієм. Про останній буде описано нижче. Загалом класи сервісів беруть на себе функції валідації та обробки помилок. Кожен сервіс відповідає за конкретний ресурс у системі. Усі сервіси мають спільний базовий клас ApiService. Код даного класу наведено нижче:

```
abstract class ApiService
{
    protected ApiRepository $repository;
    public function __construct(ApiRepository $repository)
    {
        $this->repository = $repository;
    }

    abstract protected function validate($data);

    public function getAll($params)
    {
        $sort  = $params->sort;
        $order = $params->order;
        $limit = $params->limit;
        $paginated = $params->paginated;

        if (filter_var($paginated, FILTER_VALIDATE_BOOLEAN)) {
            return $this->repository->getAllPaginated($limit, $sort, $order);
        }

        return $this->repository->getAll($limit, $sort, $order);
    }

    public function getById($id)
    {
        if (!$id) {
            throw MissingParameter::create('id');
        }

        return $this->repository->get($id);
    }

    public function store($data)
    {
        $validator = $this->validate($data);
```

```
        if ($validator->fails()) {
            throw new InvalidArgumentException($validator->errors()->first());
        }

        return $this->repository->save($data);
    }

    public function update($data, $id)
    {
        $validator = $this->validate($data);

        if ($validator->fails()) {
            throw new InvalidArgumentException($validator->errors()->first());
        }

        return $this->repository->update($data, $id);
    }

    public function delete($id)
    {
        return $this->repository->delete($id);
    }
}
```

Сервіси містять базові методи для CRUD операцій. В більшості випадків для сервісів достатньо поведінки яку впроваджує базовий клас. За єдиним виключенням. Кожен сервіс повинен реалізувати абстрактний метод *validate*. Даний метод відповідає за валідацію ресурсу, і оскільки усі моделі мають різні атрибути, кожен сервіс повинен визначити яким чином перевіряти свій ресурс. Наприклад для класу AppointmentService метод *validate* матиме наступний вигляд:

```
protected function validate($data)
{
    return Validator::make($data, [
        'appointment_date' => 'required|date',
        'time_start' => 'required',
        'cancellation_reason' => 'nullable|max:255',
        'client' => 'nullable|exists:App\Models\Client,id',
        'doctor' => 'nullable|exists:App\Models\Doctor,id',
        'specialization' => 'nullable|exists:App\Models\Specialization,id',
        'procedure' => 'nullable|exists:App\Models\Procedure,id',
        'service' => 'nullable|exists:App\Models\Service,id',
    ]);
}
```

На наступному рівні знаходяться класи репозиторіїв. На даному рівні відбувається взаємодія із базою даних через моделі. Відповідно до контролерів та сервісів, репозиторії також мають один спільний клас ApiRepository:

```
abstract class ApiRepository
```

```
{
    protected static int $DEFAULT_PAGINATION_LIMIT = 50;
    protected static string $DEFAULT_SORT_COLUMN = 'id';
    protected static string $DEFAULT_SORT_ORDER = 'asc';
    protected Model $model;

    public function __construct(string $class)
    {
        $this->model = new $class;
    }

    abstract public function save($data);
    abstract public function update($data, $id);

    public function getAll(?int $limit = null, ?string $sort = null, ?string
$order = null)
    {
        $limit = $limit ?? self::$DEFAULT_PAGINATION_LIMIT;
        $sort  = $sort ?? self::$DEFAULT_SORT_COLUMN;
        $order = $order ?? self::$DEFAULT_SORT_ORDER;

        return DB::table($this->model->getTable())
            ->orderBy($sort, $order)
            ->paginate($limit)
            ->items();
    }

    public function getAllPaginated(?int $limit = null, ?string $sort = null,
?string $order = null)
    {
        $limit = $limit ?? self::$DEFAULT_PAGINATION_LIMIT;
        $sort  = $sort ?? self::$DEFAULT_SORT_COLUMN;
        $order = $order ?? self::$DEFAULT_SORT_ORDER;

        return DB::table($this->model->getTable())
            ->orderBy($sort, $order)
            ->paginate($limit);
    }

    public function get($id) { return $this->model::where('id', $id)-
>firstOrFail(); }

    public function delete($id)
    {
        $record = $this->model::find($id);

        if (!$record) {
            abort(404, "Not Found");
        }

        $record->delete();

        return $record;
    }
}
```

Репозиторій містить основні методи для виконання CRUD операцій над моделлю, а також ряд статичних констант – *$DEFAULT_PAGINATION_LIMIT*, *$DEFAULT_SORT_COLUMN* та *$DEFAULT_SORT_ORDER* які відповідають за

параметри пагінації та сортування за замовчуванням. Також клас містить екземпляр моделі, через який і здійснюється доступ до бази даних.

Усі репозиторії повинні реалізувати два методи – *save* та *update*, з тієї ж причини, що і у випадку з сервісами. Таким чином клас AppointmentRepository буде виглядити наступним чином:

```
class AppointmentRepository extends ApiRepository
{
    public function __construct()
    {
        parent::__construct(Appointment::class);
    }

    public function save($data)
    {
        $appointment = new $this->model;

        $appointment->user_comment        = $data['user_comment'] ?? null;
        $appointment->appointment_date    = $data['appointment_date'];
        $appointment->time_start          = $data['time_start'];
        $appointment->time_end            = $data['time_end'] ?? null;
        $appointment->completed           = $data['completed'] ?? null;
        $appointment->cancelled           = $data['cancelled'] ?? null;
        $appointment->cancellation_reason = $data['cancellation_reason'] ?? null;
        $appointment->client              = $data['client'] ?? null;
        $appointment->doctor              = $data['doctor'] ?? null;
        $appointment->specialization      = $data['specialization'] ?? null;
        $appointment->procedure           = $data['procedure'] ?? null;
        $appointment->service             = $data['service'] ?? null;

        $appointment->save();
        return $appointment->fresh();
    }
    public function update($data, $id)
    {
        $appointment = $this->model::find($id);

        if (array_key_exists('user_comment', $data))
            $appointment->user_comment = $data['user_comment'];
        if (array_key_exists('completed', $data))
            $appointment->completed = $data['completed'];
        if (array_key_exists('cancelled', $data))
            $appointment->cancelled = $data['cancelled'];
        if (array_key_exists('cancellation_reason', $data))
            $appointment->cancellation_reason = $data['cancellation_reason'];
        if (array_key_exists('doctor', $data))
            $appointment->doctor = $data['doctor'];
        if (array_key_exists('service', $data))
            $appointment->service = $data['service'];

        $appointment->update();

        return $appointment;
    }
}
```

