2Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем та програмної інженерії
(повна назва факультету)
Кафедра програмної інженерії
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

бакалавр
(назва освітнього ступеня)

на тему: Розробка веб-застосунку для роботи з клієнтами для працівників медичної клініки на основі технологій JavaScript

Виконав(ла): студент(ка) 4 курсу, групи СП-41
спеціальності 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

(підпис) Колтун В. О. (прізвище та ініціали)

Керівник (підпис) Коноваленко І. В. (прізвище та ініціали)

Нормоконтроль (підпис) Стоянов Ю. М. (прізвище та ініціали)

Завідувач кафедри (підпис) Петрик М. Р. (прізвище та ініціали)

Рецензент (підпис) (прізвище та ініціали)

Тернопіль
2024

АНОТАЦІЯ

Колтун В. О. Розробка веб-застосунку для роботи з клієнтами для працівників медичної клініки на основі технологій JavaScript : кваліфікаційна робота на здобуття освітнього ступеня бакалавр за спеціальністю „121 — інженерія програмного забезпечення“ / Колтун Віталій Олегович. — Тернопіль: ТНТУ, 2024, сторінок – 79, рисунків – 76, таблиць – 1, частин – 5, додатків – 1, посилань – 26.

Метою дипломної роботи є розробка веб-застосунку, спрямованого на оптимізацію взаємодії працівників медичного закладу з клієнтами.

В процесі роботи проводиться аналіз існуючих рішень і технологій для розробки веб-застосунків, визначаються основні вимоги до системи, проєктується та розробляється власна веб-система з використанням технологій JavaScript, зокрема фреймворки і бібліотеки (React, Node.js, Express.js).

Результатом виконаної роботи буде розроблений функціональний веб-застосунок, який спрощує та оптимізує взаємодію співробітників клініки з клієнтами.

Ключові слова: ОПЕРАТОР, ЛІКАР, ПАЦІЄНТ, ЗАПИС НА ПРИЙОМ, МЕДИЧНА КАРТКА, ВЕБ-ЗАСТОСУНОК, JAVASCRIPT, NODE.JS, EXPRESS.JS, MANGODB, REACT.

ABSTRACT

Koltun V. O. Development of a web-application for working with clients for employees of a medical clinic based on JavaScript technologies: qualifying work for obtaining a bachelor's degree in the specialty "121 — software engineering" / Koltun Vitalii Olegovych. — Ternopil: TNTU, 2024, pages – 79, figures – 76, tables – 1, parts – 5, appendices – 1, references – 26.

The aim of the diploma work is to develop a web application aimed at optimizing the interaction of employees of a medical institution with clients.

In the process of work, the analysis of existing solutions and technologies for the development of web applications is carried out, the main requirements for the system are determined, the own web system is designed and developed using JavaScript technologies, in particular frameworks and libraries (React, Node.js, Express.js).

The result of the completed work will be the development of a functional web application that simplifies and optimizes the interaction of clinic employees with clients.

Keywords: OPERATOR, DOCTOR, PATIENT, APPOINTMENT, MEDICAL CARD, WEB APPLICATION, JAVASCRIPT, NODE.JS, EXPRESS.JS, MANGODB, REACT.

ЗМІСТ

ВСТУП

Сучасний розвиток інформаційних технологій кардинально змінює різні галузі діяльності людини, і медична сфера не є виключенням. Інтеграція новітніх технологій у медичну практику дозволяє значно покращити якість надання медичних послуг, підвищити ефективність управління медичними закладами та забезпечити зручний доступ до медичної інформації для пацієнтів. Зокрема, розробка спеціалізованих веб-застосунків стає важливим інструментом для оптимізації процесів у медичних установах.

Актуальність даної роботи полягає у необхідності створення гнучкого, масштабованого та високопродуктивного веб-застосунку для медичної клініки, який би відповідав сучасним вимогам щодо швидкості, надійності та зручності використання. Такий застосунок дозволить спростити роботу медичних працівників, автоматизувати процеси запису на прийом, управління медичними картками, обробки запитів та покращити комунікацію між лікарями та пацієнтами.

Метою даної дипломної роботи є розробка веб-застосунку для медичної клініки, який забезпечить автоматизацію основних процесів та підвищення ефективності взаємодії між медичними працівниками та пацієнтами.

Для досягнення цієї мети необхідно вирішити наступні завдання:

Провести аналіз існуючих рішень та визначити їхні переваги та недоліки.

Розробити архітектуру веб-застосунку, що відповідатиме вимогам гнучкості, масштабованості та продуктивності.

Виконати розробку клієнтської та серверної частин застосунку з використанням сучасних технологій JavaScript

Забезпечити інтеграцію веб-застосунку з базами даних та іншими зовнішніми сервісами.

Розробити систему безпеки для захисту медичних даних від несанкціонованого доступу.

Також, особлива увага буде приділена зручності інтерфейсу користувача. Веб-застосунок повинен бути інтуїтивно зрозумілим та легким у використанні, щоб медичні працівники могли швидко та ефективно виконувати свої завдання. Для цього буде використано сучасні підходи до дизайну інтерфейсів, такі як модульний дизайн та адаптивні компоненти.

Галузь застосування результатів даної роботи охоплює медичні установи різного рівня, які прагнуть підвищити якість надання медичних послуг за рахунок впровадження сучасних інформаційних технологій. Веб-застосунок, розроблений у рамках цієї дипломної роботи, може бути використаний для автоматизації роботи клінік, поліклінік, лікарень та інших медичних закладів. Це дозволить їм покращити свої внутрішні процеси, удосконалити взаємодію з пацієнтами та забезпечити швидкий доступ до необхідної медичної інформації.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Актуальність предметної області

Актуальність досліджуваної теми визначається поєднанням сучасних завдань, які стоять перед медичними установами, і загальної потреби в оптимізації медичного управління документацією. Однією з причин актуальності розробки даного веб-застосунку є зростаючий обсяг медичної інформації. Це стосується не лише обліку особистої історії хвороби пацієнта, але й даних про проведені терапевтичні заходи та лабораторні дослідження. Збільшення обсягу інформації ставить перед медичними закладами завдання забезпечити ефективне зберігання, обробку та доступ до цих даних [1].

Також, для медичних працівників важливим є швидкий та точний доступ до медичної інформації, щоб вони могли приймати обґрунтовані рішення щодо діагностики, лікування та догляду за пацієнтами. Традиційні методи управління документами на паперовій основі не можуть задовольнити ці вимоги в середовищах зі зростаючими обсягами даних і швидким темпом роботи. В результаті, використання сучасних інформаційних технологій для автоматизації управління документами може значно підвищити продуктивність медичних працівників. Надання можливості швидкого доступу до необхідної інформації та автоматизації рутинних процесів економить медичному персоналу час для безпосередніх медичних процедур і взаємодії з пацієнтами [2].

Ще однією, не менш важливою причиною є необхідність забезпечення безпеки даних. Адже, забезпечення безпеки та конфіденційності медичних даних пацієнтів — це важлива складова у створенні веб-додатку. При зберіганні більшої частини інформації в папері, підвищується шанс витоку важливих документів. Використання новітніх технологій дозволяє створити надійну систему захисту, що зводить до мінімуму ризик витоку і несанкціонованого доступу до конфіденційної інформації [1, 2].

1.2 Огляд існуючих рішень

Перед тим як розпочати сам процес розробки, важливо добре зрозуміти, які вже існують рішення на ринку, їхні переваги, недоліки та технологічні аспекти. Для цього проведемо огляд функціональностей існуючих програмних продуктів для роботи з клієнтами медичних закладів. Проаналізуємо їхні основні функції, спробуємо виокремити їхні переваги та недоліки, щоб мати чітке уявлення про те, що вже доступно на ринку. Це допоможе покращити розуміння існуючих рішень та краще підготуватися до процесу розробки веб-застосунку для медичної сфери.

Щоб провести аналіз конкурентних рішень, скористаємось даними про діючі та підключені медичні інформаційні системи (МІС) до електронної системи охорони здоров'я України – eHealth [3].

МедЕйр — це програмне забезпечення яке розробляється та вдосконалюється ще з далекого 2010 року. Це комплексна система, що автоматизує бізнес-процеси в лікувально-профілактичних установах різних профілів і рівнів. На сьогоднішній день, система використовується на 11000 робочих місцях і має понад 35000 користувачів [4]. Інтерфейс додатку зображено на рисунку 1.1.

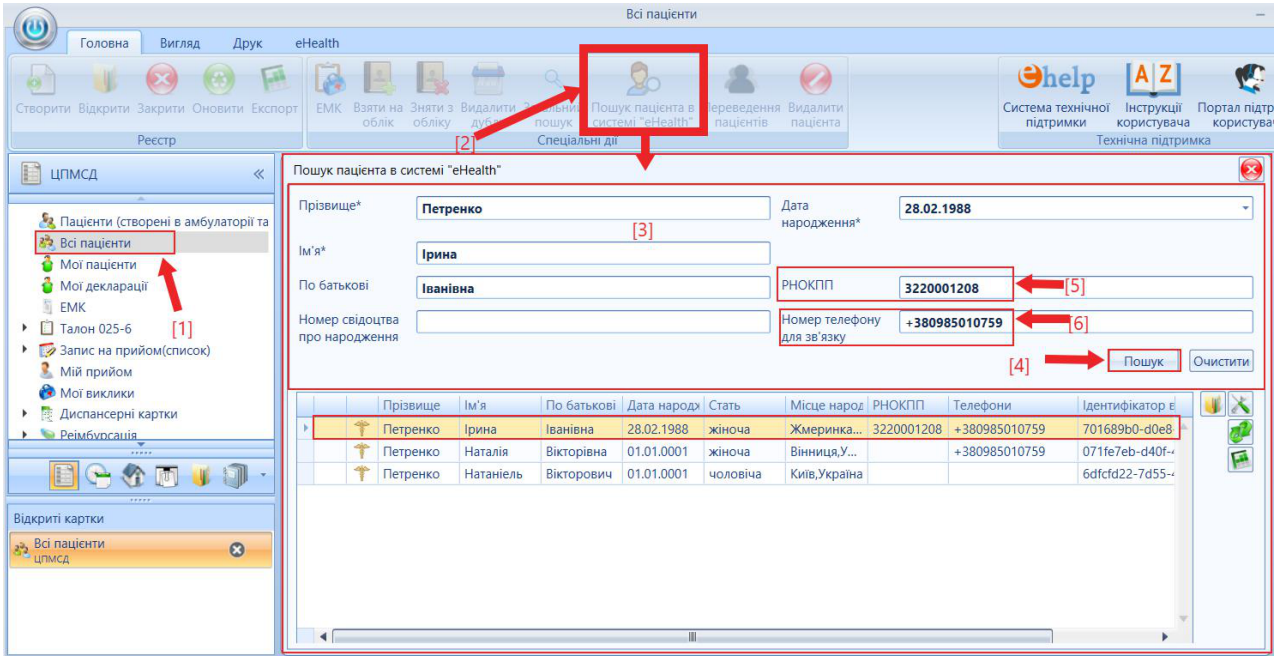


Рисунок 1.1 – Інтерфейс “МедЕйр”

До основних функціональних переваг даної системи, можна віднести: можливість ведення медичної карти, запис пацієнтів на прийом до лікаря, перегляд медичних зображень (PACS), видача електронних рецептів та направлень. Окрім, цього програма надає можливість грамотно керувати ресурсами клініки, а саме банком медичних засобів та препаратів.

Не зважаючи на перелічені переваги, є й достатньо значні недоліки: Обмежена сумісність з різними операційними системами (ОС), оскільки офіційно програмний продукт підтримує лише Windows 10, тож це може бути проблемою для деяких клінік, які до прикладу використовують новіші версії Windows або ж взагалі інші ОС: macOS або Linux. Також можна підкреслити, складний та застарілий інтерфейс.

Доктор Елекс це сучасне технологічне рішення, яке переважно використовується для приватних медичних закладів [5]. Система охоплює безліч функцій, починаючи від первинного звернення до подальшої комунікації після завершення лікування. Система складається з понад 40 модулів, що забезпечують автоматизацію та ефективне управління різними аспектами медичної діяльності [6]. Графічний інтерфейс медичної системи подано на рисунку 1.2.

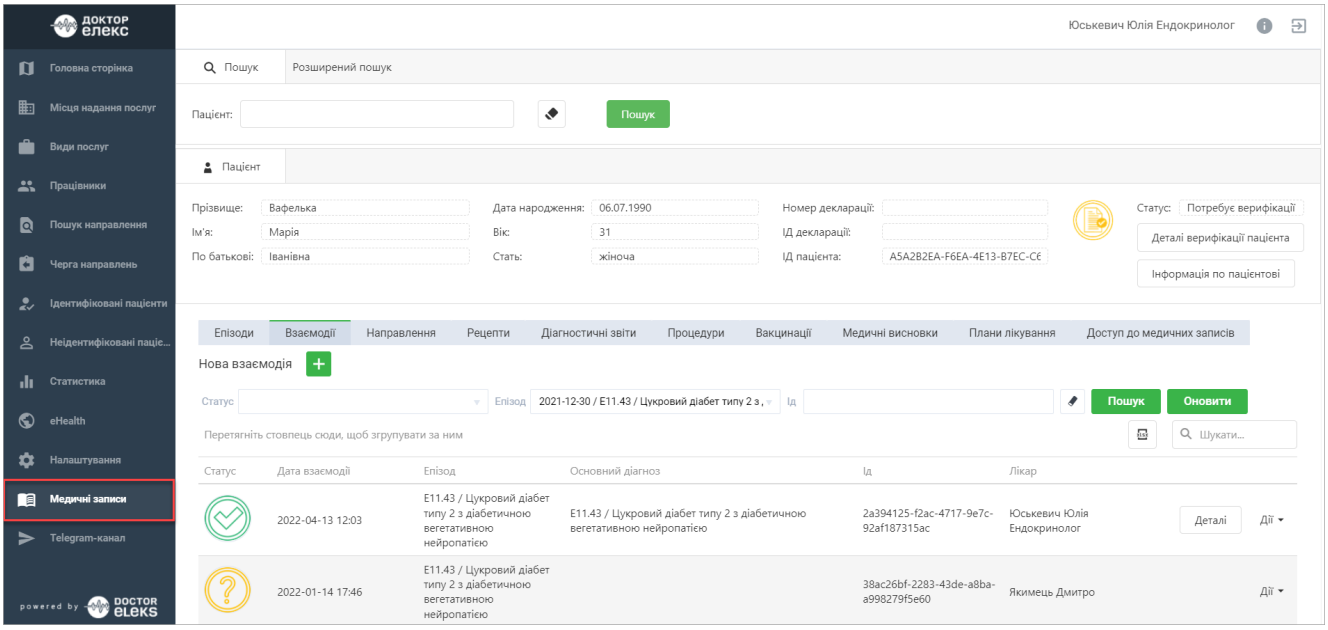


Рисунок 1.2 – Інтерфейс “Доктор Елекс”

На відміну від попереднього розглянутого рішення, “Доктор Елекс” – не прив’язує своїх клієнтів до технічних вимог, тим самим надає можливість обирати між веб рішенням чи десктопним варіантом. Для веб рішення, клієнтом виступатиме будь-який браузер, тож не важливо, яка буде встановлена ОС. Для десктопного – потрібна 10 версія Windows або новіша. Важливо додати, що система може працювати, як на власному сервері, так й у “хмарі” компанії.

Окрім, технічних переваг, можна виділити й функціональні: інтеграція з діагностичним обладнанням, інтеграція з іншими системами (зовнішні лабораторії такі як: Esculab, Synevo, IP телефонія), кабінети лікаря та пацієнта, запис пацієнтів на прийом, перегляд медичної картки та документів, керування ліжкомісцями, фінансовий облік та статистика.

До недоліків, можна віднести високу вартість впровадження. Особливо це стосується великих приватних клінік, де вартість ліцензії від 50 ПК на одне робоче місце становитиме 990 $ і це без додаткових встановлених модулів [6].

Helsi – найпопулярніша інформаційна медична система України на даний час, вона обробляє 200.тис записів до лікарів щодня. В основному з нею працюють державні та комунальні заклади, що складає 81%, інша частка припадає на приватні клініки [7]. Інтерфейс системи зображено на рисунку 1.3.

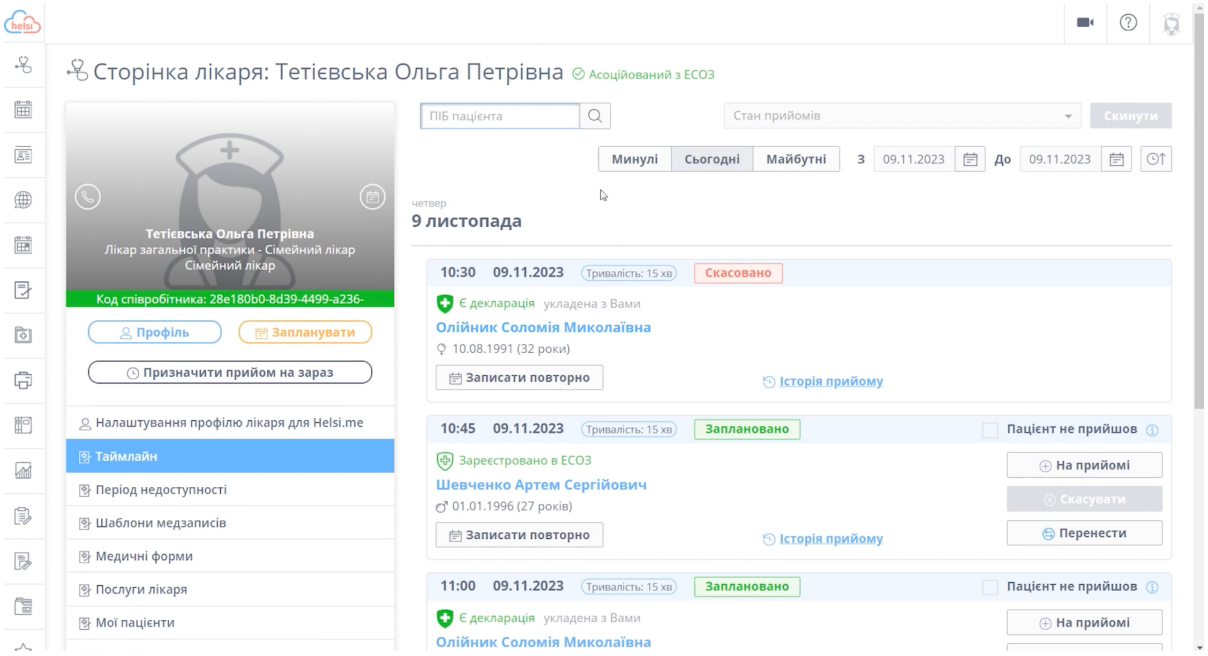


Рисунок 1.3 – Інтерфейс “Helsi”

Функціональні переваги: особисті кабінети для лікаря та пацієнта, самостійний запис пацієнта до лікаря, ведення електронних медичних карток можливість проведення online консультацій, зміна статусів прийому пацієнта, формування звітів та статистики. Крім того, система пропонує функцію для перегляду доступних лікарів, годинами їх прийому й адресами лікарень [8].

Недоліки: сервіс має обмежені можливості інтеграції з платіжними системами, IP-телефонією, зовнішніми лабораторіями (наприклад: Діла. Esculab, Synevo). Також, медичні установи обмежені у вільному виборі розміщення персональних даних своїх пацієнтів, Helsi обробляє та зберігає їх у власному дата центрі [8].

Health24 – хмарна веб-система від української IT компанії ”E-tech” розроблена за закордонними стандартами ведення електронної документації. Надає комплексні рішення для медичних закладів усіх рівнів — первинної, вторинної та третинної медичної допомоги. Система містить електронні дані про понад 13 мільйонів українців та активно використовується в державних і приватних медичних закладах. Основними користувачами Health24 є працівники державних та приватних закладів, лікарі-ФОП [9, 10]. Інтерфейс веб-системи відображено на рисунку 1.4.

