

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «бакалавр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)

на тему:

«Система підтримки та планування роботи пансіонату»

Виконав студент групи ПЗ-20-3
БАШМАНІВСЬКИЙ Максим Олексійович

Керівник роботи:
ВЛАСЕНКО Олег Васильович

Рецензент:
САВІЦЬКИЙ Роман Святославович

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

«ЗАТВЕРДЖУЮ»

Зав. кафедри інженерії
програмного забезпечення

Тетяна ВАКАЛЮК

«26» лютого 2024 р.

ЗАВДАННЯ
на кваліфікаційну роботу

Здобувач вищої освіти: **БАШМАНІВСЬКИЙ Максим Олексійович**

Керівник роботи: **ВЛАСЕНКО Олег Васильович**

Тема роботи: **«Система підтримки та планування роботи пансіонату»**,
затверджена Наказом закладу вищої освіти від **«23» лютого 2024 р., №74/с**

Вихідні дані для роботи: **інтеграція сучасних веб-технологій у сферу
надання послуг поновлення фізичних ресурсів рекреаційним закладом з
метою оптимізації надання послуг пансіонатом.**

Консультанти з бакалаврської кваліфікаційної роботи із зазначенням розділів,
що їх стосуються:

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Власенко О.В.	27.02.2024р.	27.02.2024р.
2	Власенко О.В.	27.02.2024р.	27.02.2024р.
3	Власенко О.В.	27.02.2024р.	27.02.2024р.

РЕФЕРАТ

Випускна кваліфікаційна робота бакалавра складається з програмного комплексу вебдодатку системи підтримки та планування роботи пансіонату і пояснювальної записки. Пояснювальна записка до випускної роботи містить текстову частину, викладену на 86 сторінках друкованого тексту, має 19 сторінок додатків. Список використаних джерел містить 30 найменувань і займає 3 сторінки. В роботі наведено 60 рисунків та 16 таблиць. Загальний обсяг роботи – 105 сторінки.

Метою роботи є розробка вебдодатку системи підтримки та планування роботи пансіонату засобами HTML, CSS, JavaScript, TypeScript, а також розроблення системи адміністрування за допомогою мови програмування TypeScript із використанням бази даних PostgreSQL.

У роботі було визначено вимоги користувача та обґрунтовано створення концептуальної моделі. Також було проаналізовано аналоги програмних продуктів та обрано засоби реалізації, проведено розробку структури бази даних, а також було розроблено алгоритм роботи, обрано архітектуру системи. Також описано програмну реалізацію вебдодатку.

КЛЮЧОВІ СЛОВА: ВЕБДОДАТОК, БАЗА ДАНИХ, ПАНСІОНАТ, ПРОЦЕДУРИ.

					ІПЗ.КР.Б-121-24-ПЗ			
Змн.	Арк.	№ докум.	Підпис	Дата	Система підтримки та планування роботи пансіонату	Літ.	Арк.	Аркушів
Розроб.		Башиманівський М.О.					3	
Керівник		Власенко О.В.				Житомирська політехніка, група ІПЗ-20-3		
Керівник								
Н. контр.		Тичина О.П.						
Зав. каф.		Вакалюк Т.А.						

ABSTRACT

The bachelor's qualifying thesis consists of a software complex of a web application for the support and planning system of a boarding house and an explanatory note. The explanatory note to the graduation work contains a textual part, laid out on 86 pages of printed text, and includes 19 pages of appendices. The list of references includes 30 titles and occupies 3 pages. The work includes 60 figures and 17 tables. The total volume of the work is 105 pages.

The aim of the work is to develop a web application for the support and planning system of a boarding house using HTML, CSS, JavaScript, and TypeScript, as well as to develop an administration system using the TypeScript programming language with the PostgreSQL database.

The work defines user requirements and justifies the creation of a conceptual model. It also analyzes analogs of software products and selects means of implementation, develops the structure of the database, and designs the algorithm of the system. The architectural design of the system and the software implementation of the web application are also described.

KEYWORDS: WEB APPLICATION, DATABASE, BOARDING HOUSE, PROCEDURES.