Заключним етапом розробки API є налаштування маршрутів. Маршрути, які використовуються у якості API оголошуються у файлі routes/api.php. В Laravel можна згенерувати маршрути для ресурсів за допомогою метода *apiResource*. Таким чином будуть згенеровані лише необхідні для API ресурсів маршрути. Загалом маршрути мають наступний вигляд:

```
Route::get('/abilities/current', [AbilityController::class, 'current']);

Route::middleware(['auth:sanctum'])->group(function () {
    Route::get('/me', function () {
       return Auth::user();
    });

    Route::get('/abilities', [AbilityController::class, 'index']);
    Route::get('/abilities/{user}', [AbilityController::class, 'show']);
    Route::patch('/abilities/{user}', [AbilityController::class, 'update']);

    Route::apiResource('images', ImageController::class);
    Route::apiResource('authors', AuthorController::class);
    Route::apiResource('blogs', BlogController::class);
    Route::apiResource('articles', ArticleController::class);
    Route::apiResource('services', ServiceController::class);
    Route::apiResource('specializations', SpecializationController::class);
    Route::apiResource('procedures', ProcedureController::class);
    Route::apiResource('doctors', DoctorController::class);
    Route::apiResource('clients', ClientController::class);
    Route::apiResource('reviews', ReviewController::class);
    Route::apiResource('appointments', AppointmentController::class);
});
```

Даний лістинг включає усі маршрути ресурсів, та також декілька додаткових маршрутів для визначення прав доступу користувача. Основна частина маршрутів захищена за допомогою Sanctum для впровадження додаткового рівня захисту.

4.3 Реалізація клієнтської частини вебзастосунку

При розробці клієнтської частини застосунку використовується фреймворк Vue.js. У даному розділі не будуть зачеплені такі аспекти розробки, як реалізація IU елементів та компонентів користувацького інтерфейсу, включаючи шаблони та стилі, оскільки дана частина не несе інформаційної цінності с точки зору розробки системи.

4.3.1 Реалізація клієнтського API

На стороні клієнта API розділені на два різних типи – API ресурсів та API автентифікації. Перші використовуються для доступу ресурсів системи, другі для реалізації функцій, які надає Laravel Fortify. Для кожного із двох типів створено окремі класи які реалізують основну логіку здійснення запитів до сервера.

API ресурсів реалізовані у класі ApiResource:

```
class ApiResource extends Api {
    constructor(resource, requestConfig = {}) {
        super(requestConfig);
        this._api = this._api + resource;
    }
    async create(data) {
        return super.post(this._api, data).then(response => response.data);
    }

    async get(id) {
        return super.get(this.generateResourceUrlForInstance(id))
            .then(response => response.data);
    }

    async update(id, data) {
        return super.patch(
            this.generateResourceUrlForInstance(id), data).then(response =>
response.data);
    }

    async delete(id) {
        return super.post(this.generateResourceUrlForInstance(id))
            .then(response => response.data);
    }
}
```

Інтерфейс даного класу має сходу структуру з інтерфейсом класів контролерів. Такий підхід було використано для того, щоб реалізацію API в різних частинах застосунку можна було легко зіставити.

Для використання класу ApiResource достатньо створити новий екземпляр, та при створенні обов’язково потрібно передати в конструктор назву ресурсу, для якого будуть виконуватись запити. Наприклад для ресурсу «appointments» створення екземпляру класу буде мати наступний вигляд:

```
const appointmentsApi = new ApiResource("appointments");
```

У свою чергу для реалізації API автентифікації створено клас Auth:

```
export class Auth extends Api {
    constructor(configs = {}) { super(configs); }
```

```
    async csrf () {
        return this.get(endpoints.CSRF);
    }

    async login (data) {
        return this.csrf().then(() => {
            return this.post(endpoints.LOGIN, data);
        });
    }

    async register (data) {
        return this.csrf().then(() => {
           return this.post(endpoints.REGISTER, data);
        });
    }

    async logout () {
        return this.csrf().then(() => {
            return this.post(endpoints.LOGOUT);
        });
    }
}
```

Інкапсуляція логіки виконання запитів до серверу у окремі класи надає можливість позбавитись від потенційних проблем, які можуть виникнути при внесені змін у API на стороні сервера. Таким чином достатньо додати зміни у одному місці, щоб застосувати їх у всьому застосунку.

Оскільки даний функціонал винесений у окремий модуль, також з’являється можливість повторного використання коду. Представлені класи доступні з будь-якої частини застосунку, тому усі підсистеми можуть використовувати даний функціонал.

4.3.2 Управління станом застосунку

Складність великих застосунків нерідко зростає через розподіл шматочків стану за багатьма компонентами та зв’язками між ними. Для вирішення цієї проблеми, Vue пропонує Vuex – бібліотеку управління станом.

Спершу потрібно створити контейнер стану. Для цього Vuex спеціальну функцію *createStore*:

```
import { createStore } from "vuex";
import { modules } from "./modules";

export default createStore({
    modules: {
        ...modules,
```

```
    }
});
```

У даному прикладі використовуються модулі для описання стану. Для кожного типу ресурсу реалізований окремий модуль, але як і у випадку із ресурсами у серверній частині застосунку – усі ресурси мають схожий базовий набір характеристик, тому на примітивному рівні модулі будуть більш-менш однаковими. Для уникнення дублювання великої кількості коду, створено фабричний метод для створення модулів стану застосунку (додаток В). З урахуванням цього, код створення модулю «appointments» буде мати наступний вигляд:

```
import { createModule } from "../module-factory";
import { appointmentsApi } from "../../../api/apiResources/appointments";

export default createModule(appointmentsApi);
```

Кожен модуль виділений у окремий файл, для уникнення плутанини. Але даний підхід залишається не досить зручним у використанні, тому що модулів у системі може буди велика кількість, і кожен з них доведеться підключати в ручну. Дану проблему можна вирішити, додавши скрипт, який буде компанувати усі модулі, та передаватиме їх до функції створення стану застосунку:

```
const requireContext = require.context('./', true, /.*\.js$/)


const modules = requireContext.keys()
    .map(file =>
        [file.replace(/(^.\/)|(\.js$)/g, ''), requireContext(file)]
    )
    .reduce((modules, [name, module]) => {
        const splitPath = name.split("/");

        if (splitPath.length <= 1) {
            return modules;
        }

        let [moduleName] = splitPath;

        if (module.namespaced === undefined) {
            module.namespaced = true
        }

        module = {
            ...module,
            ...module.default
        };
```

```
        delete module.default;

        moduleName = moduleName
            .split("-")
            .map((part, index) =>
                index ? part[0].toUpperCase() + part.substring(1).toLowerCase() :
part
            )
            .join("");

        return { ...modules, [moduleName]: module }
    }, {});

export {modules};
```