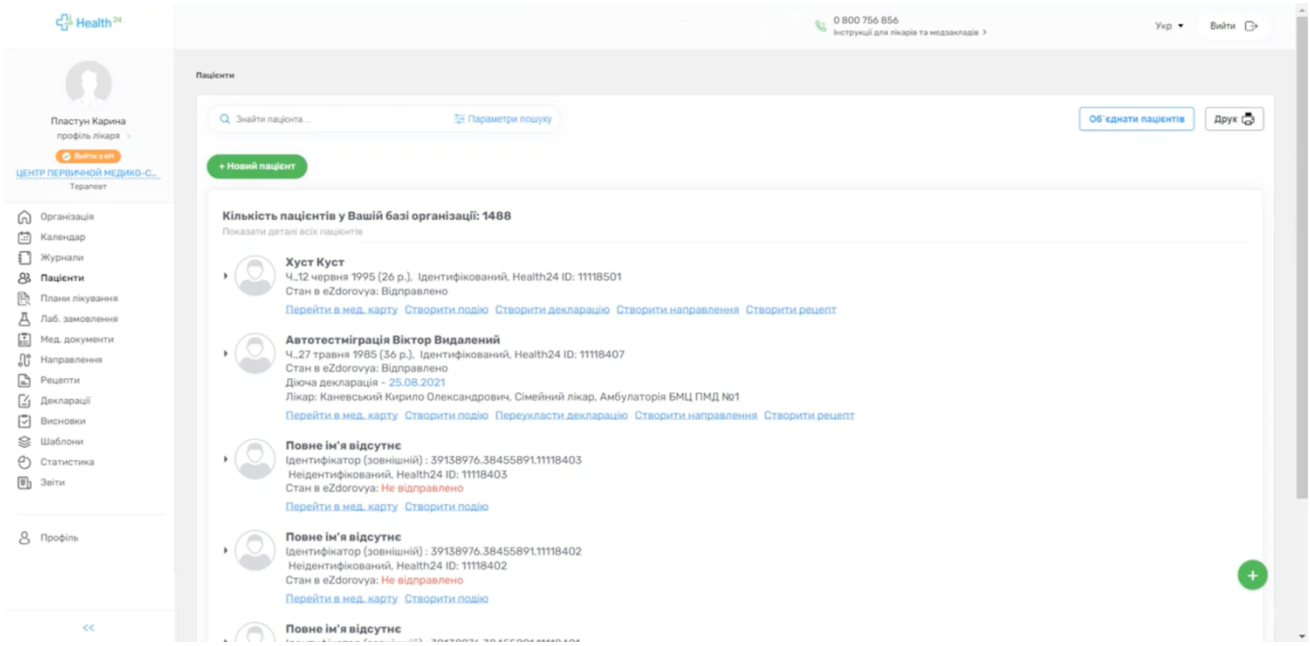


Рисунок 1.4 – Інтерфейс “Health24”

Основні функціональні можливості: запис пацієнта на прийом, особисті кабінети пацієнта та лікаря, ведення журналів приймання, доступ до медичної карти пацієнта, керування залишками матеріальних ресурсів клініки, перегляд звітності та статистики медичного закладу, інтеграції з аналізаторами лабораторії EMCIlab, управління взаємозв'язками з контрагентами. Окрім цього, системою можна користуватися на будь-яких пристроях, будь це: смартфон, планшет чи комп’ютер [11].

Недоліки: Як і більшість “хмарних рішень” зберігання особистої інформації відбувається на сторонніх дата-центрах, це обмежує мед. заклад у вільному виборі місця зберігання персональних даних своїх пацієнтів та створює певну залежність від них. Також, система має обмежені можливості, щодо інтеграції зі зовнішніми лабораторіями, такими як: Esculab, Synevo, Діла та ін.

Отже, розглянувши існуючі рішення, було створено порівняльну таблицю медичних інформаційних систем (МІС) по критеріях та оцінено за 5-бальною системою (див. таблиця 1.1).

Таблиця 1.1 – Оцінка інуючих МІС

Критерій	МедЕйр	Doctor Eleks	Helsi	Health24
Сумісність з різними ОС	1	5	4	4
Гнучкість зберігання даних	4	5	3	3
Функціональні можливості	3	4	3	3
Легкість в користуванні	2	4	4	4

1.3 Опис вимог до проєкту

Розробка веб-застосунку для роботи з клієнтами для працівників медичної клініки є важливим завданням, що дозволяє оптимізувати процеси взаємодії з пацієнтами та підвищити ефективність надання медичних послуг. Основні вимоги до проєкту, включатимуть функціональні та нефункціональні вимоги, що є необхідними для забезпечення належної роботи застосунку, вимоги ґрунтуються на аналізі вище згаданих рішень. До уваги будуть братися не лише переваги або недоліки кожної з систем, але й наявні у всіх них опції.

Функціональні вимоги:

Реєстрація профілів працівників у веб-застосунку

Вхід/Вихід

Редагування даних профілю працівників

Додавання/Видалення пацієнтів та їхніх особистих даних у БД

Редагування особистих даних пацієнтів

Створення/Видалення медичної картки

Додання/Перегляд/Редагування/Видалення медичних записів

Запис пацієнтів на прийом до лікаря

Зміна статусу запису на прийом (Заплановано, Розпочато, Завершено, Скасовано)

Зміна дати та часу запису до лікаря (перенесення прийому)

Пошук/Сортування/Фільтрування даних

Не Функціональні вимоги:

Забезпечення захисту конфіденційної інформації пацієнтів від несанкціонованого доступу. Для цього варто впровадити надійні механізми шифрування для зберігання та передачі медичних даних. Також застосунок повинен забезпечувати швидке виконання запитів та відгук користувацького інтерфейсу. Система повинна бути здатна обробляти збільшену кількість даних та кількості користувачів без значного погіршення продуктивності.

Не можна залишити без уваги й інтерфейс користувача. Він повинен бути інтуїтивно зрозумілим і зручним для використання, щоб медичні працівники могли швидко освоїти роботу з системою.

Визначення чітких вимог до проєкту є критично важливим етапом розробки веб-застосунку для медичної клініки. Вимоги, описані в цьому розділі, забезпечать основу для створення ефективної, безпечної та зручної системи, яка оптимізує роботу медичного персоналу та підвищить якість обслуговування пацієнтів.

2 ПРОЄКТУВАННЯ ВЕБ-ЗАСТОСУНКУ

2.1 Визначення акторів та варіантів використання

Важливим етапом розробки функціонального веб-застосунку для роботи з клієнтами для клініки є коректне визначення ролей та відповідальностей користувачів. В даній розробленій системі можна виділити двох акторів – оператора та лікаря. Ці учасники виконують різні ролі та беруть на себе певні обов'язки в системі, що забезпечує злагоджену роботу та підвищує загальну ефективність надання медичних послуг. Далі буде детальніше розглянуто функції кожного з цих учасників, а також їхні конкретні завдання та відповідальності у межах веб-застосунку:

Оператор відіграє ключову роль в адмініструванні медичної інформаційної системи, забезпечуючи ефективне управління даними пацієнтів. Його основні обов'язки включають створення нових пацієнтів, редагування їх даних та видалення профілів з системи. Оператор також може здійснювати перегляд бази даних пацієнтів клініки, пошук конкретних пацієнтів за їхніми особистими даними, перегляд медичних карток. Додатково, оператор відповідає за організацію записів до лікарів, а саме запис пацієнтів до лікарів. Він може змінювати статус запису, переносити прийоми до лікаря. Ще однією оцією є фільтрування та сортування даних. Ці функції є важливими для забезпечення злагодженого робочого процесу в клініці та ефективної координації між різними відділами.

Лікар, з іншого боку, є основним медичним користувачем системи, який безпосередньо взаємодіє з медичними записами пацієнтів. Лікар має можливість переглядати записи до лікаря, здійснювати пошук записів, змінювати їх статус та переносити прийоми. Це дозволяє лікарю оперативно керувати своїм графіком. Крім того, лікар може переглядати медичні картки та записи в них, створювати нові медичні записи, редагувати чи видаляти існуючі. Важливою функцією є також фільтрування та сортування медичних даних, що допомагає лікарю швидко знаходити необхідну інформацію.

Для визначених акторів веб-застосунку було створено UML-діаграму варіантів використання, яка подана на рисунку 2.1. Інструментом для створення виступає онлайн-інструмент draw.io [12].

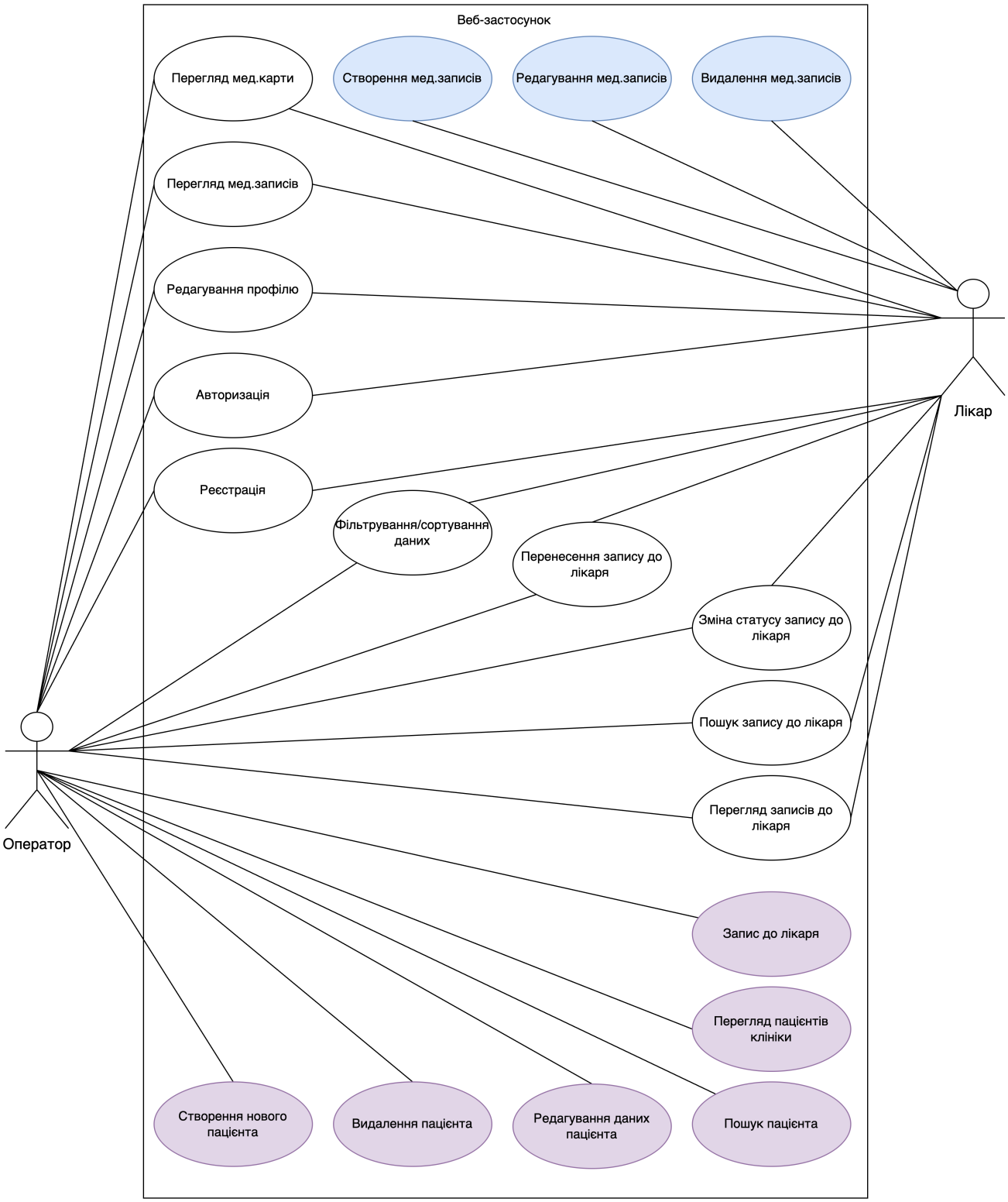


Рисунок 2.1 – Діаграма варіантів використання

2.2 Вибір методології розробки

Вибір правильної методології є важливим для забезпечення успіху проекту, оскільки вона допомагає організувати роботу команди, контролювати процес розробки та гарантувати, що кінцевий продукт буде відповідати вимогам.

У контексті веб-розробки було обрано методологію RAD (Rapid Application Development), яка є гнучкою і відома своєю здатністю прискорювати процес розробки та забезпечувати високу адаптивність до змінних вимог користувачів [13].

Як бачимо з рисунку 2.2, методологія RAD зосереджена на швидкому прототипуванні та ітераційному розвитку, що дозволяє розробникам та користувачам активно взаємодіяти на ранніх етапах розробки. Це не тільки допомагає забезпечити, що кінцевий продукт буде точно відповідати очікуванням клієнта, але й значно скорочує ризики невідповідності специфікаціям, оскільки зміни можуть бути внесені набагато швидше порівняно з традиційними методами [13].

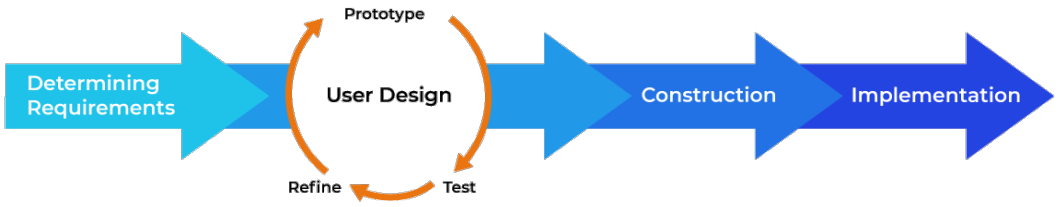


Рисунок 2.2 – Етапи методології розробки RAD

Однією з ключових переваг даної методології є її спроможність адаптуватися до швидко змінюваних умов ринку, що робить її ідеальною для стартапів та проектів, які потребують швидкого розвитку. У веб-розробці, де нові технології та тренди постійно змінюються, здатність швидко реагувати на нові вимоги є критичним. Завдяки RAD розробники можуть швидко створювати та тестувати

прототипи, забезпечуючи тим самим оперативність випуску продуктів, яка є життєво необхідною для збереження конкурентоспроможності [13].

Використання методології RAD також сприяє кращій колаборації між усіма учасниками проекту. Регулярний зворотний зв'язок від користувачів і активна участь клієнтів у процесі розробки забезпечують високу прозорість, що веде до більшої задоволеності стейкхолдерів і кращої відповідності продукту до бізнес-цілей [13].

2.3 Вибір технологій розробки

Для серверної частини застосунку був вибраний Node.js, який може запускати код JavaScript поза браузером. Node.js має асинхронну архітектуру, керовану подіями, яка дозволяє обробляти велику кількість одночасних з'єднань, не блокуючи їх. Це ідеально підходить для створення додатків, які вимагають взаємодії з сервером у режимі реального часу, таких як чати та веб-застосунки. Крім того, велика екосистема модулів і пакетів, включених в NPM (Node Package Manager), значно прискорює розробку і підвищує продуктивність [14].

Для створення API та управління маршрутизацією використовуватиметься Express.js - це легка та гнучка платформа, яка полегшує розробку серверних додатків [15].

Mongoose був обраний для роботи з базами даних MangoDB, він забезпечує належне управління документами та підтримує перевірку даних. Безпека забезпечується за допомогою JsonWebToken – ця бібліотека забезпечує функціонал авторизації та керування доступу до бази даних зовнішніх користувачів [16, 17].

Клієнтська частина розроблятиметься на React, популярній бібліотеці JavaScript для створення користувацьких інтерфейсів. React спрощує підтримку та розробку коду, дозволяючи створювати модульні компоненти. Однією з ключових

переваг React є використання Virtual Dom, який забезпечує ефективні оновлення інтерфейсу користувача та покращує продуктивність [18].

Щоб покращити швидкодію в React, було обрано Next.js - це платформа, яка забезпечує рендеринг на стороні сервера та статичний рендеринг сайтів. Це дозволить підвищити швидкість завантаження сторінок, відсилаючи запити на сервер. За допомогою SWR дані отриманні з сервера не будуть оновлюватись без необхідності, що забезпечить ефективне управління станом даних та знизить навантаження на сервер [19, 20].

Tailwind CSS – бібліотека, що дозволяє швидко створювати стилізовані компоненти. React-Select дозволяє реалізувати інтуїтивно зрозумілий вибір елементів, a zustand забезпечує ефективне управління станом програми. Для роботи з формами вибирається react-hook-form створення гнучких та складних форм з мінімальною кількістю коду [21].

2.4 Вибір архітектури розробки для серверної та клієтської частин

Вибір архітектури є важливим аспектом розробки програмного забезпечення, особливо для клієнт-серверних застосунків. Правильна архітектура визначає, наскільки легко буде підтримувати, масштабувати та розширювати функціонал проєкту в майбутньому. Він забезпечує чітку організацію коду, розподіл обов'язків між різними компонентами системи та дозволяє зробити розробку ефективнішою. Без правильної архітектури проєкт може швидко стати важким в обслуговуванні, що значно ускладнить реалізацію нових функцій і виправлення помилок [22].

Для серверної частини було обрано архітектуру MVC та дещо модифіковану під Node.js та Express.js. Логіка архітектури заснована на поділі коду на кілька ключових компонентів, кожен з яких має свою унікальну роль. Основна ідея полягає в поділі обов'язків, забезпечуючи модульність.

Модель відповідає за структуру даних та їх взаємодію з базою даних. Вона визначає схеми даних, правила валідації та забезпечує основні операції зі зберігання та вилучення інформації з бази даних. Це дозволяє сконцентрувати всю логіку, пов'язану з нашими даними в одному місці, що полегшує управління даними.

Сервіс містить бізнес-логіку програми та виконує операції з даними, які можуть зберігатись в одній та більше моделей. Вони забезпечують складні обчислення, обробку даних, взаємодію із зовнішніми API та інші завдання, не пов'язані безпосередньо з контролером. Помістивши бізнес-логіку в сервіс буде можливість повторно використовувати ці функції в різних частинах програми, зменшуючи дублювання коду та полегшуючи розробку.

Контролери відповідають за обробку HTTP-запитів і надсилання відповідей. Вони взаємодіють зі сервісами, щоб виконати необхідну логіку та підготувати дані для відповіді клієнту. Контролери зазвичай містять мінімум логіки, зосереджуючись на обробці запитів та виклику відповідних методів сервісу. Цей підхід полегшує розуміння та розділення коду, оскільки він чітко розрізняє рівні обробки запитів та бізнес-логіку.

Роутер (маршрутизатор) визначає шляхи, які клієнт може використовувати для зв'язку з сервером, і пов'язує ці шляхи з відповідними методами контролера. Він також включає проміжну обробку даних для авторизації, логування та валідації даних, що передає клієнт.

Описана архітектура зображена на рисунку 2.3.

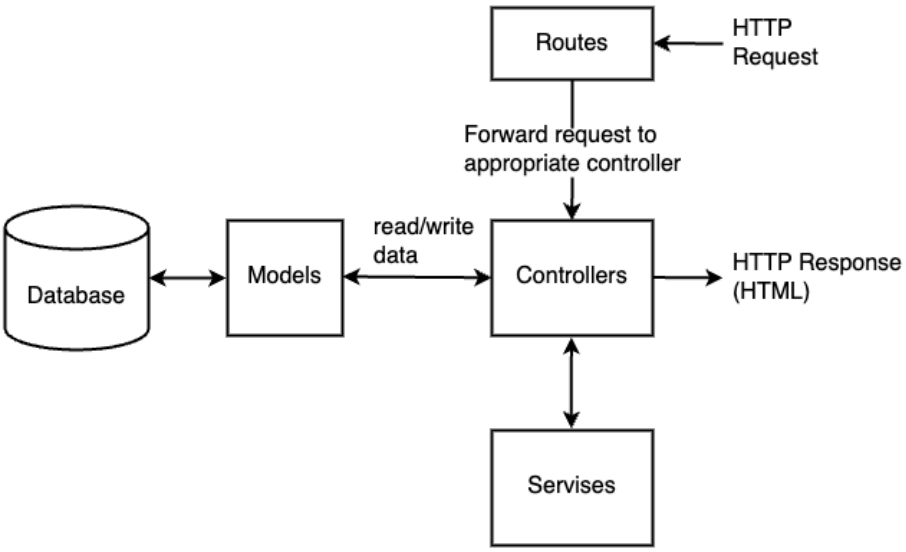


Рисунок 2.3 – Діаграма архітектури

Для розробки клієнтської частини застосунку була обрана архітектура Feature-Sliced Design. Основна суть цієї методології полягає в розподілі проекту за різними рівнями організації коду (шари, фрагменти, сегменти). Проєкт складається з шарів, кожен шар має свою сферу призначення [23].

Ось короткий опис основних рівнів такого проекту:

Shared (загальний) рівень містить функції, які не залежать від деталей проекту, таких як бібліотеки, компоненти інтерфейсу користувача, що не містять бізнес-логіки та запити до серверної частини.

Рівень Entities (сутностей) включає бізнес-об'єкти. Наприклад, користувачі, медичні карти та записи на прийом.

Рівень Features (функціональності) - містить все, що пов'язано з тим, як користувач взаємодіє в проекті, а також компоненти з бізнес-логікою.

Рівень Widgets (віджетів) об'єднує елементи з Shared, Entities та Features рівнів у самостійні компоненти.

Рівень Pages (сторінкок) містить компоненти, що складаються з всіх вищеописаних рівней для формування повноцінної сторінки.

Кожен шар розділений на підпапки, які називаються Slice (зрізами). Зрізи впорядковують код за бізнес-сферами, завдяки цьому можна легко орієнтуватися у проекті. Наприклад, є можливість згрупувати всі компоненти, що стосуються

пацієнта, в один Slice. Це дозволяє розробникам легко знаходити та змінювати необхідний їм код, зберігаючи логічно пов'язані модулі близько один до одного. Важливою особливістю є те, що зрізи на одному рівні не можуть безпосередньо використовувати інші зрізи того ж рівня. Це допомагає підтримувати чистоту та чіткість структури коду [23].

На найнижчому рівні кожен зріз складається з сегментів. Сегмент розділяє код відповідно до його технічним призначенням і забезпечує додатковий рівень організації. Наприклад, сегмент інтерфейсу пацієнта містить стилі та візуальні компоненти, сегмент моделі містить стан та методи керування, сегмент API містить логіку запитів сервера. На цьому рівні деталізації код може бути ретельно налаштований для забезпечення високої якості та повторного використання компонентів.

Описана архітектура зображена на рисунку 2.4.