ЗМІСТ

РЕФЕРАТ	3
ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ.....	6
ВСТУП	7
РОЗДІЛ 1. АНАЛІЗ ВИМОГ КОРИСТУВАЧА ТА КОНЦЕПТУАЛЬНЕ МОДЕЛЮВАННЯ	9
1.1 Постановка задачі	9
1.2 Аналіз аналогів програмного продукту	10
1.3 Вибір архітектури веб-орієнтованої системи підтримки та планування роботи пансіонату	19
1.4 Обґрунтування вибору інструментальних засобів та вимоги до апаратного забезпечення	21
Висновки до першого розділу.....	25
РОЗДІЛ 2. ПРОЕКТУВАННЯ ВЕБ-ОРІЄНТОВАНОЇ СИСТЕМИ ПІДТРИМКИ ТА ПЛАНУВАННЯ РОБОТИ ПАНСІОНАТУ	26
2.1 Визначення варіантів використання та об'єктно-орієнтованої структури системи.....	26
2.2 Розробка бази даних системи.....	35
2.3 Проектування та реалізація алгоритмів роботи системи	41
Висновки до другого розділу	58
РОЗДІЛ 3. ІНТЕРФЕЙС ТА ПОРЯДОК РОБОТИ СИСТЕМИ ПІДТРИМКИ ТА ПЛАНУВАННЯ РОБОТИ ПАНСІОНАТУ	60
3.1 Порядок встановлення та налаштування параметрів системи	60
3.2 Структура інтерфейсу вебдодатку та порядок роботи.....	64
3.3 Тестування роботи програмного комплексу	81
Висновки до третього розділу.....	82
ВИСНОВКИ.....	84
СПИСОК ЛІТЕРАТУРИ.....	85
ДОДАТКИ.....	87

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API – прикладний програмний інтерфейс

HTTPS — протокол передачі гіпертексту з додатковим рівнем безпеки

БД – база даних

СУБД – система управління базами даних

					ІПЗ.КР.Б-121-24-ПЗ	Арк.
						6

ВСТУП

Сучасні технології швидко трансформують сферу охорони здоров'я, роблячи її більш ефективною та доступною. Розробка вебдодатку для рекреаційного закладу стає необхідністю для поліпшення організації роботи, забезпечення швидкого доступу до інформації та полегшення взаємодії між пацієнтами та медичним персоналом.

Метою цієї роботи є створення вебдодатку, спрямованого на покращення надання медичних послуг пансіонатом та оптимізацію робочого процесу. Для досягнення цього передбачається розробка інтерактивного інструменту на основі сучасних технологій. Іншими словами, планується створення інтерфейсу для користувачів з використанням JavaScript бібліотеки - React та розробка серверної частини за допомогою NestJS – платформи для створення ефективних масштабованих програм. Взаємодія між клієнтською і серверною частинами має відбуватись за допомогою REST API - стандарту архітектури програмного забезпечення, який використовує принципи і засоби HTTP-протоколу для обміну даними між комп'ютерними системами. Для зберігання даних користувачів буде використана БД – PostgreSQL. Додаток має забезпечувати безперебійну та стабільну роботу, мати інтуїтивний інтерфейс користувача, захист від потенційних загроз безпеки, таких як атаки на безпеку даних, володіти можливістю розширення - легкістю у впровадженні нового функціоналу. Також метою є поглиблення практичних та теоретичних навичок та знань, необхідних для розробки вебдодатку.

Мета визначає, що необхідно виконати наступні завдання:

- Аналіз існуючих проблем та потреб у медичній сфері, які можна вирішити використовуючи веб-технології;
- Аналіз вебсайтів, які мають побідну тематику, визначення унікальності;
- Вибір технологій для розробки та вивчення їх можливостей;
- Визначення інформаційних потреб предметної області дослідження;
- Провести порівняльний аналіз різних архітектурних варіантів з урахуванням їх переваг і недоліків;

					ІПЗ.КР.Б-121-24-ІЗ	Арк.
						7

- Вибір оптимальної архітектури з урахуванням вимог до функціональності, масштабованості, продуктивності та доступних технологій;
- Створення опису методів для взаємодії з вебдодатком;
- Розробка структури бази даних для зберігання інформації про сервіс та його користувачів;
- Розробка моделі програмного комплексу на логічному рівні;
- Визначення фізичної моделі вебдодатку;
- Написання серверної частини вебдодатку;
- Написання клієнтської частини вебдодатку;
- Тестування роботи програмного комплексу.

Об'єкт дослідження - трансформація сфери охорони здоров'я через використання сучасних веб-технологій.