Тепер замість того, щоб імпортувати кожен модуль ресурсу окремо, достатньо імпортувати об’єкт *modules*, який автоматично підвантажить усі модулі ресурсів. Після цього потрібно підключити об’єкт стану при створені застосунку Vue, за допомогою методу *use*:

```
import { createApp } from "vue";
import store from "./store";

import App from "./App.vue";

createApp(App)
  .use(store)
  .mount("#app");
```

Перевірити стан застосунку можна за допомогою розширення Vue для браузеру. Для цього потрібно відкрити інструменти розробника (F12), відкрити розширення та перейти у вкладку vuex (рис. 4.10).

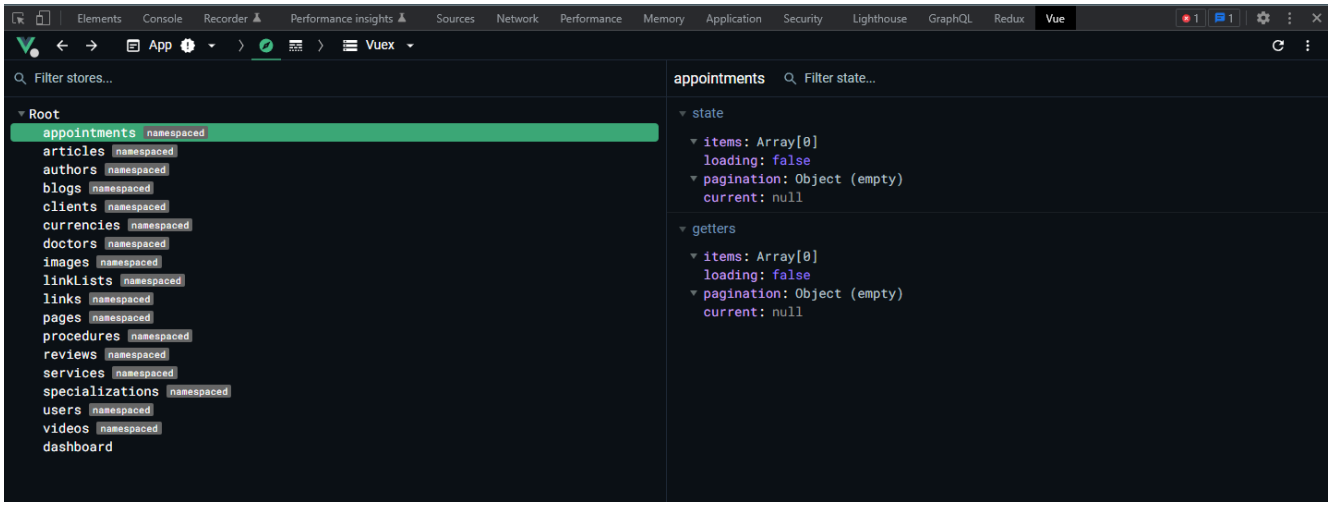

Рисунок 4.10 – Перевірка стану застосунку за допомогою розширення браузера

На рис. 4.10 можна побачити, що стан застосунку поділений на різні модулі, кожен з яких відповідає окремому ресурсу. Також можна побачити початковий стан для модуля «appointments».

4.3.3 Налаштування маршрутизації

Однією з найпотужніших функцій сучасних односторінкових вебзастосунків (SPA) є маршрутизація. Сучасні односторінкові застосунки, такі як програма Vue, можуть переходити зі сторінки на сторінку на стороні клієнта (без запиту на сервер). Vue Router – офіційна бібліотека для навігації сторінками в застосунках створених з використанням Vue.

Подібно до управління станом, спочатку потрібно створити об’єкт маршрутів застосунку. Для цього Vue Router надає функцію *createRouter*. Окрім цього потрібно вказати модифікацію для зберігання історії браузеру. Для цього рекомендується використовувати функцію *createWebHistory*, яка також надається бібліотекою. Дана функція використовується для перемикання з використання хешування в режим історії у браузері за допомогою History API HTML5.

```
import {createRouter, createWebHistory} from "vue-router";

import Dashboard from "../layouts/Dashboard";
import Website from "../layouts/Website";

import dashboard from "./dashboard";
import website from "./website";


const routes = [
    {
        path: '/',
        components: {
            default: Website,
        },
        children: website
    },
    {
        path: '/admin',
        components: {
            default: Dashboard,
        },
        children: dashboard
    }
];

export default createRouter({
```

```
    mode: 'history',
    history: createWebHistory(process.env.BASE_URL),
    routes
});
```

Наступним кроком потрібно визначити маршрути застосунку. Далі наведені деякі основні маршрути:

```
export default [
    {
        path: '',
        name: 'index',
        component: Index,
    },
  {
    path: 'contact-us',
    name: 'contact',
    component: () => import('../../views/Contact')
  },
  {
    path: 'doctors/:specialization',
    name: 'doctors',
    component: () => import('../../views/Doctors')
  },
  {
    path: 'doctor/:id',
    name: 'doctor',
    component: () => import('../../views/Doctor')
  },
  {
    path: 'book-appointment',
    name: 'book-appointment',
    component: () => import('../../views/BookAppointment')
  },
  {
    path: 'services',
    name: 'services',
    component: () => import('../../views/Services')
  },
];
```

Тут визначають шляхи маршрутів, назви а також вказується компонент, який повинен бути відображений для того чи іншого маршруту. Для деяких маршрутів, таких як «doctor» та «doctors» використовуються параметри. Для першого параметр «id» визначає ідентифікатор лікаря, інформацію про якого потрібно відобразити. Для маршруту «doctors» параметр «specialization» вказує спеціальність за якою відображається список лікарів.