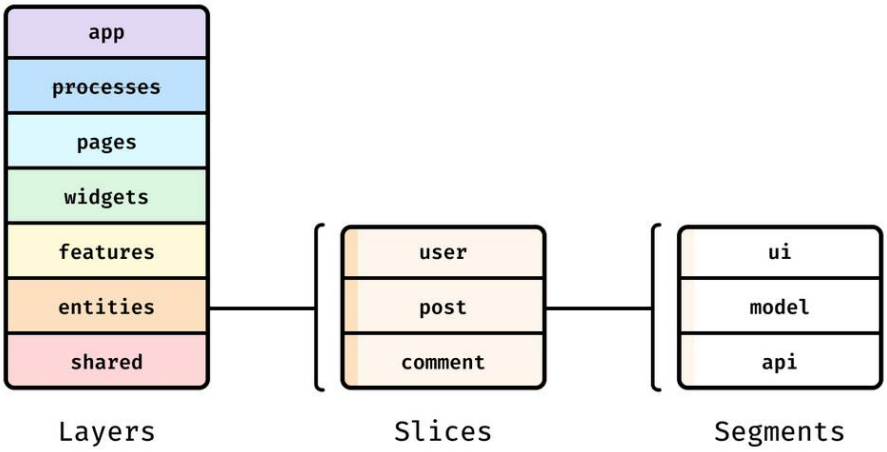


Рисунок 2.4 – Діаграма архітектури Feature-Sliced Design

Таким чином, підхід до проєктування з розділенням функцій дозволяє структурувати та редагувати код на всіх рівнях, зрізів та сегментів. Це забезпечує масштабований, зрозумілий та підтримуваний інтерфейс для користувача системи, який може легко адаптуватися до мінливих потреб медичного закладу.

3 РОЗРОБКА ВЕБ-ЗАСТОСУНКУ

3.1 Реалізація основних класів серверної частини

3.1.1 Реалізація схем бази даних

Розробка серверної частини веб-програми повинна починатися з продумання схем для бази даних, оскільки це дозволяє визначити структуру, взаємозв'язки та логіку зберігання даних на ранніх етапах реалізації проекту. Схеми забезпечують чітку структуру даних та полегшують розробку, тестування та обслуговування коду. Це також допомагає підтримувати цілісність даних та запобігати багатьом помилкам.

Документо-орієнтована база даних MongoDB зберігає дані у форматі BSON (двійковий JSON). Це дозволяє зберігати складні структуровані дані без необхідності суворого дотримання схем, специфічних для реляційних баз даних. MongoDB забезпечує гнучкість, дозволяючи динамічно змінювати структури даних при збереженні високої продуктивності. Використання індексів підвищує ефективність запитів, а функції реплікації та фрагментації забезпечують масштабованість та високу доступність.

Для визначення схем і моделей в Mongodb зазвичай використовується бібліотека Mongoose, яка надає зручний API для взаємодії з базою даних. Схема Mongoose визначає структуру документа, типи даних і взаємозв’язки між ними.

Отже, перша сформована схема – Користувач, яка призначена для зберігання інформації про користувачів системи та визначає основні поля - індекс, логін, пароль, роль, спеціалізацію та особисту інформацію (ім'я, прізвище та по-батькові користувача). Логін користувача є унікальним та проіндексованим полем, тому операції пошуку будуть виконуватись в рази швидше. Роль користувача визначається з обмеженого переліку (лікар або оператор), який забезпечує перевірку даних. Код поданий на рисунку 3.1

```
const schema = new mongoose.Schema({
    index: { type: Number,required: true},
    login: { type: String, required: true, unique: true, index: true},
    password: { type: String, required: true },
    role: { type: String, enum: [ROLES.DOCTOR, ROLES.OPERATOR], required: true},
    specialization: { type: String, default: '' },
    info: {
      name: { type: String, default: '' },
      surname: { type: String, default: '' },
      patronymic: { type: String, default: '' }}
  },
  { timestamps: true}
)
```

Рисунок 3.1 – Схема БД для користувача (User.model.js)

Далі було створено схему пацієнта, яка містить інформацію про ім'я, прізвище, по батькові, стать, дату народження та номер телефону. Поля ім'я, прізвище, стать, дата народження і номер телефону є обов'язковими, що гарантує доступність основної інформації про кожного пацієнта. Поле "стать" обмежене значенням "Чоловік" або "Жінка". Код подано на рисунку 3.2

```
const schema = new mongoose.Schema(
  {
    info: {
      name: { type: String, required: true },
      surname: { type: String, required: true },
      patronymic: { type: String, default: '' }
    },
    gender: { type: String, enum: [GENDERS.MAN, GENDERS.WOMAN], required: true },
    birthday: { type: Date, required: true },
    phone: { type: String, required: true }
  },
  {
    timestamps: true
  }
)
```

Рисунок 3.2 – Схема БД для пацієнта (Patient.model.js)

Схема медичної картки містить зв'язок між пацієнтом та його медичними записами. Вона містить посилання на об’єкт конкретного пацієнта, що забезпечує легкий доступ до інформації про пацієнта за допомогою мед. картки. Код описаної схеми відображено на рисунку 3.3.

```
const schema = new mongoose.Schema(
  {
    patient: {
      type: mongoose.Schema.Types.ObjectId,
      ref: 'Patient',
      required: true
    }
  },
  {
    timestamps: true
  }
)
```

Рисунок 3.3 – Схема БД для медичної картки (MedCard.model.js

Схема медичних записів включає: тип запису, ідентифікатор пацієнта, ідентифікатор лікаря, медичну карту, діагноз, рекомендації та дату створення. Поля пацієнт, лікар, медична картка, діагноз та рекомендації є обов'язковими. Така структура містить всі необхідні дані про стан здоров’я пацієнта. Код описаної схеми зображено на рисунку 3.4.

```
const schema = new mongoose.Schema(
  {
    type: {type: String,required: true},
    patient: {type: mongoose.Schema.Types.ObjectId,ref: 'Patient',required: true},
    doctor: {type: mongoose.Schema.Types.ObjectId,ref: 'User',required: true},
    medCard: {type: String,ref: 'MedCard',required: true},
    diagnosis: {type: String,required: true },
    recommendations: {type: String,required: true},
    date: {type: Date,required: true}
  },
  {
    timestamps: true
  }
)
```

Рисунок 3.4 – Схема БД для медичних записів (MedCardRecord.model.js)

Схема запису на прийом містить: ідентифікатор пацієнта, ідентифікатор лікаря, дату, час та статус. Поля пацієнт, лікар, дата і час є обов'язковими й дозволяють зберігати інформацію про заплановані зустрічі. Значення поля статус обмежений переліком – Заплановано, Розпочато, Завершено, Скасовано. За замовчуванням, при створенні нового запису на прийом, полю статус буде присвоєно статус Заплановано. Код даної схеми, подано на рисунку 3.5

```
const schema = new mongoose.Schema(
  {
    patient: { type: mongoose.Schema.Types.ObjectId, ref: 'Patient', required: true },
    doctor: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true },
    date: { type: Date, required: true },
    time: { type: String, required: true },
    status: {
      type: String,
      enum: Object.values(APPOINTMENT_STATUS),
      default: APPOINTMENT_STATUS.PLANNED }
  },
  {
    timestamps: true
  }
)
```

Рисунок 3.5 – Схема БД запису на прийом (Appointment.model.js)

Окрім описаних полей, Mongoose може додавати до кожної схеми додаткові поля: _id – унікальний ідентифікатор для кожного документа. createdAt, updatedAt – дати створення та останнього оновлення документа. Такі поля є дуже важливими для всіх документів. Їх автоматичне створення дуже полегшує розробку та додатково покращує безпеку та ефективність бази даних.

Для наглядності, було створено UML-діаграму для вище описаних схем БД, яка відображена на рисунку 3.6

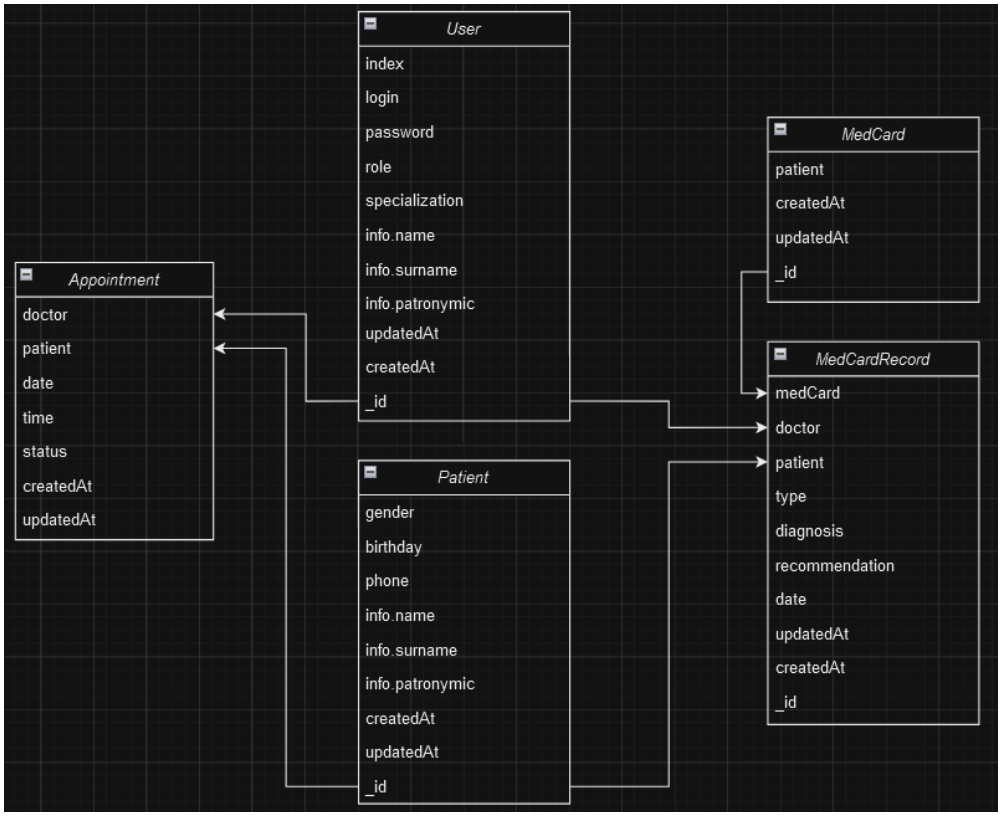


Рисунок 3.6 – UML-діаграма схем БД

3.1.2 Реалізація бізнес-логіки серверної частини

Наступним етапом розробки серверної частини застосунку, після створення схем бази даних, є створення API для взаємодії з клієнтською частиною застосунку. Насамперед варто розглянути API для користувачів. Бізнес-логіка міститься у шістьох сервісах:

UserService – створений для керування сутностями користувачів у базі даних. Код зображено на рисунку 3.7

```
const findAll = async () => {
  return await User.find({}).lean()
}
const findOne = async (filter) => {
  return await User.findOne(filter).lean()
}
const isUserExist = async ({ login }) => {
  const user = await User.countDocuments({ login })
  return user > 0
}

const create = async (data) => {
  const index = await User.countDocuments()
  return await User.create({ ...data, index })
}
const findUserById = async (userId) => {
  return await User.findById(userId).lean()
}
const findUserByLogin = async ({ login }) => {
  return await User.findOne({ login }).lean()
}
const updateUser = async (data) => {
  if (data.password.new || data.password.old) {
    const user = await User.findById(data.userId).lean()
    const isPasswordValid = await bcryptjs.compare(data.password.old, user.password)
    if (!isPasswordValid) throw new Error('Password is invalid')

    const hashedPassword = await bcryptjs.hash(data.password.new, 10)
    data.password = hashedPassword
  } else {
    delete data.password
  }

  await User.findByIdAndUpdate(data.userId, data)

  return await User.findById(data.userId).lean()
}
async function getDoctors(filter) {
  return await User.find(Object.assign({ role: ROLES.DOCTOR }, filter || {})).lean()
}
```

Рисунок 3.7 – Код сервісу UserService

Зображений на рисунку 3.7 сервіс – UserService, містить в собі методи:

findAll – Отримання всіх записів користувачів з бази даних

findOne – Пошук користувача на основі фільтра

isUserExist – Перевіряє, чи існує користувач з певним логіном у базі

create – Створює нового користувача з наданими даними при реєстрації.

findUserById – Знаходить користувача за його унікальним ідентифікатором.

updateUser – Оновлює особисті дані користувача для входу, де спочатку перевіряється валідність старого паролю перед його оновленням. Якщо пароль валідний, створюється хеш нового паролю та оновлюється в базі.

getDoctors – Видає список користувачів, які є лікарями

JwtService – створений для реалізації функцій шифрування та валідації JWT токенів, що забезпечують авторизацію користувача у системі. JWT складається з двох функцій:

createToken – де приймається об’єкт “userId” та створюється JWT. Також, містить “process.env.SECRET_KEY” – секретний ключ для підпису та “expiresIn” – вказує час на скільки токен буде дійсним.

verifyToken – де приймається токен та перевіряється його дійсність за допомогою методу “verify”.

Код описаного сервісу подано на рисунку 3.8

```
const createToken = ({ userId }) => {
  return jwt.sign({ userId }, process.env.SECRET_KEY, {
    expiresIn: '24h'
  })
}
const verifyToken = (token) => {
  let data
  jwt.verify(token, process.env.SECRET_KEY, (err, user) => {
    if (err) data = null
    data = user
  })
  return data
}
```

Рисунок 3.8 – Код сервісу JwtService

AuthService – містить методи для реєстрації та входу користувача в систему. Розглянемо метод реєстрації, насамперед у методі перевіряється чи вже існує користувач з наданим логіном, якщо так – повертається помилка. Після перевірки логіна на унікальність, створюється хешований пароль. За допомогою вже написаних UserService та JwtService створюється користувач та JWT токен. Успішним результатом роботи методу є об’єкт, що містить дані про нового користувача та JWT токен. Код відображено на рисунку 3.9

```
const signIn = async (credentials) => {
  const user = await UserService.findUserByLogin(credentials)
  if (!user) throw new Error('User not exist')
  if (credentials.role !== user.role) throw new Error('Role is invalid')
  const isPasswordValid = await bcryptjs.compare(credentials.password, user.password)
  if (!isPasswordValid) throw new Error('Password is invalid')
  const token = JwtService.createToken({ userId: user._id.toString() })
  return { ...user, token }
}
const signUp = async (data) => {
  const isUserExist = await UserService.isUserExist(data)
  if (isUserExist) throw new Error('User already exist')
  const hashedPassword = await bcryptjs.hash(data.password, 10)
  const user = await UserService.create({ ...data, password: hashedPassword })
  const token = JwtService.createToken({ userId: user._id.toString() })
  return { ...user.toObject(), token }
}
```

Рисунок 3.9 – Код сервісу AuthService

Далі був написаний PatientService, розлянемо його перші методи create, remove, update зображені на рисуеку 3.10

```
const create = async (data) => {
  const patient = await Patient.create(data)
  await MedCard.create({ patient: patient._id })
  return patient
}
const remove = async (id) => {
  await Patient.findByIdAndDelete(id)
  await MedCardService.removeByPatientId(id)
  await AppointmentService.removeAllByPatientId(id)
}

const update = async (data) => {
  await Patient.findByIdAndUpdate(data._id, data)
}
```

Рисунок 3.10 – Код методів create, remove та update з сервісу PatientService

create – Створює нового пацієнта та медичну картку зв'язану з цим пацієнтом із посиланням на ID пацієнта.

remove – Видаляє пацієнта за ID та видаляє всі пов'язані з ним медичні картки та призначення. Запускає додаткові методи з “MedCardService” та “AppointmentService” для повного очищення даних.

update – Оновлює інформацію про пацієнта, використовуючи ID для ідентифікації та оновлення запису.

Розглянемо наступний метод сервісу PatientService:

search – Виконує пошук пацієнтів на основі текстовго поля, яка може включати: ім'я, прізвище, побаткові або ID пацієнта. Також включає логіку для обробки дати народження (birthday.from та birthday.to), де для кожної встановлюються межі “$gte” (більше або дорівнює) та “$lte” (менше або дорівнює). Ці умови конвертуються у формат Date для коректної роботи з базою даних. Аналогічна логіка застосовується для фільтрації за датами створення записів (createdAt.from та createdAt.to). Також виконується запит до БД для отримання даних за допомогою функції “req”. Код описаного методу подано на рисунку 3.11

```
const search = async ({ search = '', filter }) => {
  const req = {}
  console.log(filter)
  if (filter) {
    if (filter.patients && filter.patients.length > 0) {
      req._id = { $in: filter.patients }
    }
    if (filter['birthday.from'] || filter['birthday.to']) {
      req.birthday = {}
      if (filter['birthday.from']) {
        req.birthday.$gte = new Date(filter['birthday.from'])
      }
      if (filter['birthday.to']) {
        req.birthday.$lte = new Date(filter['birthday.to'])
      }
      console.log(req.birthday)
    }
    if (filter['createdAt.from'] || filter['createdAt.to']) {
      req.createdAt = {}
      if (filter['createdAt.from']) {
        req.createdAt.$gte = new Date(filter['createdAt.from'])
      }
      if (filter['createdAt.to']) {
        req.createdAt.$lte = new Date(filter['createdAt.to'])
      }
    }
  }
```

Рисунок 3.11 – Код методу search з сервісу PatientService

MedCardService – сервіс, що забезпечує функціонал для управління медичними картками та записами. Сервіс включає бізнес-логіку для створення, пошуку, редагування та видалення медичних карток і записів пацієнтів. Містить в собі методи:

findAllRecords – забезпечує пошук усіх медичних записів за переданим фільтром, з попереднім завантаженням пов'язаних даних про лікаря, медичну картку та пацієнта.

findOne – Виконує пошук однієї медичної картки, яка відповідає фільтру, і повертає її як простий JavaScript об'єкт.

removeByPatientId – Видаляє медичну картку та всі пов’язані записи медичної картки пацієнта за ID пацієнта.

createMedCardRecord – Створює нові медичні записи у медичній карті, приймаючи дані запису та зберігаючи їх у базі даних.

updateMedCardRecord – Оновлює існуючі медичні записи за їх ідентифікатором, застосовуючи зміни до відповідних записів у базі даних.

deleteMedCardRecord – Знаходить та видаляє запис медичної картки за вказаним ID.

Код описаного сервісу подано на рисунку 3.12

```
const findAllRecords = async (filter) => {
  return await MedCardRecord.find(filter || {})
    .populate('doctor')
    .populate('medCard')
    .populate('patient')
    .lean()
}

const findOne = async (filter) => {
  return await MedCard.findOne(filter).lean()
}
async function removeByPatientId(patientId) {
  await MedCard.findOneAndDelete({ patient: patientId })
  await MedCardRecord.deleteMany({ patient: patientId })
}
async function createMedCardRecord(data) {
  return await MedCardRecord.create(data)
}
async function updateMedCardRecord(id, data) {
  return await MedCardRecord.findByIdAndUpdate(id, data)
}

async function deleteMedCardRecord(id) {
  return await MedCardRecord.findByIdAndDelete(id)
}
```

Рисунок 3.12 – Код описних методів з сервісу MedCardService

AppointmentService – останній розроблений сервіс, який надає функціонал для керування записами пацієнтів на прийом до лікарів. Він містить методи для створення нових записів, пошуку, редагування та видалення існуючих записів, а також виконання фільтрації та пошук за різними критеріями. Опис згаданих методів:

create – відповідає за створення нового запису на прийом, приймаючи дані про запис і зберігаючи їх у базі даних.

findAll і findOne – дозволяють отримувати всі записи або один запис за заданим фільтром відповідно. Вони використовують метод populate для завантаження пов'язаної інформації про пацієнтів та лікарів, що робить дані більш інформативними.

Код описаних методів зображено на рисунку 3.13

```
async function create(data) {
  return await Appointment.create(data)
}

const findAll = async () => {
  return await Appointment.find({}).populate('patient').populate('doctor').lean()
}
const findOne = async (filter) => {
  return await Appointment.findOne(filter).populate('patient').populate('doctor').lean()
}
```

Рисунок 3.13 – Код методів create, findAll та FindOne з сервісу AppointmentService

Наступні медоти сервісу PatientService:

update –дозволяє оновлювати інформацію про запис за його ідентифікатором, застосовуючи нові дані до відповідного запису в базі.

removeAllByPatientId видаляє всі записи на прийом, пов'язані з конкретним пацієнтом, що забезпечує захист персональних даних

remove видаляє конкретний запис за його ідентифікатором, забезпечуючи точкове видалення даних

Код описаних методів подано на рисунку 3.14

```
const update = async (id, data) => {
  return await Appointment.findByIdAndUpdate(id, data)
}
async function removeAllByPatientId(patientId) {
  await Appointment.deleteMany({ patient: patientId })
}
const remove = async (id) => {
  return await Appointment.findByIdAndDelete(id)
}
```

Рисунок 3.14 – Код методів update, removeAllByPatientId та remove з сервісу AppointmentService

3.1.3 Розробка контролерів, роутерів та middleware

Роутери та контролери в Express.js зазвичай містять менше бізнес-логіки та коду, що робить їх легко зрозумілими та підтримуваними. Вони зосереджуються на маршрутизації HTTP-запитів та передачі їх відповідним компонентам для виконання бізнес-логіки. Роутери визначають правила, за якими конкретні запити спрямовуються до контролерів на основі HTTP-методу та URL-шляху. Це дозволяє створювати чітку структуру застосунку, де кожен компонент виконує свою чітко визначену роль.