Предмет дослідження - інтеграція сучасних веб-технологій у сферу надання послуг поновлення фізичних ресурсів рекреаційним закладом з метою оптимізації надання послуг пансіонатом.

В роботі було використано методи проектування архітектури програмного продукту, аналіз існуючих технологій та методів розробки програмного забезпечення, реалізація та програмування функціональності програмного продукту.

За темою випускної кваліфікаційної роботи бакалавра було підготовлено тези, які будуть опубліковані [1]. Хоча тези ще не опубліковані, в програмі проведення засідань зазначена моя тема, що підтверджує участь у конференції та актуальність дослідження.

РОЗДІЛ 1. АНАЛІЗ ВИМОГ КОРИСТУВАЧА ТА КОНЦЕПТУАЛЬНЕ МОДЕЛЮВАННЯ

1.1 Постановка задачі

У зв'язку з швидким розвитком технологій та необхідністю покращення управління та надання послуг пансіонату, створення інноваційної системи для підтримки та планування роботи цього закладу є актуальним завданням. Висока конкуренція, зростання обсягів інформації, а також зміни вимог до якості та доступності рекреаційних послуг вимагають від пансіонатів ефективних рішень для управління та надання послуг.

Створення інноваційної системи для пансіонату дозволить не лише оптимізувати процеси управління та обслуговування, але й покращити якість надання рекреаційних послуг для відвідувачів. Це можливо завдяки впровадженню новітніх інформаційних технологій, які сприяють збору, обробці та аналізу даних, а також полегшують взаємодію між персоналом пансіонату та відвідувачами.

Одним із основних завдань інноваційної системи для пансіонату є забезпечення зручного та ефективного доступу до інформації про послуги, автоматизація процесів бронювання та організації дозвілля, а також забезпечення безпеки та конфіденційності особистих даних відвідувачів закладу. Така система повинна бути гнучкою та масштабованою, щоб враховувати потреби різних пансіонатів, від малих осель для відпочинку до великих готелів та курортів.

Перелік задач, що необхідно реалізувати у вигляді програмного комплексу:

- Реєстрація користувачів системи (лікарі, пацієнти), імплементація аутентифікації користувачів.
- Ведення медичних звітів щодо отриманих процедур, створення модулю для зберігання та оновлення медичної інформації кожного пацієнта;

- Можливість планування розпорядку дня пацієнта, розробка інтерфейсу для вибору графіку відвідування процедур, імплементація можливості коригування розпорядку дня;
- Створення системи обміну повідомленнями з використанням приватних чатів, забезпечення конфіденційності та безпеки даних;
- Реалізація інтерфейсу для оцінки задоволення пацієнтами після кожної процедури, зберігання даних про оцінки для подальшого аналізу;
- Використання інтелектуальних асистентів, таких як ChatGPT: Інтеграція ChatGPT дозволяє пацієнтам та лікарям отримувати швидкі та точні відповіді на їх запитання, пов'язані з медичними процедурами;
- Створення інтерфейсу з інтуїтивно зрозумілим дизайном, який спростить процес перегляду та редагування медичних записів;
- Можливість керування ролями користувачів та їхніми правами доступу, реалізація адміністративного інтерфейсу для управління ролями користувачів та налаштуванням їхніх прав доступу до функціоналу системи;
- Система повідомлень для користувачів щодо запланованих процедур;
- Адаптивність додатку для мобільних пристроїв: Інтеграція адаптивного дизайну забезпечує коректну роботу додатку на різних типах мобільних пристроїв;

Наведений перелік завдань для реалізації інноваційної системи для пансіонату охоплює широкий спектр функціональних можливостей, необхідних для забезпечення ефективного управління та надання якісних послуг відвідувачам. Реалізація цих завдань у вигляді програмного комплексу дозволить оптимізувати роботу пансіонату та покращити задоволеність клієнтів його послугами.

1.2 Аналіз аналогів програмного продукту

Вступ до аналізу аналогів програмного продукту передбачає дослідження існуючих програмних рішень на ринку з метою визначення їхніх

					ІПЗ.КР.Б-121-24-ІПЗ	Арк.
						10

функціональних можливостей та особливостей. Такий аналіз допомагає з'ясувати, які функції вже реалізовані, виявити можливі недоліки та переваги, а також знайти простір для нововведень у власній системі.