Після завершення налаштування маршрутів, об’єкт потрібно включити в застосунок:

```
import { createApp } from "vue";
import store from "./store";
import router from "./router";
```

```
import App from "./App.vue";

createApp(App)
  .use(store)
  .use(router)
  .mount("#app");
```

Перевірити коректність роботи маршрутів можна таким самим способом, як і стан застосунку – за допомогою розширення браузера Vue, але замість вкладки vuex потрібно обрати routes (рис. 4.11).

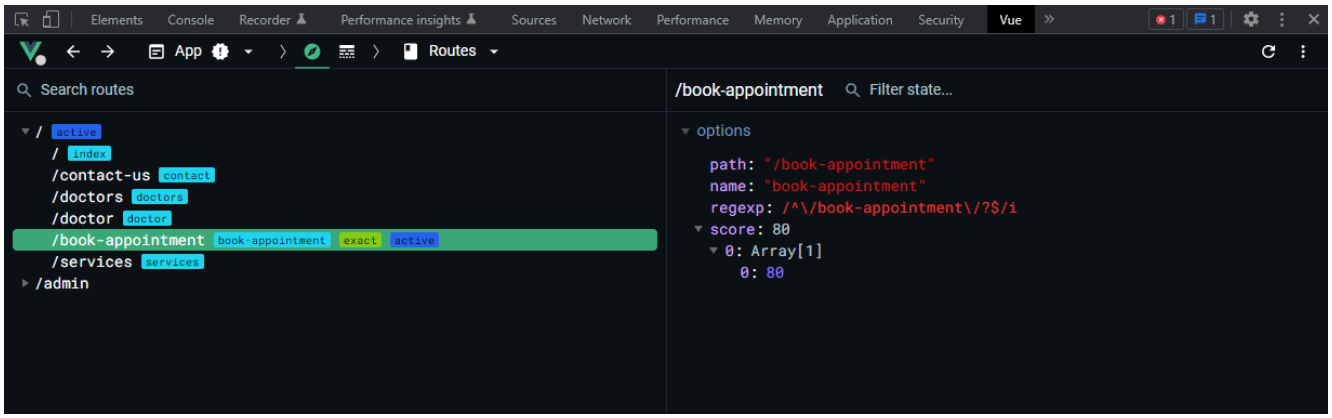

Рисунок 4.11 – Перевірка роботи маршрутизації застосунку за допомогою розширення браузера

За допомогою розширення можна також перевірити інформацію про доступні маршрути, активний маршрут, а також подивитися детальну інформацію про кожен маршрут застосунку.

4.4 Огляд користувацького інтерфейсу

Користувацький інтерфейс є однією з найважливіших частин вебзастосунку. Інтерфейс повинен бути інтуїтивно зрозумілим для користувача, лаконічним та, що не мало важливо привабливим. Далі описується інтерфейс основних сторінок вебзастосунку.

Головна сторінка

Найперше, що бачить користувач зайшовши на сайт – головна сторінка застосунку (рис. 4.12). Дана сторінка є дуже важливою, оскільки вона повинна одразу показати користувачу які функції пропонує система.

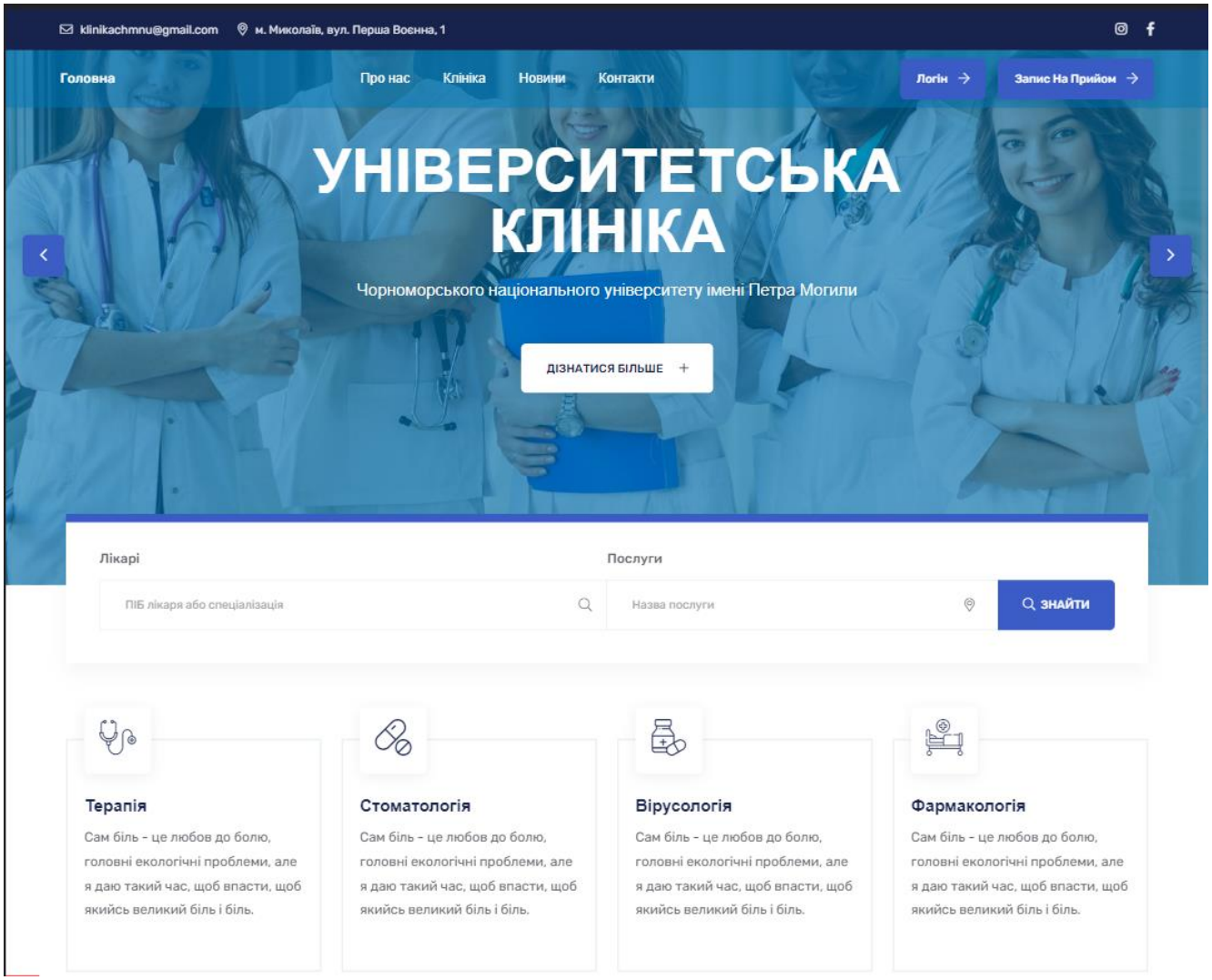

Рисунок 4.12 – Головна сторінка вебзастосунку

Головна сторінка вебзастосунку починається з банеру із закликом до дії. Нижче відразу під банером присутня форма для пошуку. За допомогою даної форми здійснюється пошук лікарів або послуг, які пропонує сервіс. Після того, як користувач введе запит та натисне кнопку «Знайти» в правій частині секції, користувач буде перенаправлений на сторінку списку лікарів або послуг, залежно від типу пошуку.