Приклад розроблено роутеру для пацієнта, подано на рисунку 3.15

```
const { Router } = require('express')
const { PatientController } = require('../controllers/patient.controller')
const router = Router()
router.post('/', PatientController.create)
router.post('/search', PatientController.search)
router.get('/', PatientController.findAll)
router.get('/:id/med-card', PatientController.findMedCardByPatientId)
router.put('/', PatientController.update)

router.get('/one', PatientController.findOne)
router.delete('/:id', PatientController.remove)
module.exports = { PatientRouter: router }
```

Рисунок 3.15 – Код роутера для пацієнта (patient.router.js)

Контролери отримують запити від роутерів і обробляють їх, викликаючи необхідні сервіси чи моделі. Вони працюють з даними, що містяться в запитах, такими як body, params і query. Body містить дані, передані в тілі запиту, наприклад, JSON, і використовується при створенні або оновленні ресурсів. Params містить параметри шляху, такі як ідентифікатори ресурсів у динамічних шляхах (наприклад, /patients/:id). Query містить параметри запиту, що передаються у рядку запиту URL, і зазвичай використовується для фільтрації або сортування даних.

Приклад розроблено контролеру для пацієнта, зображено на рисунку 3.16

```
const create = async (req, res) => {
  try {
    const data = await PatientService.create(req.body)
    res.send(data)
  } catch (error) {
    console.log(error)
    res.status(500).send({ error: error.message })
  }
}

const findOne = async (req, res) => {
  try {
    const data = await PatientService.findOne(req.query)
    res.send(data)
  } catch (error) {
    console.log(error)
    res.status(500).send({ error: error.message })
  }
}
```

Рисунок 3.16 – Код контролера для пацієнта (patient.controller.js)

Middleware призначений для авторизації є важливим компонентом в архітектурі Express.js, що забезпечує безпеку застосунку. Авторизаційне middleware перевіряє, чи має користувач відповідні права для виконання певної дії. Це включає перевірку токена за допомогою JwtService. Middleware виконується перед передачею запиту до контролера, забезпечуючи захист ресурсів від несанкціонованого доступу.

Код розробленого middleware, зображено на рисунку 3.17

```
const { JwtService } = require('../services/jwt.service')

const withAuth = (req, res, next) => {
  const authHeader = req.get('Authorization')
  console.log(authHeader)
  const token = authHeader && authHeader.split(' ')[1]

  if (token == null) return res.status(401).send({ error: 'Unauthorized: No Token' })

  const userData = JwtService.verifyToken(token)
  if (!userData) return res.status(403).send({ error: 'Unauthorized: Invalid Token' })
  req['user'] = userData
  return next()
}
module.exports = { withAuth }
```

Рисунок 3.17 – Код middleware (auth.middleware.js)

Завдяки такій організації роутери, контролери та middleware залишаються компактними та легкими для розуміння. Вони зосереджуються на маршрутизації та обробці запитів та перевірці права для виконання певної дії, делегуючи основну бізнес-логіку іншим компонентам системи. Це забезпечує високу модульність і дозволяє легко розширювати і підтримувати веб-застосунок, розділяючи різні аспекти роботи між спеціалізованими компонентами.

3.2 Розробка клієнтської частини застосунку

Клієнтська частина застосунку відіграє ключову роль у взаємодії з кінцевими користувачами, оскільки саме вона формує їхнє сприйняття та враження від продукту та є своєрідним "обличчям" застосунку. На відміну від серверної частини, яка зосереджена на обробці даних та взаємнодією з базою даних.

Як було описано в попередньому розділі, ахітектурою для розробки клієнтської частини було обрано FSD, яка розбиває код на незалежні функціональні модулі, тож це робить код більш структурованим, зручним у підтримці та масштабованим.

Розгянемо осноні елементи серверної частини по обраній архітектурі, в наступних підпунктах.

3.2.1 App

Модуль app – є основною точкою входу в додаток, він використовує React для створення інтерактивного користувацького інтерфейсу, та бібліотеку SWR для оптимізації запитів до сервера. Цей модуль включає декілька підмодулів, які забезпечують різні аспекти функціональності застосунку:

sign-in/sign-up – створює сторінку аутентифікації/реєстрації користувача, де імпортує SignInForm/SignUpForm з іншого модулю “widgets”. Код підмодулів подано на рисунку 13.18

```
import { Logo } from '@/shared/ui'
import { SignInForm } from '@/widgets/auth/ui/sign-in-form'

const page = () => {
  return (
    <main className='◦◦◦'>
      <Logo />
      <p className='◦◦◦'>
        Вітаємо! Авторизуйтесь для продовження{' '}
      </p>
      <section
        style={{
          boxShadow: '0px 4px 4px 0px #00000040',
        }}
        className='◦◦◦'
      >
        <SignInForm />
      </section>
    </main>
  )
}
```

```
import { Logo } from '@/shared/ui'
import { SignUpForm } from '@/widgets/auth/ui/sign-up-form'

const page = () => {
  return (
    <main className='◦◦◦'>
      <Logo />
      <p className='◦◦◦'>Реєстрація профілю</p>
      <section
        style={{
          boxShadow: '0px 4px 4px 0px #00000040',
        }}
        className='◦◦◦'
      >
        <SignUpForm />
      </section>
    </main>
  )
}

export default page
```

Рисунок 3.18 – Код підмодулів sign-in/sign-up

appointments – відповідає за управління та відображення записів на прийом. Детальний опис функціональностей:

Використовує хуки: useState - для зберігання стану фільтрів, useSWR для отримання даних лікарів та пацієнтів із сервера, useFilter для фільтрації записів на основі обраних критеріїв, usePagination для управління пагінацією, useSelectedTable для обробки вибору та видалення записів.

Виконує запити до API: api.get('/user/doctors') – для отримання списку лікарів, fetchAllPatients() – для отримання списку пацієнтів.

Фільтрація та пагінація: виконує ініціалізацію фільтрів за допомогою initialFilters, використовує “Filter” для відображення доступних фільтрів для користувача, а “Pagination” для управління пагінацією результатів.

Взаємодія з користувачем: AppointmentsTable – відображає список записів на прийом, CreateAppointment – дозволяє створювати нові записи, SearchAppointment – забезпечує пошук записів, useSelectedTable – для вибору та видалення записів, handleDeleteAppointments – обробляє видалення обраних записів через API.

Код підмодуля appointments подано на рисунках: 3.19 – 3.20.

```
import { STATUS_LABELS } from '@/shared/constants/status'
import { useFilter, usePagination } from '@/shared/hooks'
import { useSelectedTable } from '@/shared/hooks/use-selected-table'
import { Filter, Pagination } from '@/shared/ui'
import { UIButton } from '@/shared/uikit'
import { fullName } from '@/shared/utils'
import { useState } from 'react'
import { toast } from 'react-toastify'
import useSWR from 'swr'

const initialFilters = {
  patients: { options: [], type: 'select', label: 'Пацієнт' },
  doctors: { options: [], type: 'select', label: 'Лікар', disableRoles: [ROLES.DOCTOR] },
  statuses: { options: Object.entries(STATUS_LABELS).map(([value, label]) => ({ value, label })), type: 'select', label: 'Статус' },
  date: { type: 'date', label: 'Дата' },
}

const page = () => {
  const [filters, setFilters] = useState(initialFilters)

  useSWR('doctors', () => api.get('/user/doctors'), {
    onSuccess: ({ data }) => {
      setFilters(prev => ({
        ...prev,
        doctors: { ...prev.doctors, options: data.map(doctor => ({ value: doctor._id, label: fullName(doctor) })) },
      }))
    },
  })
  useSWR('patients', () => fetchAllPatients(), {
    onSuccess: data => {
      setFilters(prev => ({
        ...prev,
        patients: { ...prev.patients, options: data.map(patient => ({ value: patient._id, label: fullName(patient) })) },
      }))
    },
  })

  const filter = useFilter(filters)
  const { form, values, mutate } = useAppointmentSearch(filter.filter)
  const { pagination, handlers, sort } = usePagination(values.appointments)
  const handleDeleteAppointments = async ids => {
    try {
      for (const id of ids) await authApi.delete('/appointment/' + id)
      toast.success('Успішно')
      mutate()
    } catch {
      toast.error('Помилка')
    }
  }
```

Рисунок 3.19 – Код підмодуля appointments

```
const select = useSelectedTable(handleDeleteAppointments)

return (
  <>
    <h2>Журнал Записів</h2>
    <section className='◦◦◦'>
      <CreateAppointment
        mutate={mutate}
        buttonProps={{
          icon: { src: '/edit-date.svg', width: 24, height: 24, alt: '' },
          className: 'bg-[#9A49EA] text-white font-semibold text-sm gap-2 h-10'
        }}
        buttonText='Новий запис'
      />
      <SearchAppointment form={form} />
    </section>
    <section>
      <Filter filters={filters} form={filter.form} />
    </section>
    {select.ids.length > 0 && (
      <section className='◦◦◦'>
        <UIButton className={◦◦◦} type='button' onClick={select.click}>
          Видалити
        </UIButton>
        <UIButton type='button' className='◦◦◦■' onClick={select.reset}>
          Скасувати
        </UIButton>
      </section>
    )}
    <AppointmentsTable
      toggleSort={handlers.toggleSort}
      appointments={pagination.values}
      sortValues={sort}
      select={select}
    />
    <Pagination
      pages={pagination.pages}
      handlePageClick={handlers.handlePageClick}
```

Рисунок 3.20 – Код підмодуля appointments

med-card – підмодуль, який призначений для відображення та управління медичною карткою конкретного пацієнта.

Детальний опис функціональностей:

Використовує хуки: useState – для зберігання стану медичної картки, useEffect – для завантаження даних при монтуванні компонента, useRouter – допомагає отримати параметри маршруту (ідентифікатор пацієнта), що дозволяє завантажити відповідну медичну картку, useSWR – для отримання даних медичної картки з сервера та їх кешування.

Виконує запити до API: api.get('/med-card/${id}') – для завантаження медичної картки пацієнта за допомогою ідентифікатора з маршруту. Після отримання даних, вони зберігаються у стані компонента. Якщо користувач вносить зміни до медичної картки, запит api.put('/med-card/${id}') – відправляє ці зміни на сервер для збереження. Також містить функцію “handleSave”, яка відповідає за

збереження змін, здійснюючи API-запит та обробляючи результат, показуючи користувачу відповідні повідомлення про успіх або помилки.

Код описаного підмодуля подано на рисунку 3.21

```
'use client'
import { MedCardInfo } from '@/features/med-card/ui/med-card-info'
import { MedCardRecordsTable } from '@/features/med-card/ui/med-card-records-table'
import { PatientForm } from '@/features/patients/ui'
import { fetchMedCardByPatientId } from '@/shared/api/patient.api'
import { UIButton } from '@/shared/uikit'
import Image from 'next/image'
import { usePathname } from 'next/navigation'
import { useEffect, useState } from 'react'
import useSWR from 'swr'

const page = () => {
  const pathname = usePathname()
  const [id, setId] = useState('')
  useEffect(() => {
    setId(pathname.split('/').pop())
  }, [pathname])
  const { data: medCard, mutate } = useSWR(
    id ? '/med-card/patient' + id : null,
    () => fetchMedCardByPatientId(id)
  )
  if (!id || !medCard) return null
  return (
    <div className='◦◦◦'>
      <section className='◦◦◦'>
        <UIButton className='◦◦◦ '>
          <Image src='/save.svg' width={20} height={16} alt='' />
          Медична Карта
        </UIButton>
        <PatientForm
          mutate={mutate}
          type='update'
          defaultValues={medCard.patient}
        />
      </section>
      <MedCardInfo medCard={medCard} />
      <MedCardRecordsTable id={id} />
```

Рисунок 3.21 – Код підмодуля med-card

patients – підмодуль, який відповідає за управління та відображення списку пацієнтів. Він надає можливість шукати, фільтрувати, додавати та переглядати інформацію про пацієнтів.

Використовує хуки: useState – для зберігання стану фільтрів, useFilter – для фільтрації списку пацієнтів на основі обраних критеріїв, useSWR – для отримання даних пацієнтів із сервера, usePagination - для управління пагінацією.

Виконує запити до API: fetchAllPatients – для отримання списку пацієнтів із сервера.

Фільтрація та пагінація: Встановлює початкові фільтри за допомогою об'єкта “initialFilters”, що включає опції для пацієнтів, дати народження та дати створення, а “Filter” – для відображення доступних фільтрів для користувача

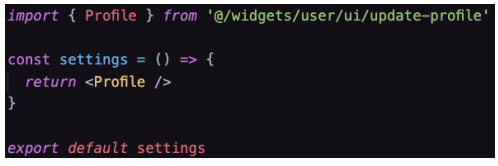
Взаємодія з користувачем: PatientsTable – відображає список пацієнтів із можливістю сортування та пагінації. PatientForm – дозволяє створювати нового пацієнтів, SearchPatient – забезпечує пошук пацієнтів.

Код описаного підмодуля “patients” відображено на рисунку 3.22

```
import { useFilter, usePagination } from '@/shared/hooks'
import { Filter, Pagination } from '@/shared/ui/'
import { fullName } from '@/shared/utils'
import { useState } from 'react'
import useSWR from 'swr'

const initialFilters = {
  patients: { options: [], type: 'select', label: 'Пацієнт' },
  birthday: { type: 'date', label: 'Дата народження' },
  createdAt: { type: 'date', label: 'Дата створення' },
}

const page = () => {
  const [filters, setFilters] = useState(initialFilters)
  const filter = useFilter(filters)

  useSWR('patients', () => fetchAllPatients(), {
    onSuccess: data => {
      setFilters(prev => ({
        ...prev,
        patients: {
          ...prev.patients,
          options: data.map(patient => ({
            value: patient._id,
            label: fullName(patient),
          })),
        },
      }))
    },
  })

  const { form, values, mutate } = usePatientSearch(filter.filter)
  const { pagination, handlers, sort } = usePagination(values.patients)

  return (
    <div className='◦◦◦'>
      <h2>Журнал Пацієнтів</h2>
      <section className='◦◦◦'>
        <PatientForm type={'create'} mutate={mutate} />
        <SearchPatient form={form} />
      </section>
      <section>
        <Filter filters={filters} form={filter.form} />
      </section>
      <PatientsTable
        sortValues={sort}
        toggleSort={handlers.toggleSort}
        patients={pagination.values}
        mutate={mutate}
```

Рисунок 3.22 – Код підмодуля patients

settings – підмодуль, який відповідає за редагування профілю користувача, а саме зміна: логіну, паролю, спеціалізації, ПІБ. Принцип дії: імпортує необхідний компонент “Profile” з /widgets/user/ui/update-profile та використовує його для відображення та оновлення профілю користувача. Код подано на рисунку 3.23



```
import { Profile } from '@/widgets/user/ui/update-profile'

const settings = () => {
  return <Profile />
}

export default settings
```

Рисунок 3.23 – Код підмодуля settings

3.2.2 Features

Модуль features – організовує код за функціональними одиницями, щоб полегшити управління та повторне використання компонентів і логіки. Містить в собі компоненти: UI – для побудови інтерфейсу користувача для конкретних функцій, таких як форми для введення даних, таблиці для відображення інформації, Хуки – які реалізують логіку, пов'язану з конкретними функціями, наприклад, хуки для завантаження даних з API.

Розглянемо підмодулі fetures:

appointment – містить в собі наступні компоненти:

UI:

appointment-change-status.jsx – для зміни статусу прийому(started, planned, finished, canceled) та виклику помилки якщо не вдалось змінити статус. Код подано на рисунку 3.24

Рисунок 3.24 – Код компонента appointment-change-status.jsx

appointments-table.jsx – для відображення таблиці записів на прийом. Містить в собі відображення колонок пацієнт, спеціалізація, лікар, дата, час, статус, дія.

Код подано на рисунку 3.25.

```
const columns = [
  {name: 'ID', w: 'w-44'},
  {
    name: 'Пацієнт', w: 'w-56',
    sort: {field: 'patient', type: 'fullname'}
  },
  {
    name: 'Спеціалізація', w: 'w-44',
    sort: {field: 'doctor.specialization', type: 'string'}
  },
  {
    name: 'Лікар', w: 'w-56',
    sort: {field: 'doctor', type: 'fullname'}
  },
  {
    name: 'Дата', w: 'w-24',
    sort: {field: 'date', type: 'date'}
  },
  {
    name: 'Час', w: 'w-16',
    sort: {field: 'time', type: 'time'}
  },
  {
    name: 'Статус', w: 'w-30',
    sort: {field: 'status', type: 'string'}
  },
  {name: 'Мед. карта', w: 'w-24'},
  {name: 'Дія', w: 'w-8'}
]
```

Рисунок 3.25 – Код компонента appointments-table.jsx

create-appointment.jsx – для створення нових записів на прийом. Містить useForm – для створення форми та управління її станом, useState – для управління локальним станом компонента, onSubmit – виконує запит до сервера для створення нового запису на прийом. Код подано на рисунку 3.26

```
export const CreateAppointment = ({
  patient,
  buttonText = 'Записати',
  buttonProps,
  mutate,
}) => {
  const [isOpen, setIsOpen] = useState(false)
  const { user, loaded } = useAuth()
  const [defaultValues, setDefaultValues] = useState(null)
  useEffect(() => {
    if (loaded && user) {
      const values = {}
      if (user.role === ROLES.DOCTOR) values.doctor = user._id
      if (patient) values.patient = patient._id
      setDefaultValues(values)
    }
  }, [user, loaded])
  if (user && user?.role !== ROLES.OPERATOR) return null
  return (
    <Fragment>
      <UIButton
        type='button'
        onClick={() => setIsOpen(true)}
        className='◦◦◦'
        {...buttonProps}
      >
        {buttonText}
      </UIButton>

      {isOpen && defaultValues && (
        <div className='◦◦◦'>
          <div className='◦◦◦'>
            <h2 className='◦◦◦'>Запис на прийом</h2>
            <Form
              defaultValues={defaultValues}
              closeModal={() => setIsOpen(false)}
              mutate={mutate}
            />
          </div>
        </div>
      )}
    </Fragment>
  )
}
```

Рисунок 3.26 – Код компонента create-appointment.jsx

hooks: use-appointment-search.js – відповідає за логіку пошуку записів на прийом, де виконує запит до API – використовуючи useSWR та fetchSearchedAppointments – для отримання даних. Також використовує, useEffect для ініціалізації стану та виклику функцій при зміні певних залежностей. Код зображено на рисунку 3.27

```
export function useAppointmentSearch(filter) {
  const [isInitiated, setInitiated] = useState(false)
  const { data: appointments, mutate } = useSWR(
    isInitiated ? 'appointments' : null,
    () => fetchSearchedAppointments({ ...values, filter })
  )
  const { user, loaded } = useAuth()

  const form = useForm()
  const values = form.watch()

  useEffect(() => {
    if (isInitiated) mutate()
  }, [values, filter])
  useEffect(() => {
    if (loaded && user) {
      if (user.role === ROLES.DOCTOR) form.setValue('doctor', user._id)
      setInitiated(true)
    }
  }, [user])
```

Рисунок 3.27 – Код компонента use-appointment-search.js

Розглянемо наступний підмодуль:

auth – містить в собі наступні компоненти:

UI:'

sign-out-button.jsx – компонент, який відповідає за вихід користувача з системи. Використовує хук “useAuth” та функцію “handleSignOut” та викликає для “signOut” для виходу користувача з системи. Код поданона рисунку 3.28

```
export const SignOutButton = () => {
  const router = useRouter()
  const { signOut } = useAuth()
  const handleSignOut = () => {
    router.replace('/sign-in')
    signOut()
  }
  return (
    <UIButton
      type='button'
      onClick={handleSignOut}
      className='◦◦◦'
    >
      <Image src={'/sign-out.svg'} width={24} height={24}
      <p>Вийти</p>
    </UIButton>
```

Рисунок 3.28 – Код компонента sign-out-button.jsx

auth.store.js – відповідає за управління станом аутентифікації користувачів у додатку. Він використовує бібліотеку Zustand для управління станом та зберігання даних у локальному сховищі. Використовує “persist”для забезпечення збереження стану в локальному сховищі браузера. Код відображено на рисунку 3.29

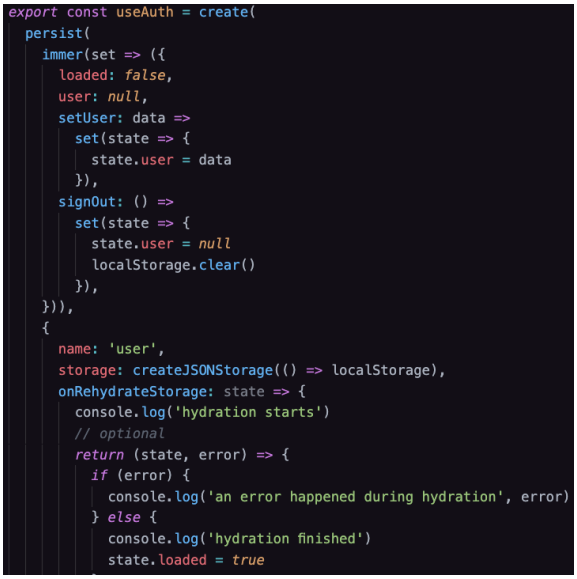


Рисунок 3.29 – Код компонента auth.store.js

Далі розгянемо наступний підмодуль –med-card, який містить в собі наступні компоненти:

med-card-info.jsx – відображає інформацію медичної картки пацієнта, імпортує: id пацієнта та його особисту інформація як: ім’я, прізвище і тд. (див.рисунок 3.30).

Рисунок 3.30 – Код компонента med-card-info.jsx

med-card-record.jsx – виконує створення, оновлення та перегляд записів медичної картки, залежно від ролі користувача та типу дії. До прикладу на рисунку 3.31, функція створення мед. запису доступна лише для лікарів допомогою (type 'watch')

```
export const MedCardRecord = ({
  buttonText = 'Створити запис',
  buttonProps,
  patientId,
  record = null,
  type = 'create',
  buttonClassName = '',
}) => {
  const [isOpen, setIsOpen] = useState(false)
  const { user, loaded } = useAuth()
  const [defaultValues, setDefaultValues] = useState(null)
  useEffect(() => {
    if (loaded && user) {
      const values = { patient: patientId }
      if (user.role === ROLES.DOCTOR) values.doctor = user._id
      if (record) {
        const patientId = record.patient._id
        const doctorId = record.doctor._id
        const medCardId = record.medCard._id
        if (typeof record.date === 'string') record.date = new Date(record.date)
        if (patientId) record.patient = patientId
        if (doctorId) record.doctor = doctorId
        if (medCardId) record.medCard = medCardId
        setDefaultValues(record)
      } else {
        setDefaultValues(values)
      }
    }
  }, [user, loaded, record])
  if (!user || (user.role !== ROLES.DOCTOR && type !== 'watch')) return null
```

Рисунок 3.31 – Код компонента med-card-record.jsx

med-card-records-table.jsx – призначений для відображення таблиці медичних записів пацієнта. Дані завантажуються за допомогою “useSWR”, який автоматично кешує та оновлює дані при необхідності. Цей хук звертається до fetchMedCardRecords, передаючи ID пацієнта. Використовується “useMemo” для оптимізації рендерингу рядків таблиці. Для кожного запису формується об'єкт з потрібними даними, які потім передаються в компонент “Table. Для кожного запису “ActionsWithMedCardRecord” надає набір дій, таких як редагування чи видалення, залежно від контексту.”

Код подано описаного компонента подано на рисунку 3.32

```jsx
export const MedCardRecordsTable = ({ id }) => {
  const { data: records } = useSWR(
    'med-card/record/' + id,
    () => fetchMedCardRecords({ patient: id }),
    { onSuccess: console.log }
  )
  const { pagination, handlers, sort } = usePagination(records)
  const rows = useMemo(
    () =>
      pagination.values.slice().map(record => {
        console.log('table', record.type)
        return {
          cells: [
            fullName(record.patient),
            fullName(record.doctor),
            record.type,
            new Date(record.createdAt).toLocaleString(),
            <ActionsWithMedCardRecord record={record} />,
          ],
        }
      }),
    [pagination.values, records]
  )
  return (
    <>
      <section className='◦◦◦'>
        <MedCardRecord buttonClassName='absolute top-1.5' patientId={id} />
        <h2 className='◦◦◦'>
          Медичні записи
        </h2>
        <div className='◦◦◦'>
          <Table
            sortValues={sort}
            toggleSort={handlers.toggleSort}
            columns={columns}
            rows={rows}
            tableClassName='h-max'
          />
        </div>
      </section>
      <Pagination
        pages={pagination.pages}
        handlePageClick={handlers.handlePageClick}
```

Рисунок 3.32 – Код компонента med-card-records-table.jsx

actions-with-med-card-record.jsx – надає інтерфейс для взаємодії з конкретними записами медичної картки, включаючи можливості перегляду, редагування та видалення записів. Цей компонент використовується залежно від ролі користувача, зокрема, за допомогою “useMemo” перевіряється, чи є користувач лікарем (ROLES.DOCTOR), тим самим дозволяючи редагувати (за допомогою виклику компонента з “MedCardRecord” з параметром “type=update”) та видаляти записи (за допомогою виклику функції “handleDelete”, яка асинхронно викликає API для видалення запису, використовуючи “deleteMedCardRecord”). Додатково виводиться повідомлення про успішність або не успішність операції.

Код описаного компоненту подано на рисунку 3.33

```jsx
export const ActionsWithMedCardRecord = ({ record }) => {
  const { user } = useAuth()
  const isDoctor = useMemo(() => user?.role === ROLES.DOCTOR, [user])
  const handleDelete = async () => {
    try {
      await deleteMedCardRecord(record._id)
      mutate('med-card/record/' + record?.patient)
      toast.success('Успішно!')
    } catch (e) {
      console.log(e)
      toast.error('Помилка :/')
    }
  }
  console.log('actions', record.type)
  return (
    <ul className='◦◦◦'>
      <li>
        <MedCardRecord
          buttonText='Відкрити'
          buttonProps={{
            className:
              'bg-green_2 text-black_2 text-sm items-center gap-2 h-6 px-1',
            icon: null,
          }}
          patientId={record.patient._id}
          record={record}
          type='watch'
        />
      </li>
      {isDoctor && (
        <>
          <li>
            <MedCardRecord
              buttonText='Редагувати'
              buttonProps={{
                className:
                  'bg-orange_1 text-black_2 text-sm items-center gap-2 h-6 px-1',
                icon: null,
              }}
              patientId={record.patient._id}
              record={record}
              type='update'
            />
          </li>
          <li>
            <UIButton
              type='button'
              onClick={handleDelete}
```

Рисунок 3.33 – Код компонента actions-with-med-card-record.jsx

Розглянемо наступний підмодуль – patients, який містить в собі наступний компонент:

patient-form.jsx – призначений для створення нових пацієнтів та редагування даних їх даних існують. Використовує: useState – для контролю відображення модального вікна форми, useForm – для управління введенням даних, валідацією та поданням форми. Форма включає поля для введення особистих даних, статі, дати народження та контактного номера. ReactDatePicker – використовується для вибору

дати народження. onSubmit – дані з форми перевіряються та відправляються через API для створення або оновлення запису пацієнта.

Код описаного підмодулю подано на рисунку 3.34

```
const onSubmit = async data => {
  try {
    if (type === 'create')
      await authApi.post('/patient', {
        ...data,
        birthday: startDate,
      })
    if (type === 'update')
      await authApi.put('/patient', {
        ...data,
        birthday: startDate,
      })
    mutate && mutate()
    toast.success('Успішно')
  } catch (e) {
    console.log(e?.message)
    toast.error('Помилка')
  } finally {
    closeModal()
  }
}
const onInvalid = () => {
  console.log(errors)
  toast.error('Помилка')
}
useEffect(() => {
  reset()
  return () => {
    reset()
  }
}, [])
useEffect(() => {
  if (defaultValues) {
    reset(defaultValues)
    setStartDate(new Date(defaultValues.birthday))
  }
}, [defaultValues])
console.log(defaultValues)
return (
  <form onSubmit={handleSubmit(onSubmit, onInvalid)}>
    <section className='◦◦◦'>
      <div className='◦◦◦'>
        <UITextField
          required
          variant='rounded'
          label='Прізвище'
          inputProps={register('info.surname')}
        />
        <UITextField
          required
          variant='rounded'
          label='Ім’я'
          inputProps={register('info.name')}
        />
        <UITextField
          variant='rounded'
          label='Побатькові'
          inputProps={register('info.patronymic')}
```

Рисунок 3.34 – Код компонента actions- patient-form.jsx

3.2.3 API

API – важлива частина веб-застосунку, вона забезпечує взаємодію між клієнтською частиною та сервером. Веб-застосунок використовує API для обміну даними, що дозволяє клієнтському додатку отримувати, створювати, оновлювати та видаляти інформацію на сервері. Це забезпечує динамічність і інтерактивність застосунку, оскільки дозволяє користувачам виконувати різноманітні дії, такі як реєстрація, авторизація, управління профілем та інші операції з даними. Використання API сприяє модульності та гнучкості архітектури застосунку, спрощує розробку та обслуговування, а також покращує масштабованість і безпеку системи.

Розгянемо реалізацію API для med-card.api.js, та його функції, специфічні для взаємодії з медичними картками в системі:

createMedCardRecord(data) – Запит POST для створення нового запису в медичній картці. “data” містить інформацію, яка буде записана, наприклад: діагноз, рекомендації лікування, дата візиту тощо.

updateMedCardRecord(recordId, data) – запит PUT для оновлення існуючого запису, де “recordId” вказує на конкретний запис, який потрібно оновити, а “data” містить нову інформацію для цього запису.

deleteMedCardRecord(recordId) – Запит DELETE для видалення запису з медичної картки, де “recordId” є ідентифікатором запису, який потрібно видалити.

fetchMedCardRecords({ patientId }) – Запит “GET” для отримання всіх записів медичної картки конкретного пацієнта. Параметр “patientId” вказує, чиї записи потрібно отримати.

fetchMedCardRecord({ patientId }) – Запит “GET” для отримання всіх записів медичної картки конкретного пацієнта. Параметр “patientId” вказує, чиї записи потрібно отримати.

Код описаного API для med-card.api.js подано на рисунку 3.35

```
import { api } from '.'

export const createMedCardRecord = async data => {
  const result = await api.post('med-card/record', data)
  return result.data
}
export const updateMedCardRecord = async (id, data) => {
  const result = await api.put(`med-card/record/${id}`, data)
  return result.data
}
export const deleteMedCardRecord = async id => {
  const result = await api.delete(`med-card/record/${id}`)
  return result.data
}
export const fetchMedCardRecords = async data => {
  const result = await api.get('med-card/record', { params: data })
  return result.data
}
export const fetchMedCardRecord = async data => {
  const result = await api.get('med-card/record', { params: data })
  return result.data
}
```

Рисунок 3.35 – Код API для med-card.api.js

Розгянемо реалізацію API для appointment.api.js, та його функції, специфічні для взаємодії із записами пацієнтів на прийом до лікаря:

fetchAllAppointments(params) – Запит “GET” для отримання всіх записів на прийом. Приймає параметри, які можуть бути використані для фільтрації результатів.

fetchSearchedAppointments(data) – Запит “POST” для пошуку записів на прийом на основі переданих даних.

createAppointment(data) – Запит “POST” для створення нового запису на прийом.

updateAppointment(id, data) – Запит “PUT” для оновлення існуючого запису на прийом. Вона приймає ідентифікатор запису та нові дані, що мають бути оновлені.

Код описаного API для appointment.api.js подано на рисунку 3.36

```
import { api } from '.'

export const fetchAllAppointments = async params => {
  const result = await api.get('appointment', { params: params ? params : {} })
  return result.data
}

export const fetchSearchedAppointments = async data => {
  const result = await api.post('appointment/search', data)
  return result.data
}

export const createAppointment = async data => {
  const result = await api.post('appointment/', data)
  return result.data
}

export const updateAppointment = async (id, data) => {
  const result = await api.put(`appointment/${id}`, data)
  return result.data
}
```

Рисунок 3.36 – Код API для med-card.api.js

Розгянемо реалізацію API для patient.api.js, та його функції, специфічні для взаємодії із пацієнтами медичної клініки:

fetchAllPatients(params) – Запит “GET” для отримання списку всіх пацієнтів. params - параметри запиту, які можна передати для фільтрації результатів.

fetchSearchedPatients(data) – Запит “POST” для пошуку пацієнтів за заданими параметрами.

fetchMedCardByPatientId(id) – Запит “GET” для отримання медичної картки пацієнта за його ID.

Код описаного API для patient.api.js подано на рисунку 3.37

```
import { api } from '.'

export const fetchAllPatients = async params => {
  const result = await api.get('patient', { params: params ? params : {} })
  return result.data
}

export const fetchSearchedPatients = async data => {
  const result = await api.post('patient/search', data)
  return result.data
}

export const fetchMedCardByPatientId = async id => {
  const result = await api.get(`patient/${id}/med-card`)
  return result.data
}
```

Рисунок 3.37 – Код API для patient.api.js

Також було створено файл базового налаштування для HTTP-клієнтів, що використовуються в проекті. Тут створюються два основних екземпляри клієнта “axios”, один загальний (який не потребує авторизації) та один авторизований (authApi), що додає токен авторизації до заголовків запиту, якщо токен є у локальному сховищі.

Код описаного файлу налаштувань – index.js подано на рисунку 3.38

```
import axios from 'axios'
const baseURL = 'http://localhost:3001/'
export const api = axios.create({
  baseURL,
  headers: {
    'Content-Type': 'application/json',
  },
})
export const authApi = axios.create({
  baseURL,
  headers: {
    'Content-Type': 'application/json',
  },
})
authApi.interceptors.request.use(config => {
  const token = localStorage.getItem('token')
  if (token) {
    config.headers.Authorization = `Bearer ${token}`
  }
  return config
})
```

Рисунок 3.38 – Файл налаштувань для API (index.js)

4 ТЕСТУВАННЯ ВЕБ-ЗАСТОСУНКУ

4.1 Тестування API

Тестування API є важливим етапом у процесі розробки програмного забезпечення, особливо коли йдеться про забезпечення коректної роботи взаємодії між клієнтом та сервером. Для тестування API було обрано інструмент REST Client, який дозволяє зручно створювати та виконувати HTTP-запити безпосередньо з текстового редактора, що значно спрощує процес тестування.

Перейдемо до тестування основних функцій:

Спочатку здійснимо перевірку можливості реєстрації користувача у системі, для цього створимо відповідний запит. Результатом, буде успішна реєстрація, варто зазначити, що пароль користувача надійно зашифровується в БД (див.рисунок 4.1).

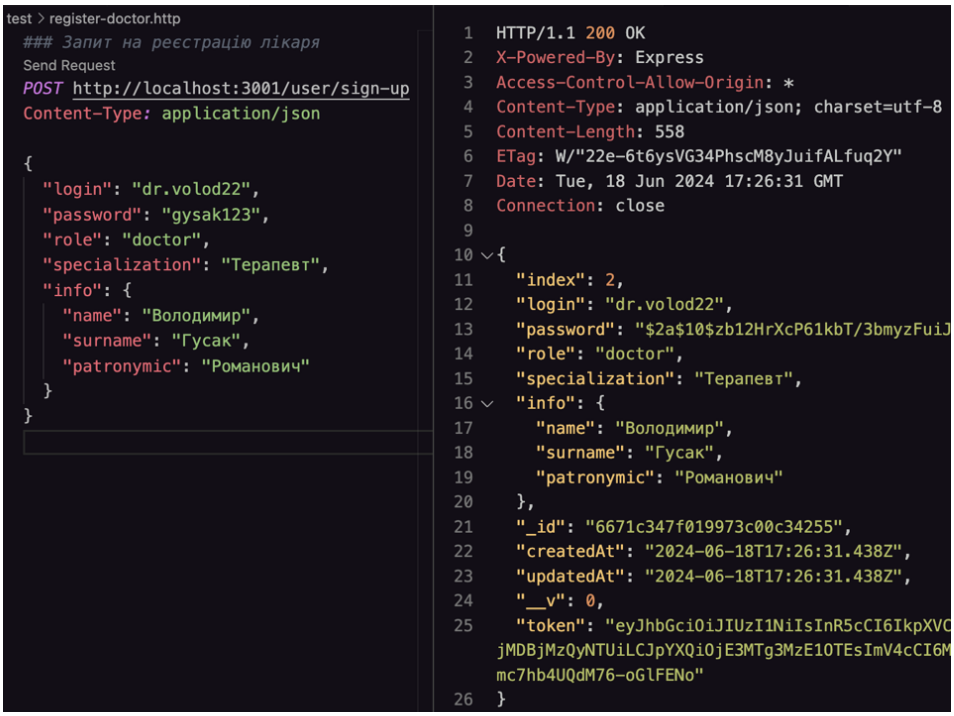


Рисунок 4.1 – Здійснення API-запиту для реєстрації

Далі перевіримо можливість авторизації користувача у системі, для цього створимо відповідний запит. Результатом, буде успішна аторизація (див.рисунок 4.2).

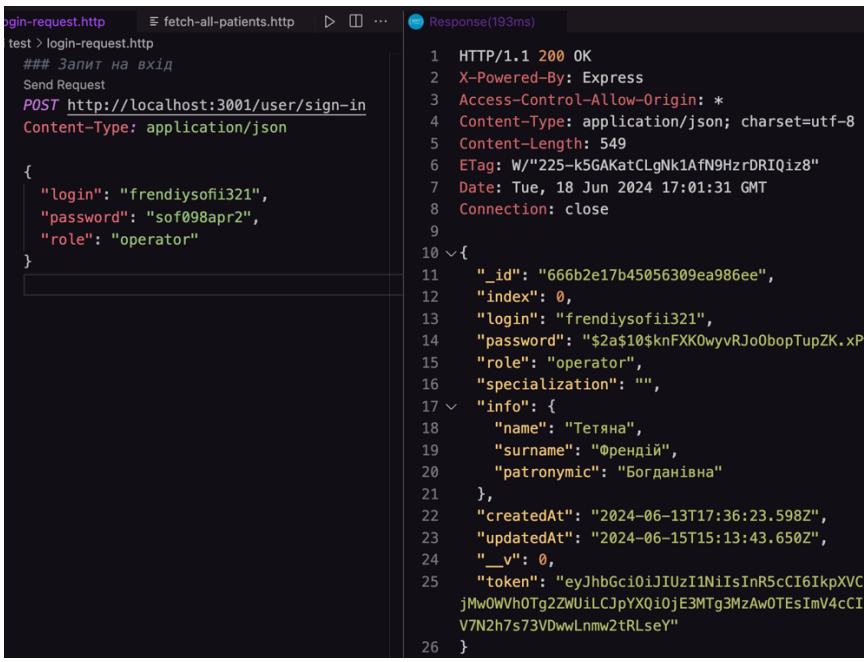


Рисунок 4.2 – Здійснення API-запиту для авторизації

Пісял цього перевіримо можливість перегляду всіх пацієнтів клініки, для цього сформуємо відповідний запит. В результаті буде відображено список пацієнтів з їхніми особистими даними (див.рисунок 4.3).

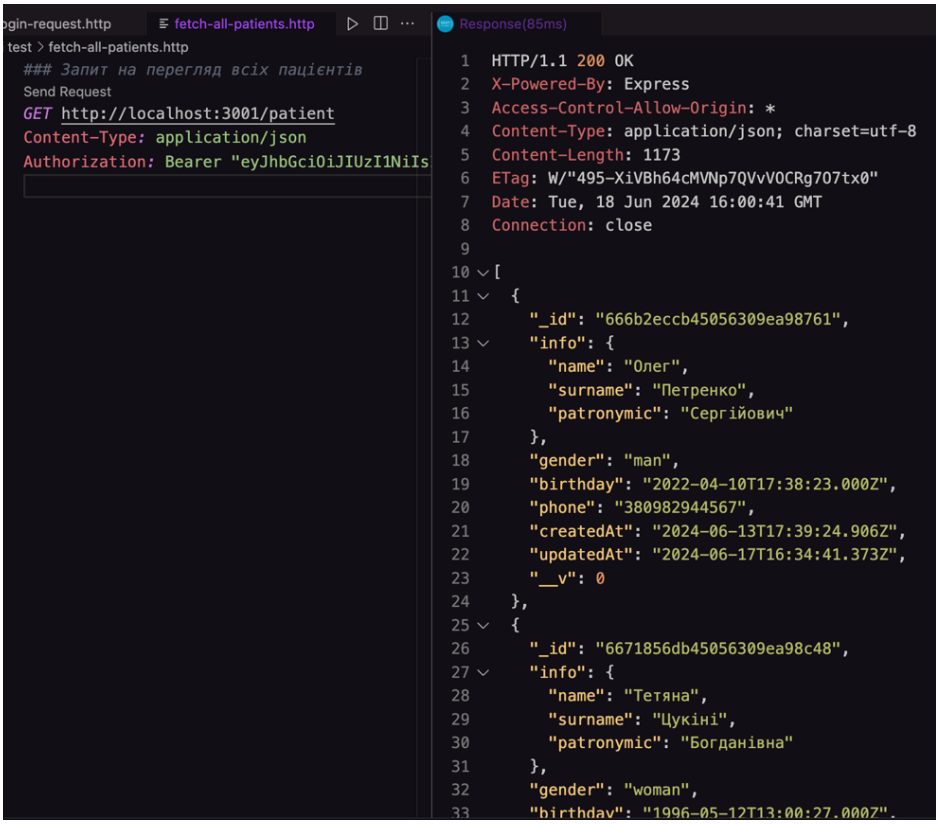


Рисунок 4.3 – Здійснення API-запиту для перегляду всіх пацієнтів клініки

Тепер спробуємо записати пацієнта на прийом до лікаря, створивши необхідний запит. В результаті, можемо переглянути створений запис на прийом, де вказується ID пацієнта та лікаря, дата, час, встановлюється час створення та оновлення запису. За замовчуванням встановлюється статус “planned” – заплановано, який вподальшому може змінюватись користувачами.