Нижче під формою пошуку знаходиться секція із списком послуг, які пропонує сервіс. Натиснувши на один з блоків користувач потрапить на сторінку з детальною інформацією про обрану послугу.

На сторінці також присутня секція, яке несе як інформаційне значення так і заклик до дії (рис. 4.13). Тут наведена інформація про те як влаштований процес запису на прийом у застосунку, а також кнопка із пропозицією записатися на прийом до лікаря.

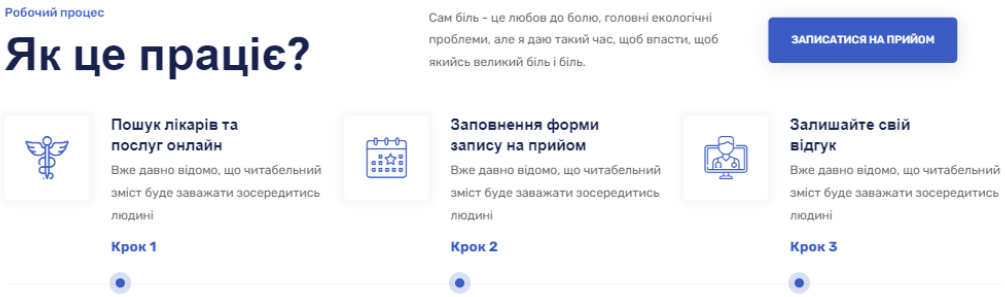

Рисунок 4.13 – Секція з інформацією про процес запису на прийом до лікаря

Сторінка списку лікарів

На даній сторінці відображається список лікарів (рис. 4.14). Список лікарів включає базову інформацію про лікарів, таку як ПІБ, спеціалізація, досвід роботи, дні прийому та рейтинг на основі користувацьких відгуків. Також на картці лікаря присутня кнопка заклику до дії «Дізнатися більше», при натисненні на якій відбувається перенаправлення користувача на сторінку із детальною інформацією про лікаря.

З лівого боку від списку лікарів розташована панель для фільтрації лікарів. Список лікарів можна відфільтрувати на спеціалізацією, рейтингом та ПІБ.

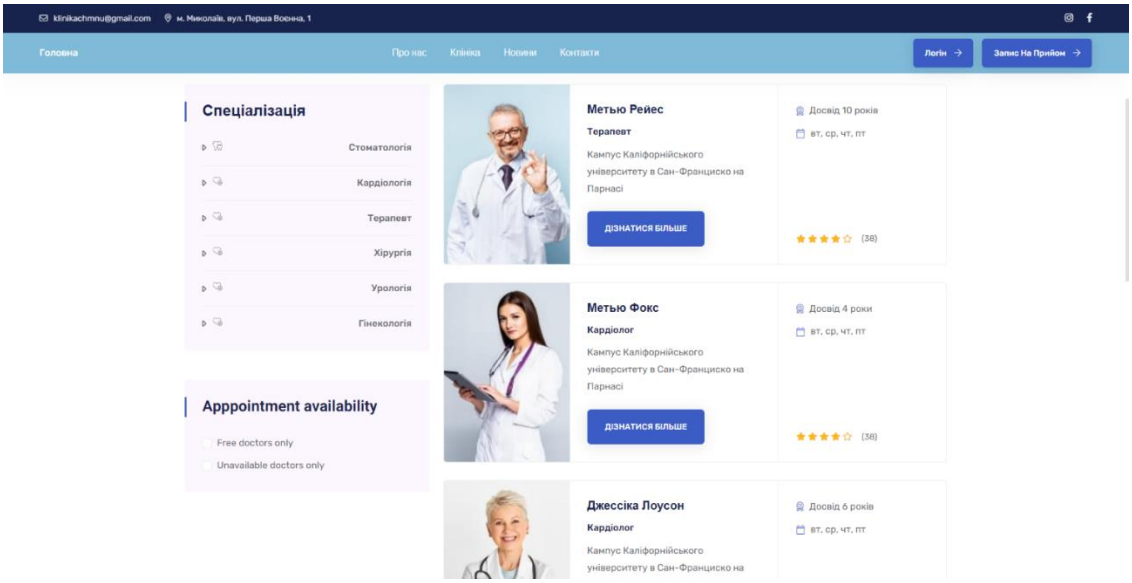

Рисунок 4.14 – Сторінка списку лікарів

Сторінка інформації про лікаря

На сторінці детальної інформації про лікаря відображається більш розширена версія інформації ніж на попередній сторінці, контактні дані, а також список послуг обраного лікаря (рис. 4.15). Також дана сторінка містить користувацькі відгуки та форму додавання нового відгуку.