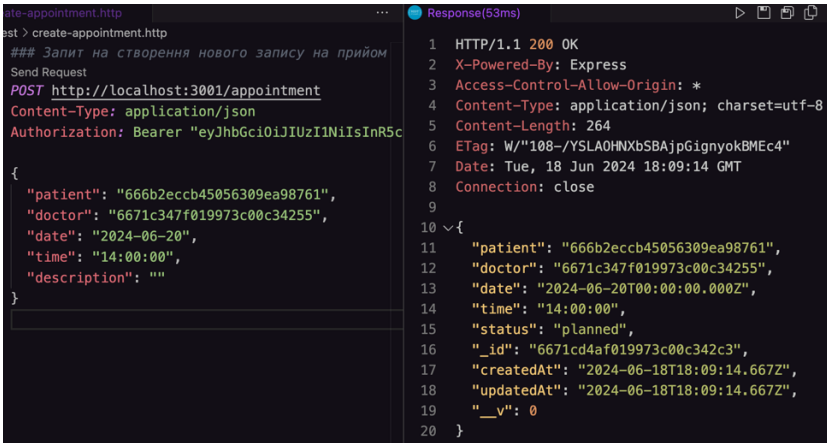


Рисунок 4.4 – Здійснення API-запиту для запису пацієнта на прийом до лікаря

4.2 Тестування інтерфейсу

Щоб розпочати тестування графічної частини, перейдемо на сторінку реєстрації, де нам запропонують ввести персональні дані (див.рисунок 4.5)

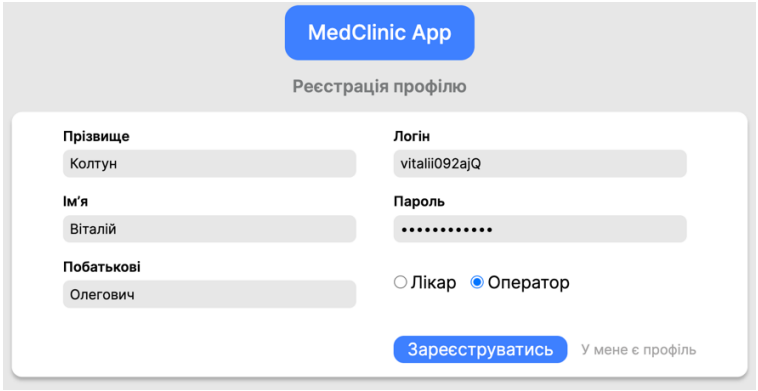


Рисунок 4.5 – Введення персональних даних у форму реєстрації

Якщо користувач вже має зареєстрований профіль, він може скористатися опцією “У мене є профіль” – щоб увійти у профіль. Форма авторизації подана на рисунку 4.6

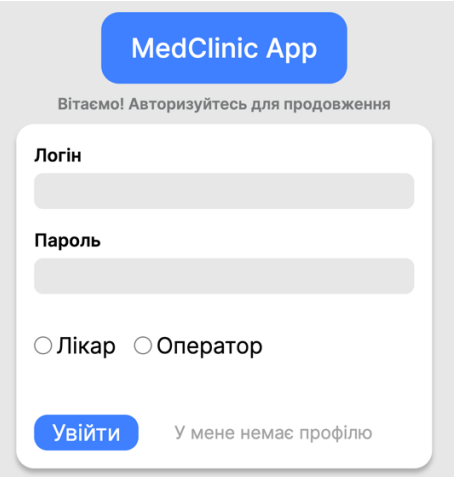


Рисунок 4.6 – Форма авторизації веб-застосунку

Після успішної авторизації або реєстрації, нас автоматично перенаправлятимн на сторінку “Пацієнти клініки”. Результат подано на рисунку 4.7

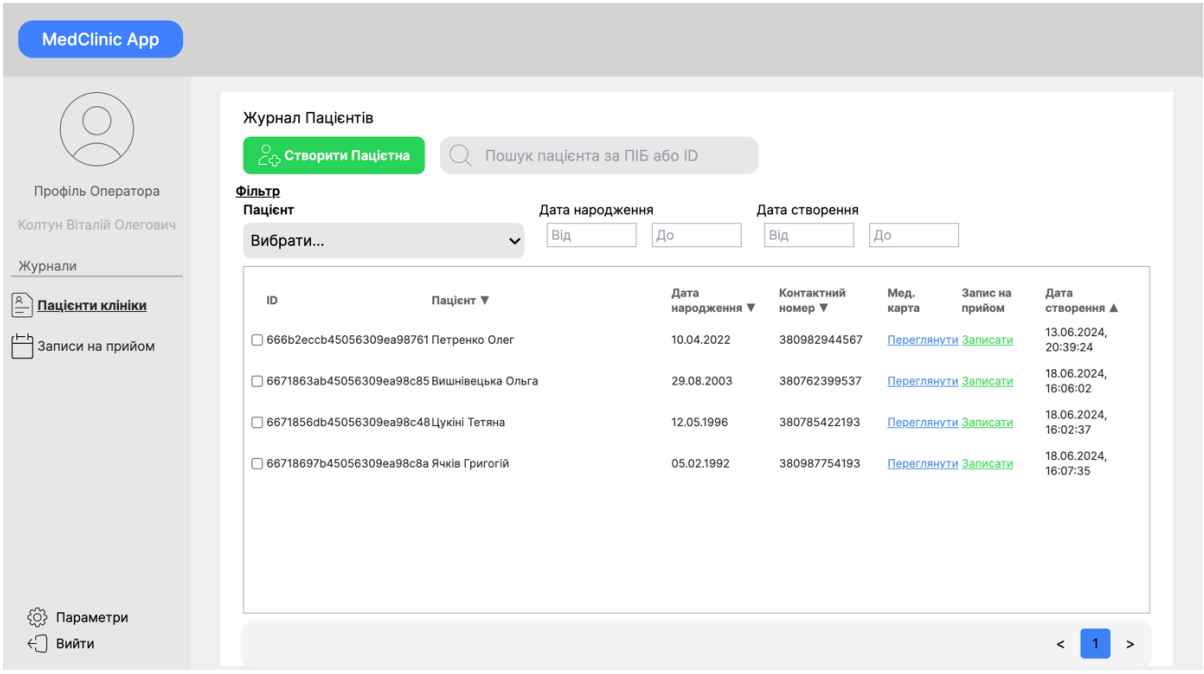


Рисунок 4.7 – Сторінка “Пацієнти клініки”

Створимо нового пацієта, для цього на сторінці “Пацієнти клініки” натисними на кнопку “Створити Пацієнта”, яка викличе форму для створення. Результатом, буде відкриття форми для створення, яка містить поля для вводу, деякі з них обов’язкові як: Прізвище, Ім’я та дата народження (див. рисунок 4.8)

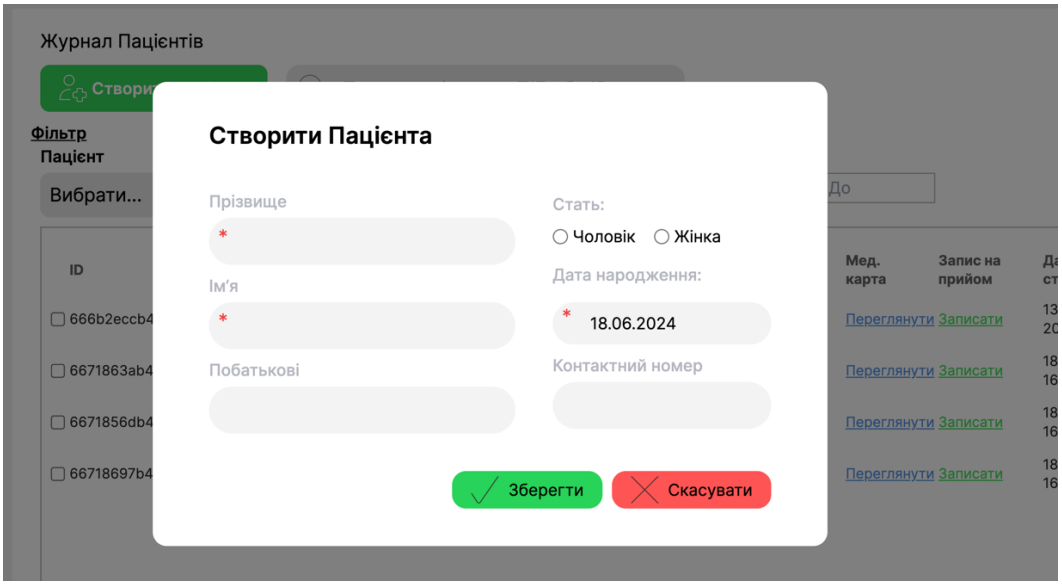


Рисунок 4.8 – Форма “ Створити Пацієнта ”

Тепер спробуємо, створити пацієнта не вказавши ім’я. Результат: перевірка пропущеного поля пройшла успішно і ми можемо побачити помилку на рисунку 4.9

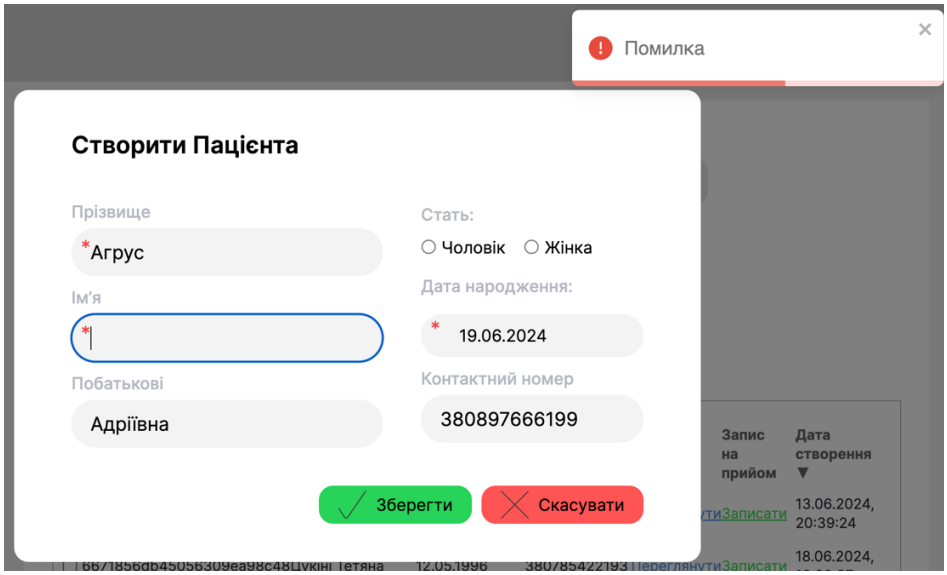


Рисунок 4.9 – Перевірка форми “ Створити Пацієнта ” на пропущене поле

Після цього, введемо дійсні дані для створення пацієнта та натиснемо на кнопку “Зберегти”. Результат, як подано на рисунку 4.10 – позитивний, бачимо доданого пацієнта в таблиці та повідомлення про успішність.

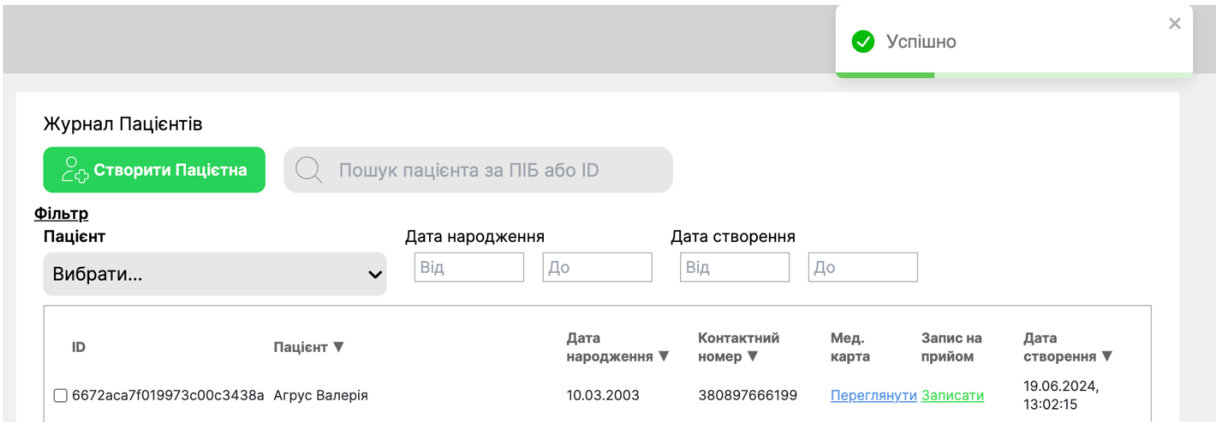


Рисунок 4.10 – Упішне створення нового пацієнта

Тепер перейдемо до перегляду мед.карти пацієнта, для цього на тиснемо на кнопку “переглянути” (див. рисунок 4.11)

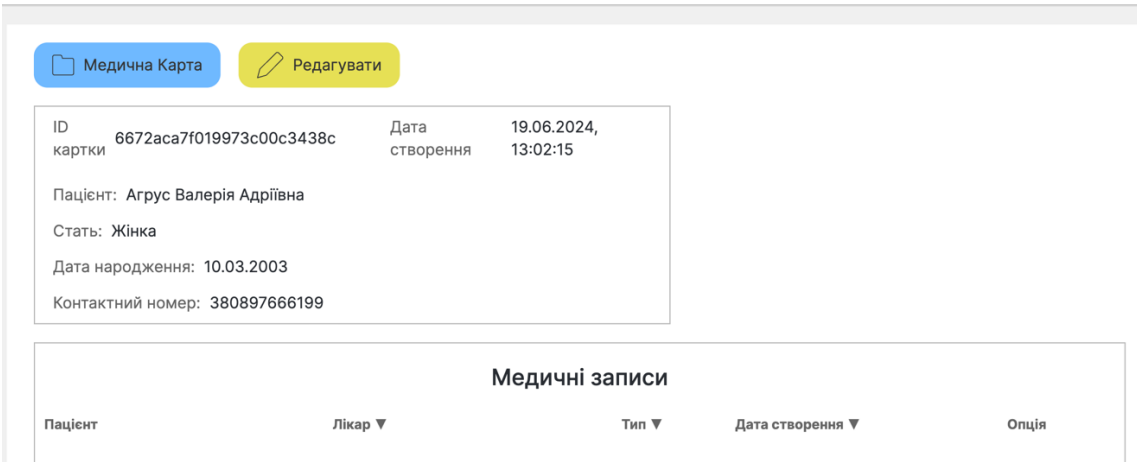


Рисунок 4.11 – Перегляд мед.карти пацієнта

Якщо була зроблена помилка при створені пацієнта, то її легко усунути настиснувши на кнопку “редагувати” після цього, нам відкриється відповідна форма “Редагувати Пацієнта”. У формі доступні дві опції “Зберегти” та Скасувати” після збереження висвічується повідомлення про успішність.

Результат відображено на рисунках 4.12 та 4.13

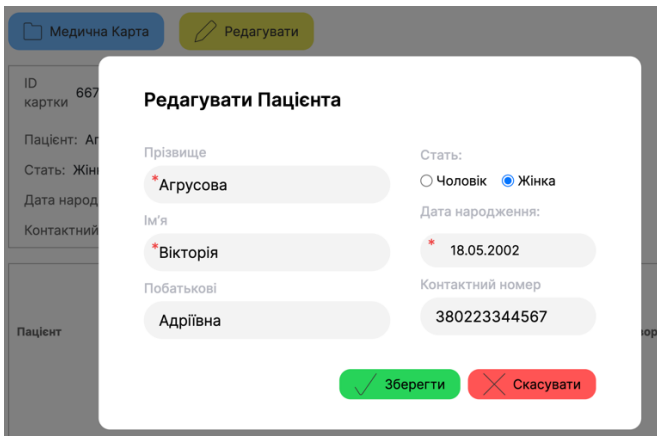


Рисунок 4.12 – Форма “Редагувати Пацієнта”

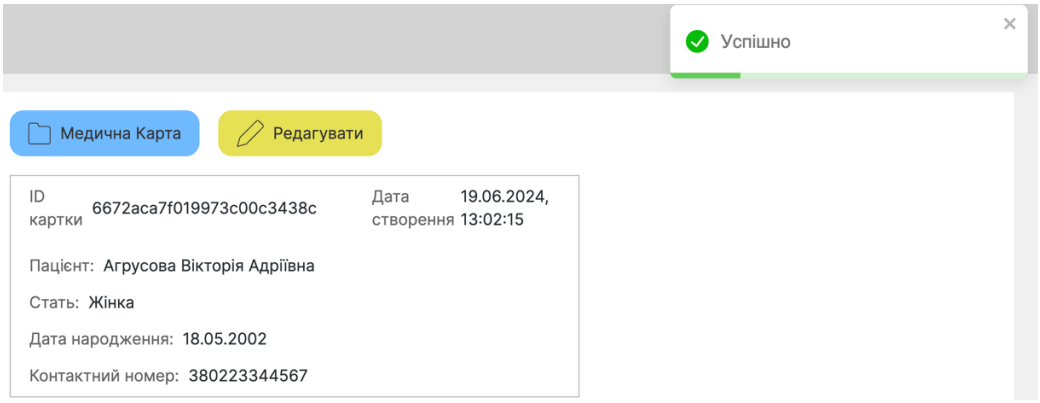


Рисунок 4.13 – Успішна зміна даних пацієнта

Далі перевіримо коректність роботи форми “Пошук пацієнта за ПІБ або ID” та наявних фільтрів (“Вибрати..”, “Дата народження: від – до”, “Дата створення: від – до”). Результати подано на рисунках 4.14 – 4.17.

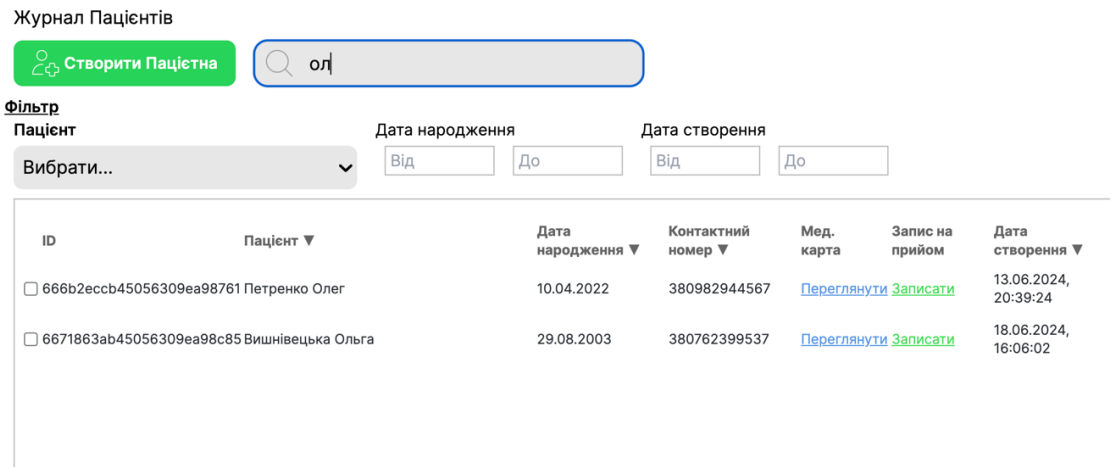


Рисунок 4.14 –Пошук пацієнта

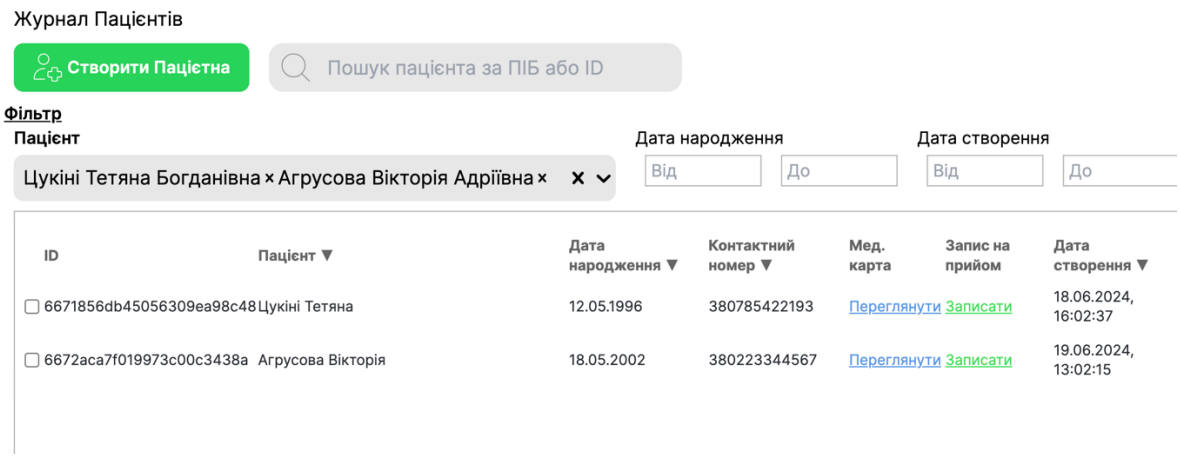


Рисунок 4.15 – Робота фільтра “Вибрати..”