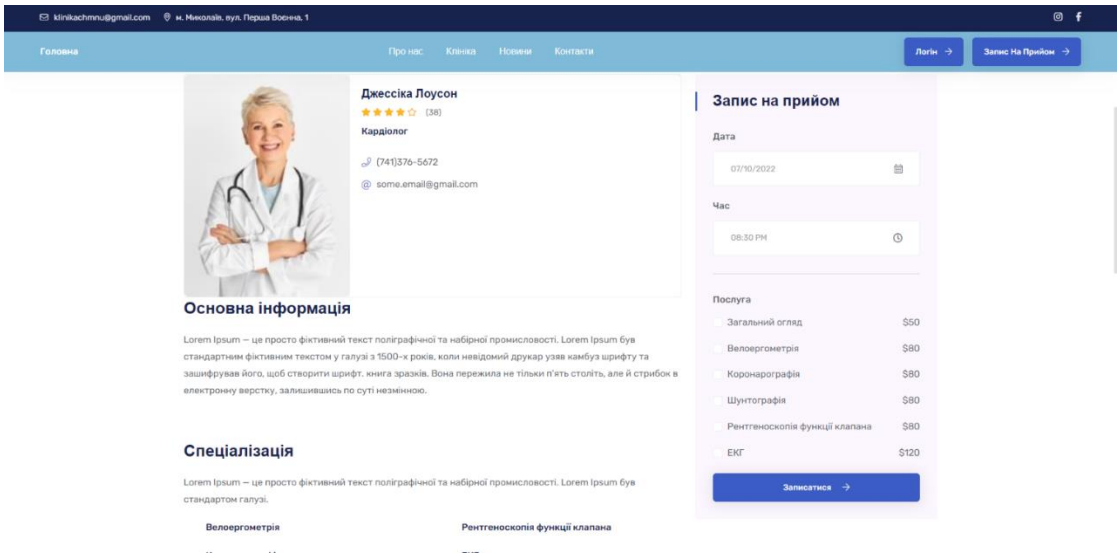

Рисунок 4.15 – Сторінка детальної інформації про лікаря

На правій частині сторінки розташована форма із закликом записатися на прийом до поточного лікаря. Форма включає в себе вибір дати та часу запису, а також список доступних для поточного лікаря послуг. Дана форма не створює

запис, після відправки форми користувач потрапляє на сторінку записку на прийом, куди автоматично заповнюються данні введені користувачем на сторінці лікаря.

Сторінка запису на прийом до лікаря

Дана сторінка є кінцевою сторінкою потоку запису на прийом. До неї можна потрапити різними способами, наприклад через сторінку лікаря або натиснувши на кнопці «записатися на прийом» в шапці сайту.

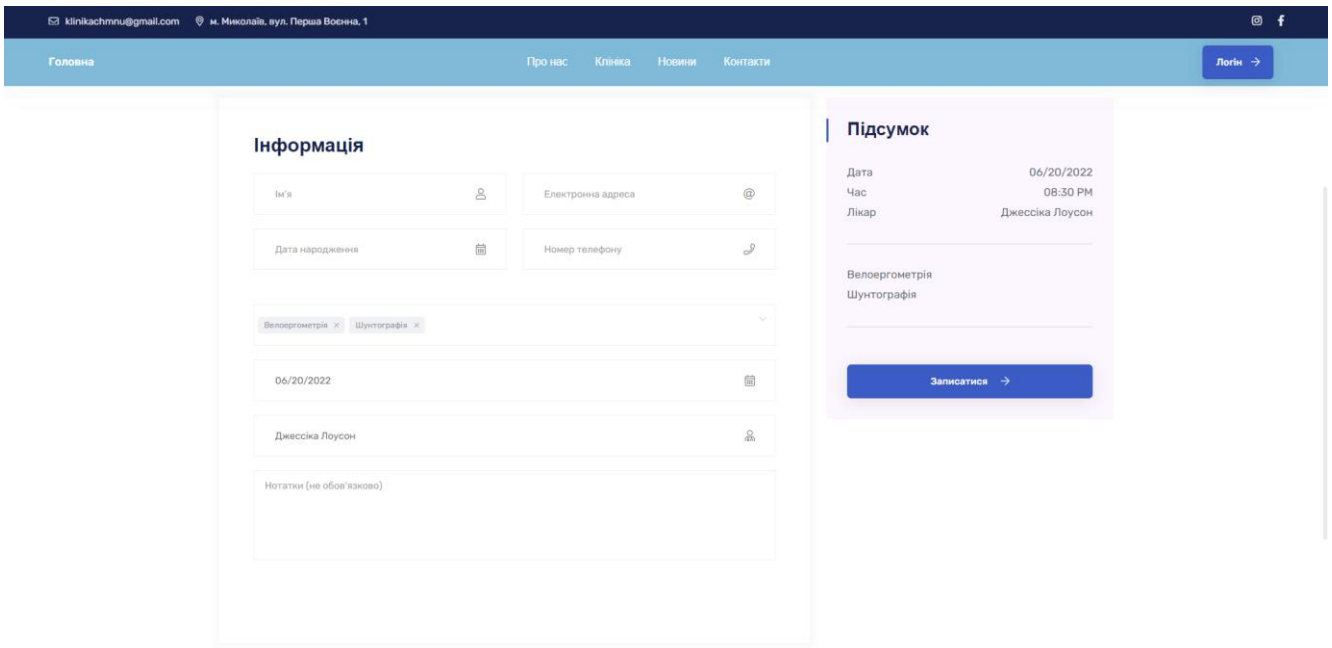

Рисунок 4.16 – Сторінка запису на прийом до лікаря

Сторінка містить форму для бронювання запису на прийом. У формі присутні поля для контактної інформації користувача, а також поля для інформації про послуги та лікаря. З правого боку знаходиться блок із підсумком. Тут відображається введена користувачем інформація. Інформація автоматично оновлюється при внесенні змін у формі. Нижче знаходиться кнопка «Записатися» при натисненні на якій відбувається створення нового запису.

Висновки до розділу 4

В ході виконання четвертого розділу був детально описаний етап програмної реалізації системи. Програмна реалізація включає в себе налаштування середовища розробки, реалізацію серверної та клієнтської частин. Вербальний опис етапу реалізації доповнений лістингами програмного коду, та зображеннями.

Реалізація є послідовним процесом створення вихідних кодів застосунку обраною мовою програмування, тестування та налагодження програмного забезпечення. Часто при створенні застосунку переважна частина часу йде не на її розробку, а на налагодження та тестування. Тому програма має бути наочною, легко читаною, супроводжуватися коментарями.

Також у розділі було продемонстровано користувацький інтерфейс системи. Опис зовнішнього вигляду вебзастосунку подано у вигляді керівництва користувача з описанням основних сторінок та UI елементів.

Результатом виконання даного розділу стало реалізоване програмне забезпечення для університетської поліклініки. Програмне забезпечення представляє з себе вебзастосунок побудований на архітектурах клієнт-сервер та SPA. Розроблене програмне забезпечення відповідає усім поставленим у першому розділі вимогам, та створено відповідно із спроєктованими моделями розробленими у другому та третьому розділах.

ВИСНОВКИ

При підготовці кваліфікаційної роботи бакалавра було встановлено основну мету роботи – забезпечити рішення для створення та подальшого управління системою університетської поліклініки за рахунок розробки програмного забезпечення з вебінтерфейсом.