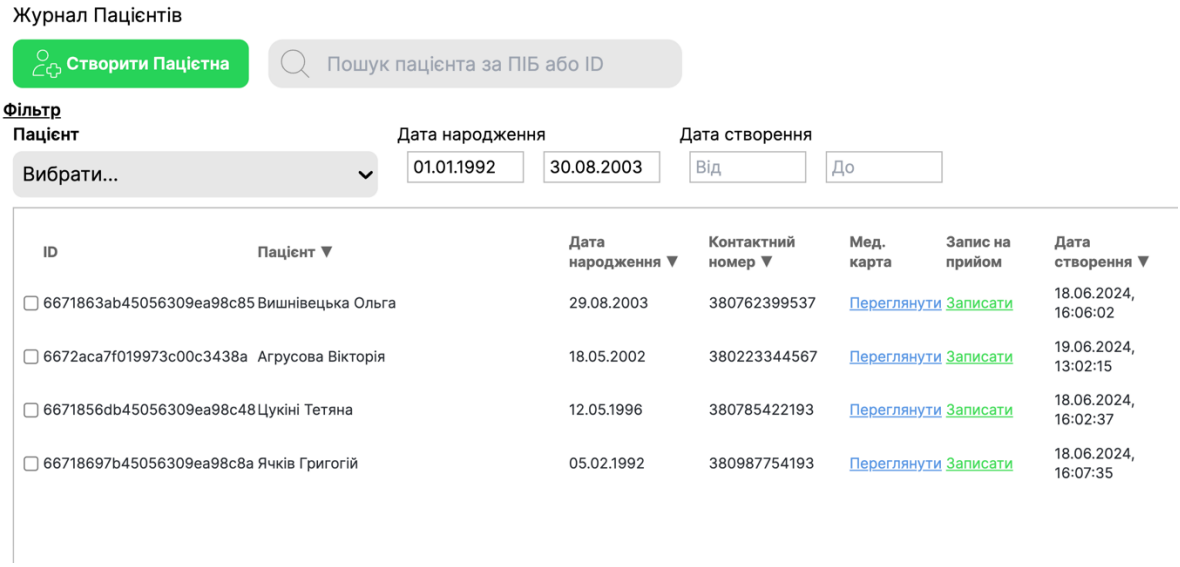


Рисунок 4.16 – Робота фільтра “ Дата народження: від – до ”

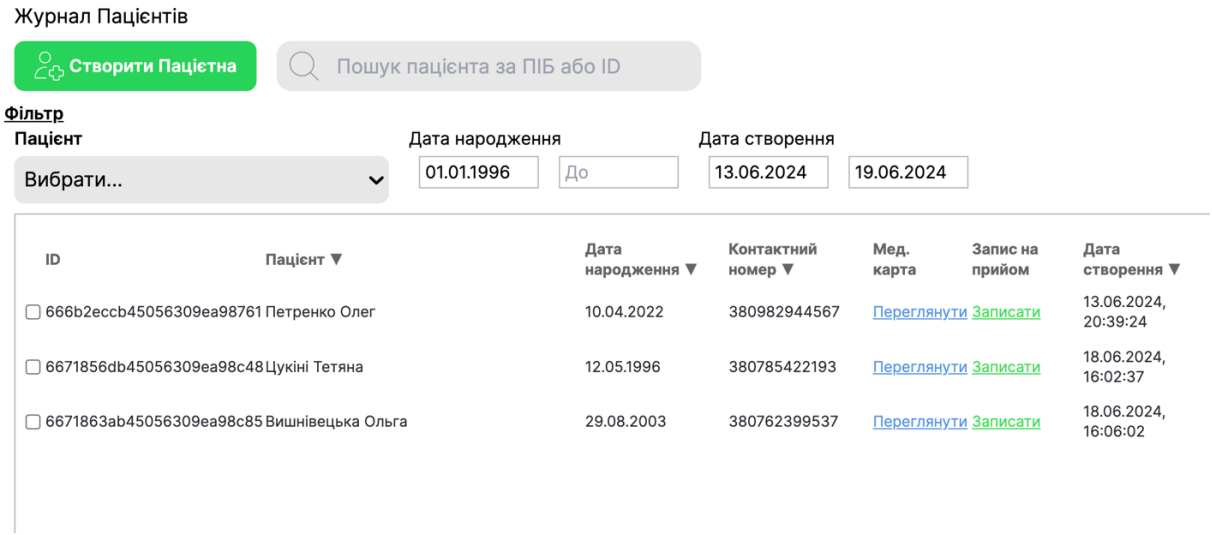


Рисунок 4.17 – Робота фільтра “Дата створення: від – до”

Тепер запишемо пацієнта на прийом до лікаря для цього відкриємо форму запису, натиснувши на кнопку “Записати” та оберемо лікаря, назаначимо дату та час (див.рисунок 4.18)

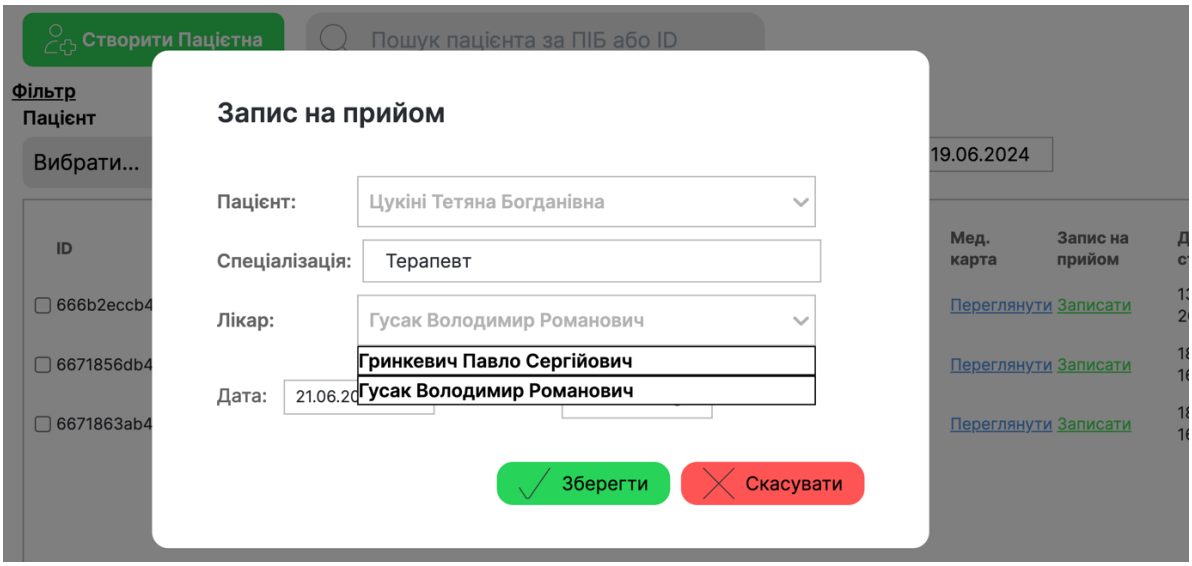


Рисунок 4.18 – Робота фільтра “Дата народження: від – до”

Перейдемо на сторінку “Записи на прийом” де можна побачити щойно створений запис на прийом (див.рисунок 4.19)

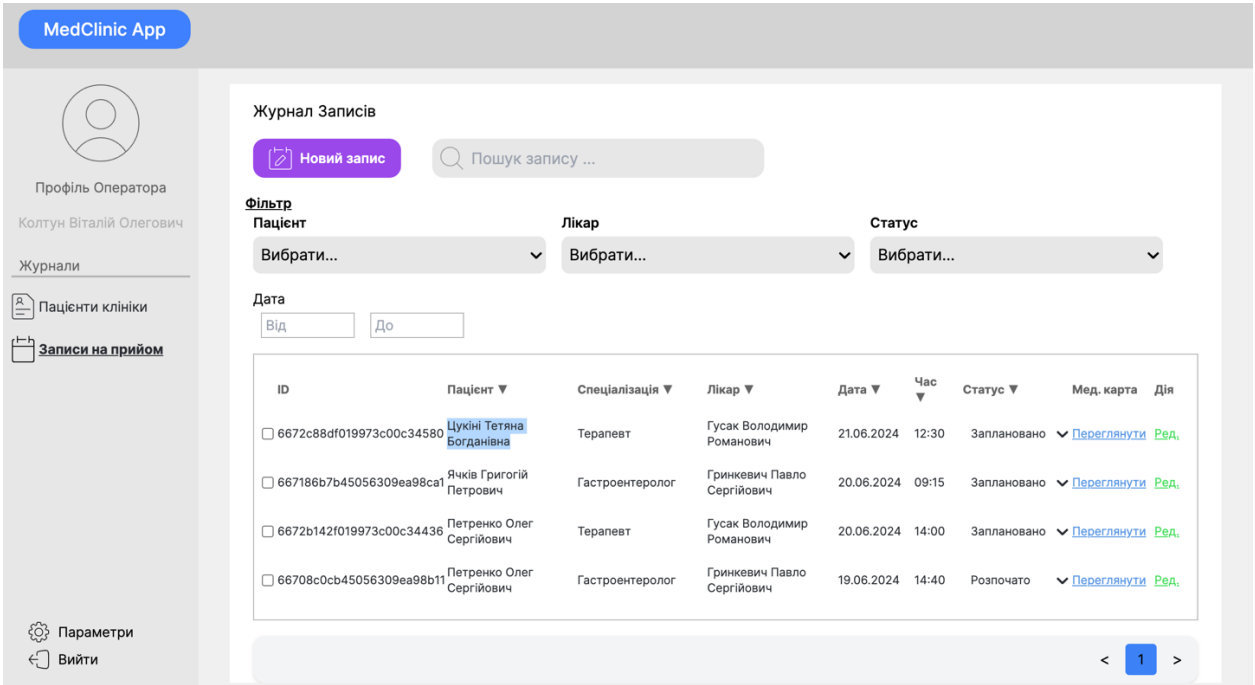


Рисунок 4.19 – Сторінка “Записи на прийом”

Відмінністю між сторінкою “Записи на прийом” для оператора та лікаря, полягає у відоброжані елементтів: кнопка “Новий запис” та опція фільтрування лікарів. Адже, згідно вимог, лікар не може записувати пацієнтів сам до себе та може переглядати записи пацієнів лише до себе (див.рисунок 4.20).

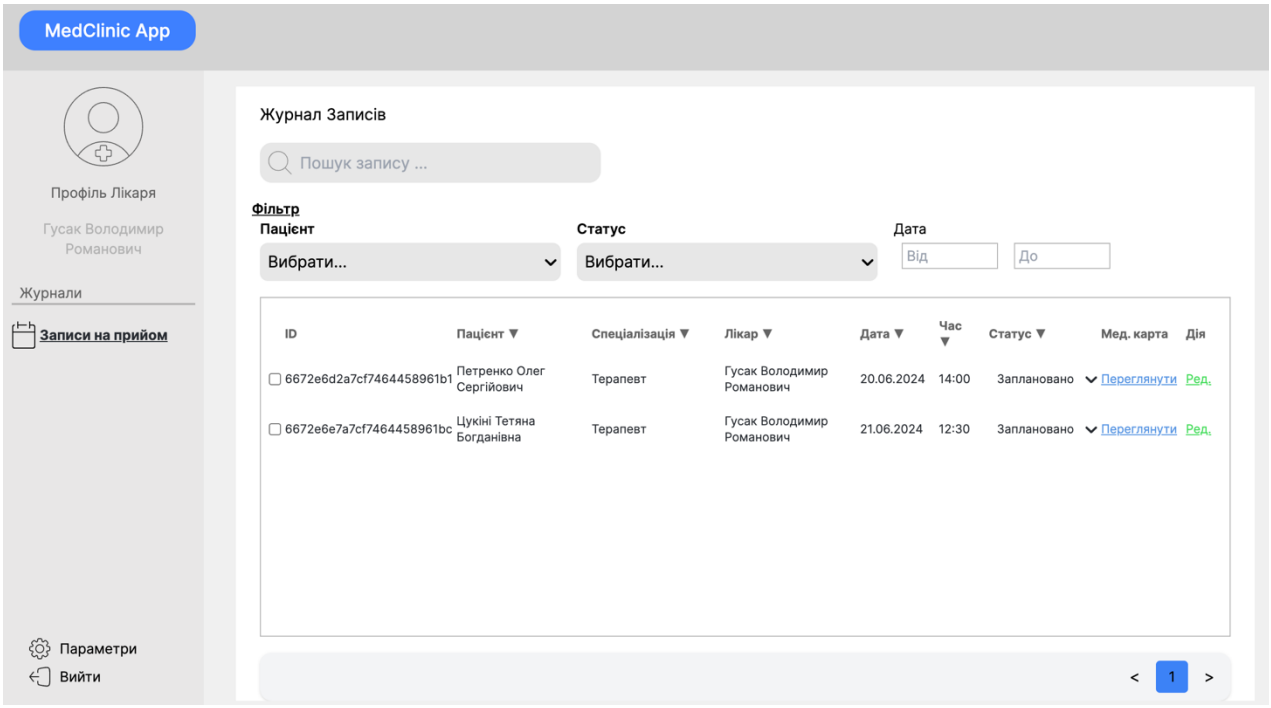


Рисунок 4.20 – Сторінка “Записи на прийом” зі сторони лікаря

Далі відкриємо мед.карту, де можемо побачити, що добавилась кнопка ”Створити запис” та зникла кнопка для редагування профілю пацієнта (див.рисунок 4.21)

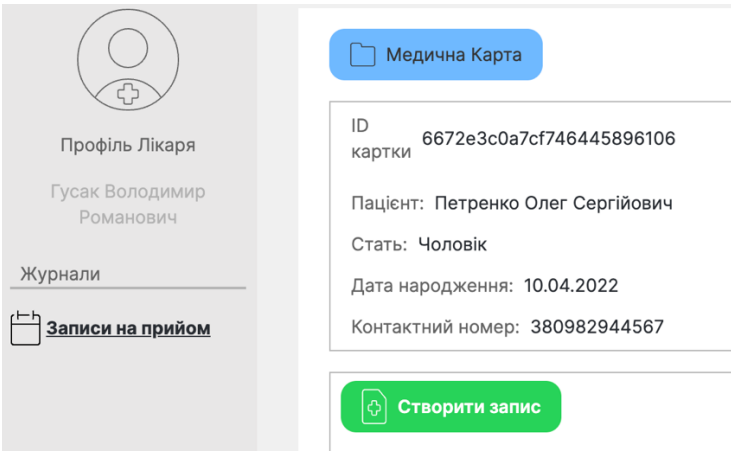


Рисунок 4.21 – Сторінка мед.карти пацієнта зі сторони лікаря

Відкриємо форму для створення нового запису, запишемо у неї дані, та збережемо результат (див.рисунок 4.22 – 4.24)

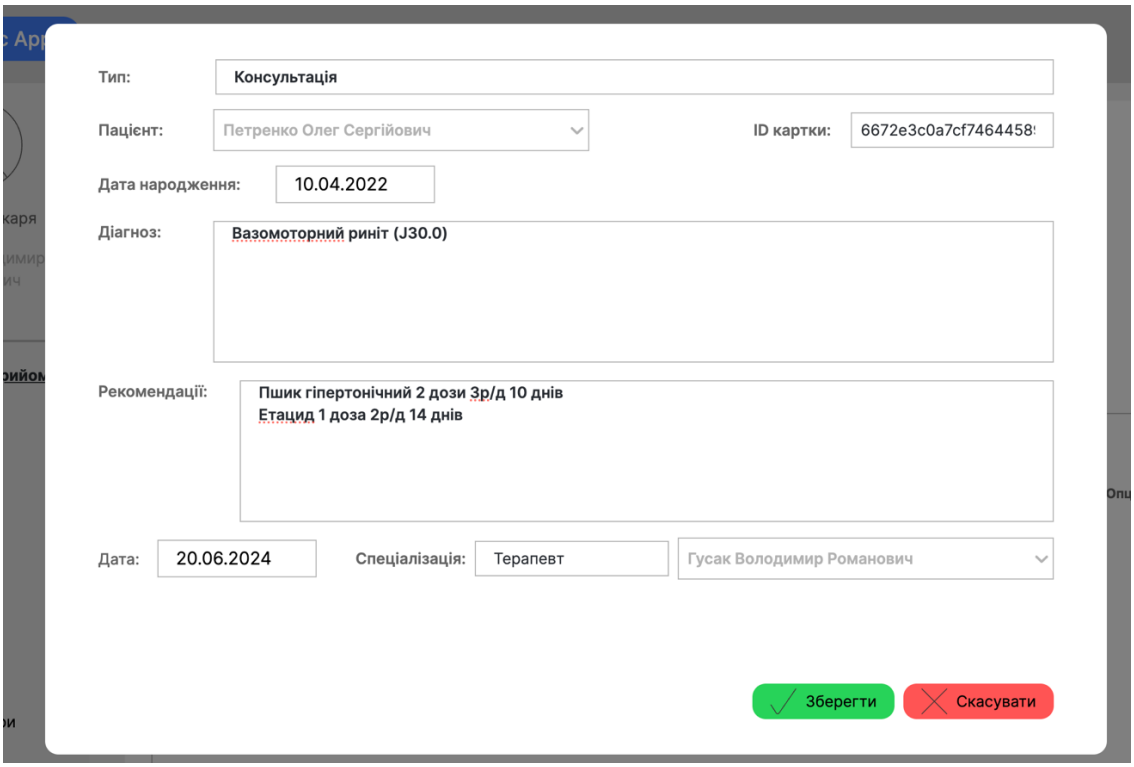


Рисунок 4.22 – Створення медичного запису

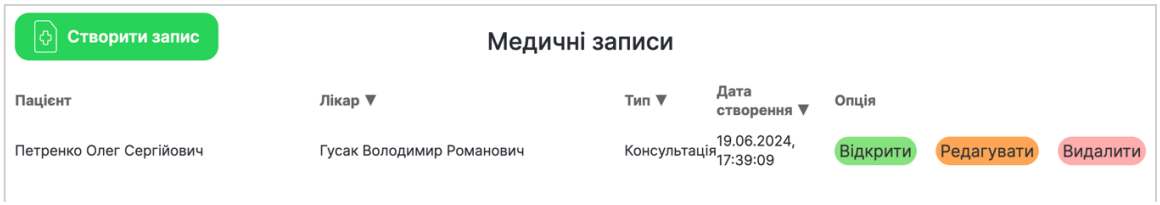


Рисунок 4.23 – Результат створення медичного запису для лікаря

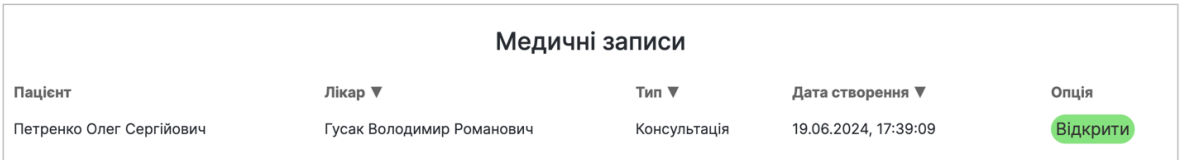


Рисунок 4.24 – Результат створення медичного запису для оператора

Далі відредагуємо медичний запис, після цього відкриємо та видалимо. Результати подано на рисунках 4.25 – 4.27.

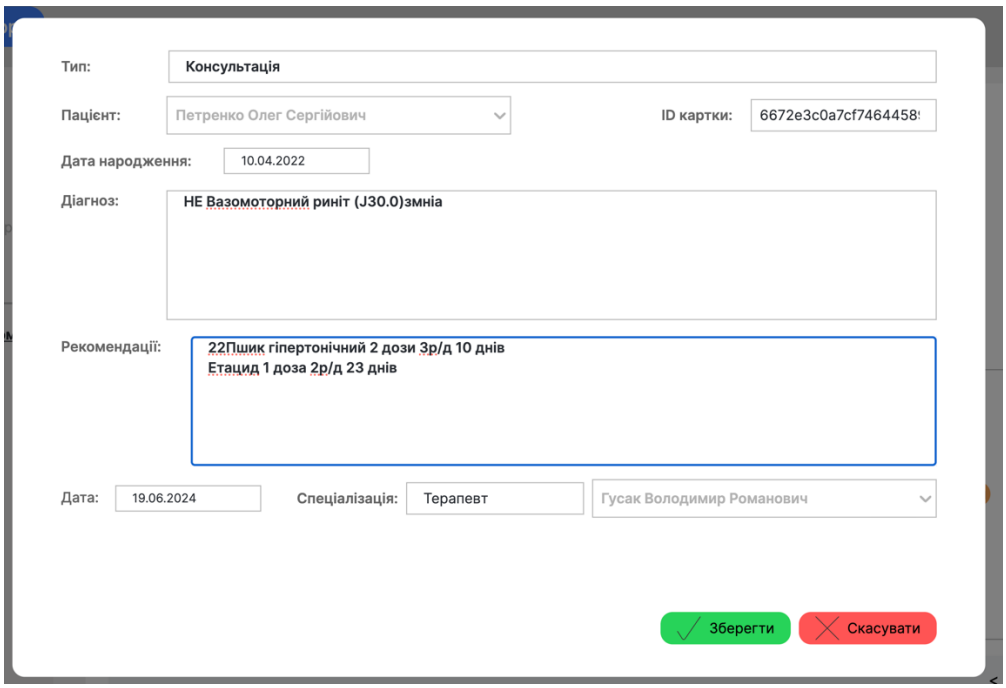


Рисунок 4.25 – Редагування медичного запису

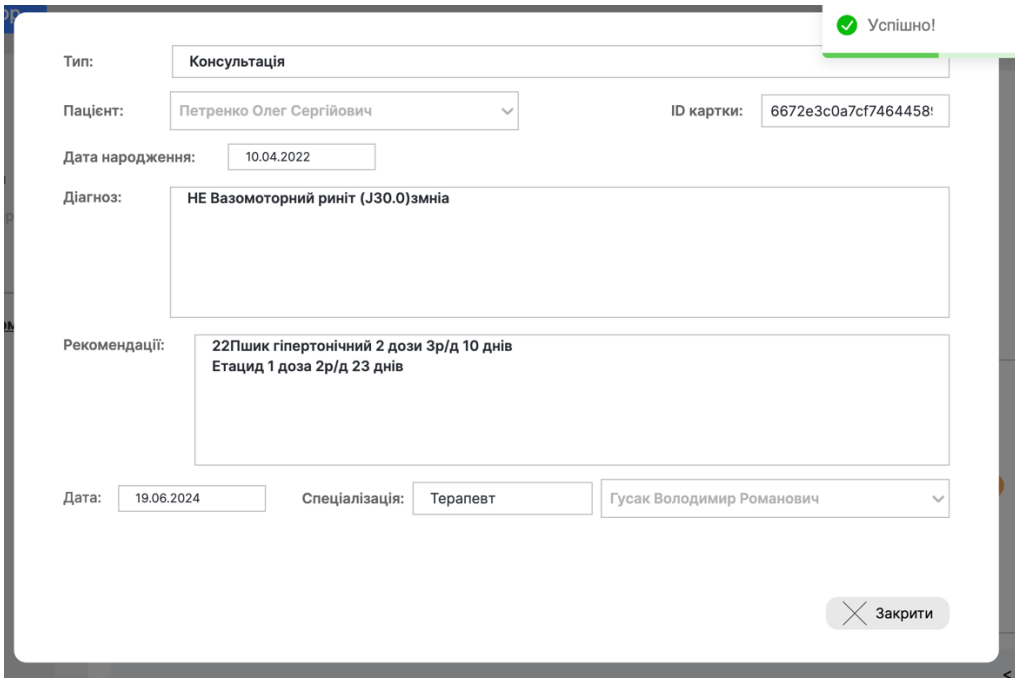


Рисунок 4.26 – Резульат редагування медичного запису

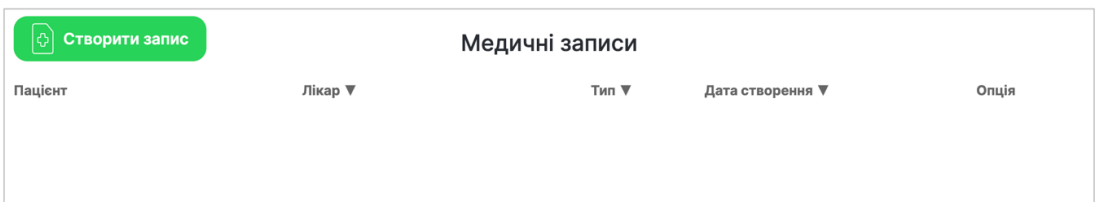


Рисунок 4.27 – Результат видалення медичного запису

Додаткові зручні функції для користувачів – “Зміна статусу прийому” та “Перенесення прийому” (див.рисунки 4.28 та 4.29)