Для досягнення поставленої мети було вирішено ряд задач пов’язаних з аналізом, проєктуванням на розробкою:

– аналіз предметної області та методологій розробки;

– аналіз аналогічних систем та на основі отриманих результатів побудувати діаграму використання та описати у вигляді сценаріїв із використанням шаблонів;

– виявлення специфікації вимог до системи;

– проєктування архітектури системи, та огляд сучасних інструментів розробки програмного забезпечення;

– проєктування бази даних системи;

– розробка, тестування та розгортання застосунку.

В ході виконання кваліфікаційної роботи були проаналізовані предметна область, а також поточний стан ринку у даній предметній області. Під час аналізу були виявлені основні конкуренти у сфері надання медичних послуг онлайн, для кожного були виявлені основні переваги та недоліки. На основі отриманих результатів був виявлений ряд вимог, яких повинна дотримуватись подібного роду система. Також були проаналізовані сучасні підходи та технології розробки вебзастосунків.

Під час вирішення задач пов’язаних з проєктуванням були побудовані моделі різних типів, що відображають основні концепції системи. Моделі побудовані для демонстрації різних аспектів системи, від поведінки при взаємодії користувача з системою до архітектурних рішень для побудови системи. Процес проєктування відбувався з урахуванням усіх розроблених до системи вимог.

Результатом виконання кваліфікаційної роботи бакалавра є вебзастосунок університетської поліклініки. Кінцевий застосунок відповідає всім поставленим вимогам до системи.

Перевагою розробленого застосунку є те, що він включає найкращі риси конкурентів на ринку, а також покриває всі недоліки та недопрацювання аналогічних систем, що дає змогу скласти значну конкуренцію у сфері надання медичних послуг.

Ще однією перевагою системи можна визначити те, що для розробки були підібрані найсучасніші інструменти та підходи, що дає великий потенціал для розширення системи без додаткових труднощів. У майбутньому функціонал системи може бути розширеним. Наприклад такі функції, як мультимовність, функції відображення контенту для користувачів з обмеженими можливостями, підключення аналітичних систем, таких як Google Analitics, розширення можливостей налаштування застосунку.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Стив Круг. Не заставляйте меня думать. Эксмо, 2017. 256 с.

2. Dunn M., Lewis S. Native mobile development: A cross-reference for IOS and android. O'Reilly Media, Incorporated, 2019. 394 с.

3. Frain B. Responsive Web Design with HTML5 and CSS: Develop future-proof responsive websites using the latest HTML5 and CSS techniques, 3rd Edition. Packt Publishing, 2020, 408 с.

4. Філіп Котлер. Основи маркетингу, 5-е європейське видання. Williams: 2019. 752 с.

5. Практика формування вимог у IT-проєктах від А до Я. Частина 3. Функції системи та Кордони проєкту. URL: https://habr.com/ru/post/336950/ (дата звернення: 16.05.2022)

6. Buch G., 2018. Introduction to uml from the creators of the language. Book ON DEMAND LTD. 25с.

7. Stephens M., Rosenberg D. Use case driven object modeling with UML: theory and practice. Apress, 2013. 472 с.

8. Джозеф Шмуллер. Освой самостоятельно UML 2 за 24 часа. Практическое руководство: Вильямс, 2005. 130 с.

9. UML для разработчиков.URL: https://habr.com/ru/company/ppr/blog/491146/ (дата звернення: 22.05.2022)

10. Bambara R. J., Allen P. R., Bambara J. J. Informix: client/server application development (mcgraw-hill series on client/server computing). McGraw-Hill Companies: 1997. 576 с.

11. Enterprise system architectures: Building client/server and Web-based systems. Boca Raton, FL : CRC Press: 2000. 959 с.

12. Scott E. SPA design and architecture: understanding single page web applications. Manning Publications: 2015. 275 с.

13. Macrae C. Vue.js: up and running: building accessible and performant web apps. O'Reilly Media: 2018. 174 с.

14. Biehl M. RESTful API design. CreateSpace Independent Publishing Platform: 2016. 294 с.

15. Гайдаржі В., Ізварін І. Бази даних в інформаційних системах. Університет "Україна": 2018. 418 с.

16. Fjordvald M., Nedelcu C. Nginx HTTP Server - Fourth Edition: Harness the power of Nginx to make the most of your infrastructure and serve pages faster than ever before. Packt Publishing - ebooks Account: 2018. 348 с.

17. Laurie P., Laurie B. Apache: the definitive guide (3rd edition). O'Reilly Media, Inc.: 2002. 536 с.

18. Laravel sanctum. URL: https://laravel.com/docs/9.x/sanctum (дата звернення: 06.06.2022).

19. Laravel fortify. URL: https://laravel.com/docs/9.x/fortify (дата звернення: 07.06.2022).

20. Amazon S3. AWS Amazon. URL: https://aws.amazon.com/s3/?nc1=h_ls (дата звернення: 09.06.2022).

21. Мілл І., Хобсон Сейєрс Е. Docker на практиці. Shelter Island: Manning Publications, 2019. 516 с.

ДОДАТОК А

Лістинг коду для docker-compose

```
version: "3.7"
services:
    app:
        build:
            args:
                user: pmbssuadmin
                uid: 1000
            context: ./
            dockerfile: Dockerfile
        image: pmbssu-polyclinic
        container_name: pmbssu-polyclinic-app
        restart: unless-stopped
        working_dir: /var/www/
        volumes:
            - ./:/var/www
            - ./php/local.ini:/usr/local/etc/php/conf.d/local.ini
        networks:
            - pmbssu-polyclinic
        stdin_open: true
        tty: true
    db:
        image: postgres
        container_name: pmbssu-polyclinic-db
        restart: unless-stopped
        environment:
            POSTGRES_USER: ${DB_USERNAME}
            POSTGRES_PASSWORD: ${DB_PASSWORD}
            POSTGRES_DB: ${DB_DATABASE}
            SERVICE_TAGS: dev
            SERVICE_NAME: pgsql
        volumes:
            - ./docker-compose/postgres/datadir:/var/lib/postgresql/data
        networks:
            - pmbssu-polyclinic
    nginx:
        image: nginx:1.17-alpine
        container_name: pmbssu-polyclinic-nginx
        restart: unless-stopped
        ports:
            - "8000:80"
        volumes:
            - ./:/var/www
            - ./docker-compose/nginx:/etc/nginx/conf.d
        networks:
            - pmbssu-polyclinic
networks:
    pmbssu-polyclinic:
        driver: bridge
```

ДОДАТОК Б

Лістинг коду базового класу контролерів ApiController.php

```php
<?php

namespace App\Http\Controllers\API;

use App\Http\Controllers\Controller;
use App\Services\ApiService;
use Illuminate\Http\Request;
use function response;

class ApiController extends Controller
{
    protected ApiService $apiService;

    public function __construct(ApiService $apiService)
    {
        $this->apiService = $apiService;
    }

    public function index(Request $request)
    {
        $result = [
            'status' => 200,
            'has_errors' => false
        ];

        try {
            $result['data'] = $this->apiService->getAll($request);
        }
        catch (\Exception $e) {
            $result = [
                'status' => 500,
                'has_errors' => true,
                'error' => $e->getMessage()
            ];
        }

        return response()->json($result, $result['status']);
    }

    public function store(Request $request)
    {
        $result = [
            'status' => 201,
            'has_errors' => false
        ];

        try {
            $data = $request->all();
            $result['data'] = $this->apiService->store($data);
        } catch (\Exception $e) {
            $result = [
                'status' => 415,
                'has_errors' => true,
                'error' => $e->getMessage()
            ];
        }
```

```
        return response()->json($result, $result['status']);
    }

    public function show($id)
    {
        $result = ['status' => 200, 'has_errors' => false];

        try {
            $result['data'] = $this->apiService->getById($id);
        } catch (\Exception $e) {
            $result = [
                'status' => 404,
                'has_errors' => true,
                'error' => "Not Found"
            ];
        }

        return response()->json($result, $result['status']);
    }

    public function update(Request $request, $id)
    {
        $result = ['status' => 200, 'has_errors' => false];

        try {
            $data = $request->all();
            $result['data'] = $this->apiService->update($data, $id);
        } catch (\Exception $e) {
            $result = [
                'status' => 500,
                'has_errors' => true,
                'error' => $e->getMessage()
            ];
        }

        return response()->json($result, $result['status']);
    }

    public function destroy($id)
    {
        $result = [
            'status' => 200,
            'has_errors' => false
        ];

        try {
            $result['data'] = $this->apiService->delete($id);
        } catch (\Exception $e) {
            $result = [
                'status' => 500,
                'has_errors' => true,
                'error' => $e->getMessage()
            ];
        }

        return response()->json($result, $result['status']);
    }
}
```

ДОДАТОК В

Лістинг коду фабричного методу модулів ресурсів

```
import * as constants from "../constants";

export const createModule = (api, module = {}) => {
    if (!api) throw new Error("Store::createModule::Argument 'api' is required!");
    let state = {};
    const { getters, actions, mutations } = {...module};

    if (module.hasOwnProperty("state")) {
        state = typeof module.state === "function"
            ? module.state()
            : Array.isArray(module.state)? Object.fromEntries(module.state)
                : module.state; }

    return {
        namespaced: true,
        state: () => ({
            [constants.STATE_RESOURCE]: [],
            [constants.STATE_RESOURCE_PAGINATION]: {},
            [constants.STATE_CURRENT_RESOURCE_ITEM]: null,
            ...state
        }),
        getters: {
            [constants.GETTER_RESOURCE](state) {
                return state[constants.STATE_RESOURCE];
            },
            [constants.GETTER_RESOURCE_PAGINATION](state) {
                return state[constants.GETTER_RESOURCE_PAGINATION];
            },
            [constants.GETTER_CURRENT_RESOURCE_ITEM](state) {
                return state[constants.GETTER_CURRENT_RESOURCE_ITEM];
            },
            ...getters
        },
        mutations: {
            [constants.MUTATION_UPDATE_RESOURCE](state, resource) {
                state[constants.STATE_RESOURCE] = resource;
            },
            [constants.MUTATION_UPDATE_RESOURCE_PAGINATION](state, pagination) {
                state[constants.STATE_RESOURCE_PAGINATION] = pagination;
            },
            [constants.MUTATION_UPDATE_CURRENT_RESOURCE_ITEM](state, current) {
                state[constants.STATE_CURRENT_RESOURCE_ITEM] = current;
            },
            [constants.MUTATION_APPEND_RESOURCE](state, items) {
                state[constants.STATE_RESOURCE].push(...items);
            },
            ...mutations
        },
        actions: {
            async [constants.ACTION_LOAD_RESOURCE]({ commit }) {
                const {resource, pagination} = await api.getAllPaginated()
                    .then(response => {
                        const resource = response.data.data;
                        delete response.data.data;
                        const pagination = response.data;
                        return { resource, pagination }
```

```
                    });
                commit(constants.MUTATION_UPDATE_RESOURCE, resource);
                commit(constants.MUTATION_UPDATE_RESOURCE_PAGINATION, pagination);
            },
            async [constants.ACTION_LOAD_MORE_RESOURCE]({commit, getters}) {
                const currentPagination =
getters[constants.GETTER_RESOURCE_PAGINATION];

                if (!currentPagination || !currentPagination.next_page_url)
                    return;

                const next = currentPagination.current_page + 1;

                commit(constants.MUTATION_UPDATE_RESOURCE_LOADING, true);

                const {resource, pagination} = await api.getAllPaginated(next)
                    .then(response => {
                        const resource = response.data.data;
                        delete response.data.data;

                        const pagination = response.data;

                        return {
                            resource,
                            pagination
                        }
                    });
                commit(constants.MUTATION_APPEND_RESOURCE, resource);
                commit(constants.MUTATION_UPDATE_RESOURCE_PAGINATION, pagination);
            },
            async [constants.ACTION_GET_RESOURCE]({ commit }, id) {
                const resource = await api.get(id)
                    .then(({ data }) => data);

                commit(constants.MUTATION_UPDATE_CURRENT_RESOURCE_ITEM, resource);
                return resource;
            },
            async [constants.ACTION_UPDATE_RESOURCE]({ commit }, payload) {
                const id = payload.id;

                const updated = await api.update(id, payload)
                    .then(({ data }) => data);

                commit(constants.MUTATION_UPDATE_CURRENT_RESOURCE_ITEM, updated);

                return updated;
            },
            async [constants.ACTION_DELETE_RESOURCE]({ commit }, payload) {
                const id = payload.id;

                const deleted = await api.delete(id)
                    .then(({ data }) => data);

                commit(constants.MUTATION_UPDATE_CURRENT_RESOURCE_ITEM, deleted);

                return deleted;
            },
            ...actions
        }
    };
};
```