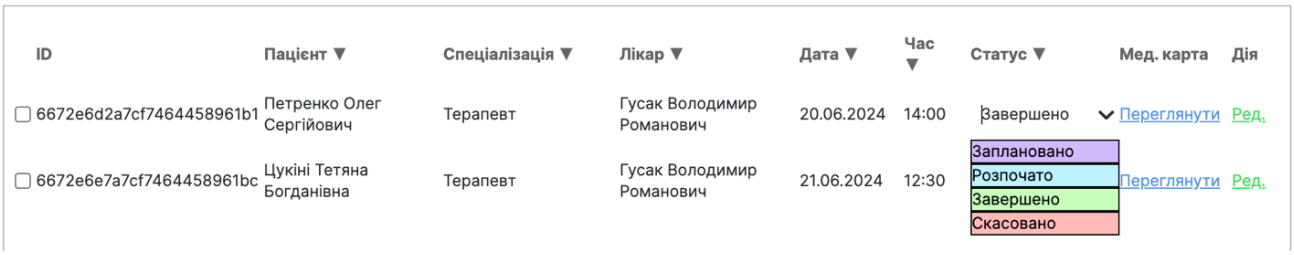


Рисунок 4.28 – Зміна статусу прийому до лікаря

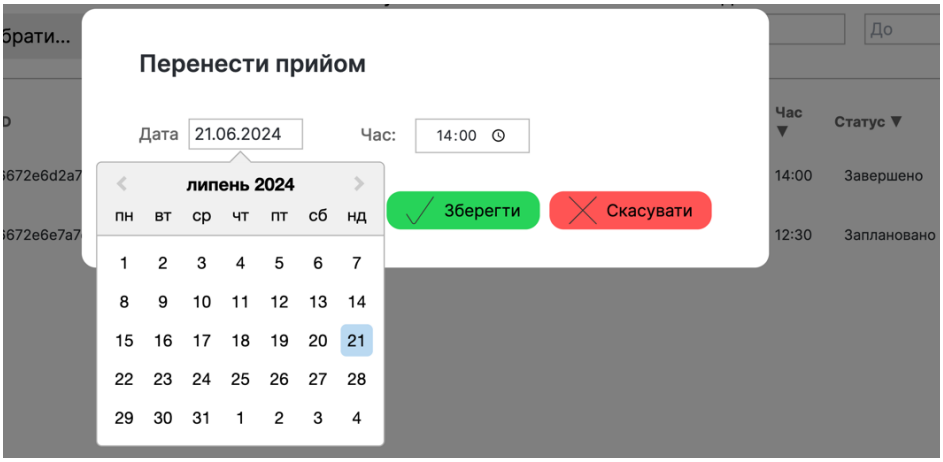


Рисунок 4.29 – Перенесення прийому до лікаря

Також кожен профіль має змогу змінювати свої особисті дані (див.рисунок 4.30)

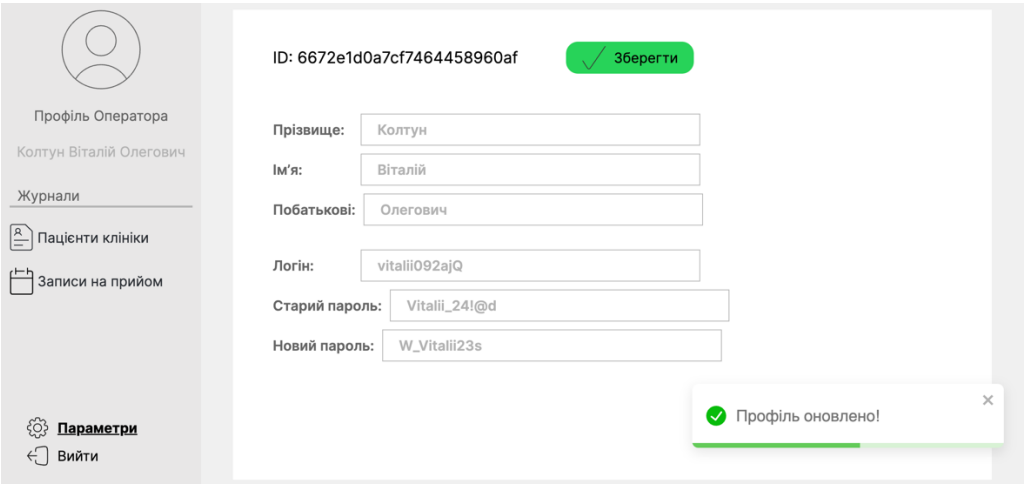


Рисунок 4.30 – Зміна паролю користувача

5 БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ, ОСНОВИ ОХОРОНИ ПРАЦІ

5.1 Ергономічні проблеми безпеки життєдіяльності

Ергономіка відіграє вирішальну роль у забезпеченні безпеки життєдіяльності в медичних клініках, зокрема при взаємодії персоналу з комп'ютерними системами, адже це є ключовим аспектом, що впливає на ефективність роботи та здоров'я працівників. Ергономіка охоплює широкий спектр питань, спрямованих на оптимізацію взаємодії між людиною та обладнанням, створюючи сприятливі умови для праці, знижуючи її важкість та напруженість, а також досягаючи високої ефективності робочого процесу.

Ергономічні вимоги до організації робочих місць враховують антропометричні, фізіологічні та психофізіологічні показники, що дозволяють створити комфортні та безпечні умови праці. Виробниче обладнання має забезпечувати оптимальний розподіл функцій між людиною і машиною, відповідати розмірам та можливостям людини, щоб знизити навантаження та запобігти втомі. Наприклад, конструкція робочого місця повинна враховувати оптимальну висоту столу та стільця, положення монітора та клавіатури, а також налаштування освітлення, щоб знизити напруження очей [24].

Розглянемо, кожен із них детальніше:

– Антропометричні показники – включають відповідність знаряддя праці розмірам, формі та масі тіла людини, а також силі та напрямку її рухів. Врахування цих показників дозволяє запобігти фізичним травмам та патологіям, пов’язаним з неправильною робочою позою або надмірними навантаженнями на певні частини тіла. Наприклад, робочий стіл адміністратора повинен бути таким, щоб працівник міг зручно розміщувати руки на клавіатурі, не напружуючи плечі і спину. Висота стільця має регулюватися, щоб ноги могли стояти на підлозі або на підставці для ніг, зменшуючи напруження в спині та ногах. Для лікарів, які проводять багато часу стоячи або сидячи під час обстежень пацієнтів, важливо забезпечити ергономічні стільці та належне освітлення. Крім того, конструкція обладнання має забезпечити

такі фізичні навантаження на працівника, за яких енергозатрати людського організму протягом зміни не перевищували б 1046,7 кДж/год.

– Фізіологічні та психофізіологічні показники – встановлюють відповідність робочих процесів швидкісним, енергетичним, зоровим та іншим фізіологічним можливостям людини. Це дозволяє оптимізувати робочий процес, знизити рівень втоми та підвищити загальну продуктивність праці. Наприклад, правильне налаштування монітора комп’ютера за висотою та кутом нахилу знижує напруження очей і запобігає виникненню головного болю та дискомфорту в шиї. Також важливо забезпечити належну вентиляцію та освітлення робочого місця, щоб зменшити втомлюваність та підтримувати високу концентрацію уваги. Для лікарів належна вентиляція і освітлення в кабінетах обстежень, також мають вирішальне значення для їхньої продуктивності [24].

Особлива увага приділяється організації робочих місць користувачів комп'ютерної техніки, до яких належать як лікарі, так і оператори. Конструкція такого робочого місця має забезпечувати підтримку оптимальної робочої пози з ергономічними характеристиками, такими як регулювання висоти сидіння, нахилу сидіння і спинки, відстані спинки до переднього краю сидіння, висоти підлокітників. Це дозволяє знизити навантаження на опорно-руховий апарат та зменшити ризик розвитку захворювань, пов'язаних з тривалою сидячою роботою [24].

Підйомно-поворотні стільці з можливістю регулювання є важливим елементом ергономічного робочого місця. Вони дозволяють налаштувати положення стільця під індивідуальні потреби кожного працівника, забезпечуючи комфорт та знижуючи ризик розвитку болю в спині та шиї. Крім того, робоче місце повинно бути обладнане підставкою для ніг, яка допомагає підтримувати правильне положення ніг і запобігати надмірному напруженню м'язів [24].

Ергономічне розміщення монітора також має велике значення. Монітор повинен бути розташований на рівні очей або трохи нижче, щоб зменшити напруження на шию та очі [24].

Оптимальна відстань від монітора до очей складає приблизно 50-70 см, що дозволяє уникнути зайвого напруження зору та покращує комфорт під час роботи. Це особливо важливо для операторів клініки, які проводять більшу частину свого робочого часу перед комп'ютером [24].

Ергономічні рекомендації, щодо організації робочого простору користувачів компютера:

– Висота робочого сидіння: для жінок: 400 мм, для чоловіків: 430 мм, для чоловіків і жінок: 420 мм

– Висота робочої поверхні під час роботи "сидячи" на комп'ютері: для жінок: 630 мм, для чоловіків: 680 мм, для чоловіків і жінок: 655 мм

– Розміри робочого стола: висота: 725 мм, Ширина: 600–1400 мм, Глибина: 800–1000 мм [24].

Також робочий стіл повинен мати простір для ніг висотою не менше 600 мм, шириною не менше 500 мм, глибиною на рівні колін не менше 450 мм та глибиною на рівні витягнутої ноги не менше 650 мм [24].

5.2 Контроль за станом охорони праці

Контроль за станом охорони праці в Україні є ключовим елементом забезпечення безпечних і здорових умов праці для працівників. Цей контроль включає в себе різні рівні нагляду, починаючи від законодавчих актів і закінчуючи безпосередніми заходами на підприємствах. Основною метою є запобігання нещасним випадкам і професійним захворюванням.

Основним законодавчим актом, що регулює питання охорони праці в Україні, є Закон України «Про охорону праці» (№ 2694-XII від 14 жовтня 1992 року, з останніми змінами від 2023 року). Цей закон визначає основні принципи державної політики у сфері охорони праці, встановлює права та обов'язки

роботодавців і працівників, а також передбачає заходи щодо забезпечення безпечних умов праці [25].

Контроль за дотриманням законодавства про охорону праці здійснює наглядовий орган, а саме: Державна служба України з питань праці. Цей орган виконує функції нагляду за дотриманням вимог охорони праці, проводить планові та позапланові перевірки підприємств, видає приписи на усунення порушень, накладає штрафи, а також може зупиняти виробничі процеси у разі серйозних порушень [25].

Згідно з законодавством, роботодавці зобов'язані створити службу охорони праці або призначити відповідального за охорону праці, якщо кількість працівників на підприємстві менше 50 осіб. Вони також повинні проводити навчання, інструктажі та перевірки знань з питань охорони праці. Інструктажі поділяються на вступний, первинний на робочому місці, повторний (не рідше одного разу на рік), позаплановий та цільовий [25].

Розглянемо їх детальніше:

1. Вступний інструктаж проводиться для всіх новоприйнятих на роботу працівників, незалежно від їх попереднього стажу роботи або освіти. Цей інструктаж охоплює основні положення законодавства про охорону праці, правила внутрішнього трудового розпорядку, а також загальні вимоги безпеки на робочому місці.

2. Первинний інструктаж на робочому місці проводиться безпосередньо на робочому місці перед початком роботи. Він включає в себе ознайомлення з робочим місцем, безпосередніми обов'язками працівника, потенційними небезпеками, що можуть виникнути, а також з засобами захисту, які необхідно використовувати під час роботи.

3. Повторний інструктаж проводиться не рідше одного разу на рік і спрямований на оновлення знань працівників з питань охорони праці. Він включає перевірку знань і навичок працівників щодо безпечного виконання робіт, а також ознайомлення з новими нормативними документами і змінами у законодавстві.

4. Позаплановий інструктаж проводиться у випадках, коли відбулися зміни у технологічному процесі, обладнанні або умовах праці, які можуть вплинути на безпеку працівників. Також він може проводитися у разі порушення працівниками вимог охорони праці, що призвели або могли призвести до нещасного випадку.

5. Цільовий інструктаж проводиться при виконанні разових робіт, не пов'язаних з безпосередніми обов'язками працівника, а також при ліквідації аварій, стихійних лих та інших надзвичайних ситуацій [26].

Також, роботодавці повинні забезпечувати працівників засобами індивідуального та колективного захисту, а також проводити атестацію робочих місць за умовами праці. Атестація включає вимірювання шкідливих і небезпечних виробничих факторів, визначення відповідності умов праці вимогам охорони праці та розробку заходів з поліпшення умов праці. Цей процес допомагає виявити та оцінити ризики на робочому місці, що, у свою чергу, сприяє розробці ефективних заходів для їх усунення або мінімізації [26].

ВИСНОВОК

У процесі виконання дипломної роботи було розроблено веб-застосунок для медичної клініки, що відповідає поставленим вимогам. Здійснений аналіз існуючих рішень дозволив виявити ключові переваги та недоліки різних підходів до автоматизації медичних процесів. Це дозволило обрати оптимальну архітектуру та технології для реалізації поставлених завдань.

Розроблений веб-застосунок на основі сучасних технологій JavaScript, таких як React для клієнтської частини та Node.js для серверної частини, демонструє високу продуктивність та гнучкість. Інтеграція з базами даних забезпечує зберігання та обробку великої кількості медичної інформації, що дозволяє швидко та ефективно отримувати доступ до необхідних даних.

Особлива увага була приділена забезпеченню безпеки медичних даних. Впроваджені механізми аутентифікації та авторизації користувачів, шифрування даних, які захищають персональні дані пацієнтів від несанкціонованого доступу. Ще одним, досягненням роботи є створення зручного та інтуїтивно зрозумілого інтерфейсу користувача, який забезпечує легкість та простоту у використанні для медичних працівників.

Проведене тестування веб-застосунку показало його високу продуктивність та надійність. Результати тестів підтвердили відповідність розробленого застосунку вимогам проєкту. Застосунок здатний обробляти велику кількість запитів одночасно, забезпечуючи стабільну роботу навіть при високих навантаженнях.

Загалом, результати даної дипломної роботи підтверджують доцільність впровадження сучасних інформаційних технологій у медичну сферу. Розроблений веб-застосунок є важливим кроком на шляху до діджилізації медичних послуг, що сприяє підвищенню продуктивності, гнучкості та безпеки, що є ключовими аспектами для успішного функціонування медичних закладів у сучасних умовах.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Leelakumar R. L. APPLICATIONS AND CHALLENGES IN HEALTHCARE BIG DATA [Електронний ресурс] / Raja Lekkala Leelakumar // International Journal of Information Technology. – 2023. – Режим доступу до ресурсу: https://dx.doi.org/10.17605/OSF.IO/AT8JR
2. Richemond D, Huggins-Jordan T. D The Impact of Health Information Systems on Patient Outcomes [Електронний ресурс] / D. Richemond, T. D. Huggins-Jordan // Open Access Library Journal. – 2023. – Режим доступу до ресурсу: https://doi.org/10.4236/oalib.1110518
3. Підключені до ЦБД Медичні Інформаційні Системи [Електронний ресурс] // Національна служба здоров'я України – Режим доступу до ресурсу: https://ehealth.gov.ua/pidklyucheni-do-ehealth-mis/.
4. Про МІС «МедЕйр» [Електронний ресурс] – Режим доступу до ресурсу: https://info.e-life.com.ua/pages/viewpage.action?pageId=1245273
5. Історія сторення «Доктор Елекс» [Електронний ресурс] – Режим доступу до ресурсу: https://doctor.eleks.com/landings/istoria-stvorenia
6. Рішення для великих приватних закладів «ДокторЕлекс» [Електронний ресурс] – Режим доступу до ресурсу: https://doctor.eleks.com/landings/dlja-velikih-privatnih-zakladiv
7. Некрасов В. Helsi.me і Doc.ua: як заробляють дві найбільші в Україні медичні цифрові платформи [Електронний ресурс] / Всеволод Некрасов // Економічна Правда. – 2021. – Режим доступу до ресурсу: https://www.epravda.com.ua/publications/2021/03/2/671524/.
8. Про Helsi [Електронний ресурс] – Режим доступу до ресурсу: https://helsi.me/about
9. Медична інформаційна система Health24 [Електронний ресурс] – Режим доступу до ресурсу: https://doc.h24.ua/

10. Група українських IT-продуктових компаній [Електронний ресурс] – Режим доступу до ресурсу: https://e-tech.ua/
11. Можливості MIC Health24 [Електронний ресурс] – Режим доступу до ресурсу: https://emci.ua/products/health24/
12. Інтрумент для створення UML діаграм [Електронний ресурс] – Режим доступу до ресурсу: https://www.drawio.com
13. Gorman K. Rapid Application Development Practice in 2023: Advantages and Disadvantages [Електронний ресурс] / Karl Gorman // Chudovo. – 2023. – Режим доступу до ресурсу: https://chudovo.com/rapid-application-development-practice-advantages-and-disadvantages/.
14. Event-Driven Architecture in Node.js [Електронний ресурс]. – 2023. – Режим доступу до ресурсу: https://dev.to/learn-to-earn/event-driven-architecture-in-nodejs-1o98.
15. Express/Node introduction [Електронний ресурс] – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction
16. Schemas Mongoose [Електронний ресурс] – Режим доступу до ресурсу: https://mongoosejs.com/docs/guide.html
17. Introduction to JSON Web Tokens [Електронний ресурс] – Режим доступу до ресурсу: https://jwt.io/introduction/
18. Negi S. Understanding React Virtual DOM: How it Improves Performance [Електронний ресурс] / Suraj Negi – Режим доступу до ресурсу: https://cloudactivelabs.com/en/blog/understanding-react-virtual-dom-how-it-improves-performance.
19. Getting started with Reactjs and Nextjs [Електронний ресурс] – Режим доступу до ресурсу: https://dev.to/sarahadewale/getting-started-with-reactjs-and-nextjs-4g1g
20. Ding S. Announcing SWR 2.0 [Електронний ресурс] / S. Ding, J. Liu, T. Kobayashi // SWR. – 2022. – Режим доступу до ресурсу: https://swr.vercel.app/blog/swr-v2.

21. Introduction to Tailwind CSS [Електронний ресурс] – Режим доступу до ресурсу: https://www.geeksforgeeks.org/introduction-to-tailwind-css/
22. Modern Web App Architectures: How to Choose the Right One for Your Project [Електронний ресурс] // Digiteum. – 2023. – Режим доступу до ресурсу: https://www.digiteum.com/web-application-architecture/
23. Sypiahin O. Understanding Feature-Sliced Design: Benefits and Real Code Examples [Електронний ресурс] / Oleh Sypiahin // Hackernoon. – 2023. – Режим доступу до ресурсу: https://hackernoon.com/understanding-feature-sliced-design-benefits-and-real-code-examples
24. Безпека життєдіяльності: навч. посібник [Електронний ресурс] / Т. Є.Стиценко, Г. В. Пронюк, Н. М. Сердюк, І. І. Хондак // Харків: ХНУРЕ. – 2018. – Режим доступу до ресурсу: https://os.nure.ua/wp-content/uploads/2019/05/posibnik-bgd_2018_p.1.pdf
25. Закон України про охорону праці [Електронний ресурс] // Верховна Рада України – Режим доступу до ресурсу: https://ips.ligazakon.net/document/T269400?an=1.
26. Про затвердження Типового положення про порядок проведення навчання і перевірки знань з питань охорони праці та Переліку робіт з підвищеною небезпекою [Електронний ресурс] // Наказ Державного комітету України з нагляду за охороною праці – Режим доступу до ресурсу: https://ips.ligazakon.net/document/RE10511?an=481

ДОДАТКИ

Додаток А – Диск із кваліфікаційною роботою бакалавра