На додаток до розглянутих аналогічних програмних рішень, важливо визначити їхню поширеність на ринку та рівень задоволеності користувачів. Аналіз включає огляд функціональних можливостей, інтерфейсу користувача, безпеки даних, швидкодії та інших параметрів, що визначають якість програмного продукту. Знання сильних та слабких сторін конкурентів дозволить краще розуміти потреби ринку та вибрати стратегію подальшого розвитку продукту що створюється.

Helsi — це сучасна, зручна та надійна електронна медична система, створена для пацієнтів, лікарів, державних і приватних медичних закладів[6].

Можливості Helsi:

1. Автоматизація реєстратури та роботи лікаря;
2. Управління розкладом лікаря;
3. Ведення електронних медичних карток (ЕМК);
4. Формування звітів та статистики;
5. Підтримка формату DICOM та стандарту HL7;
6. Конструктор бланків і форм;
7. Інтуїтивно зрозумілий веб-інтерфейс;
8. Гнучке налаштування прав доступу;
9. Фіксація дій користувачів;
10. Доступ пацієнтів до своєї електронної медичної картки (ЕМК);
11. Забезпечення надійного шифрування та безпека даних;
12. Портал з інформацією про лікарів, годинами їх прийому та адресами лікарень;
13. Підтримка для лікарів та пацієнтів від контакт-центру.

Дизайн головної сторінки виконаний у сучасному та мінімалістичному стилі з використанням білого, сірого та зеленого кольорів. Шрифти чіткі та легко читаються, а зображення та іконки якісні та інформативні (рис. 1.1).

відгуки. Сторінка надає можливість вибору фільтрів для пошуку лікаря, перегляду інформації про нього та запису на прийом.

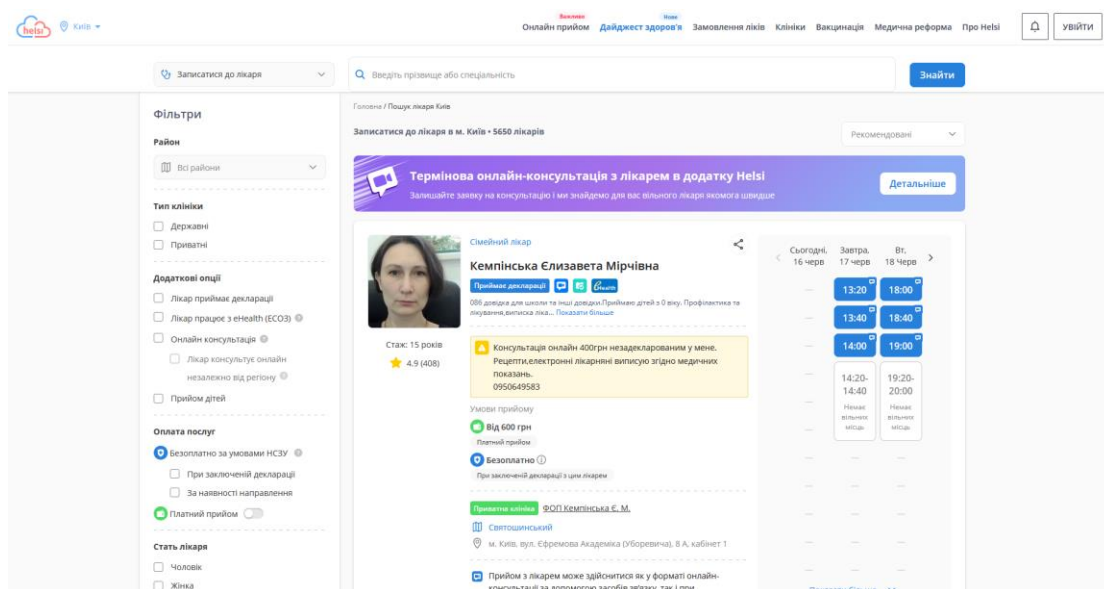


Рисунок 1.2 – Сторінка запису на прийом helsinki.me

Сторінка облікового запису містить інформацію про записи на прийоми до лікарів, анкету донорства крові, направлення до лікарів, рецепти, записи про імунізації, медичні документи, історія онлайн замовлень в аптеках, страхові поліси, медкарти близьких людей, сповіщення та налаштування профілю (рис. 1.3). Загалом, сторінка облікового запису на Helsi.me забезпечує зручний та централізований доступ до важливої медичної інформації та послуг для користувачів.

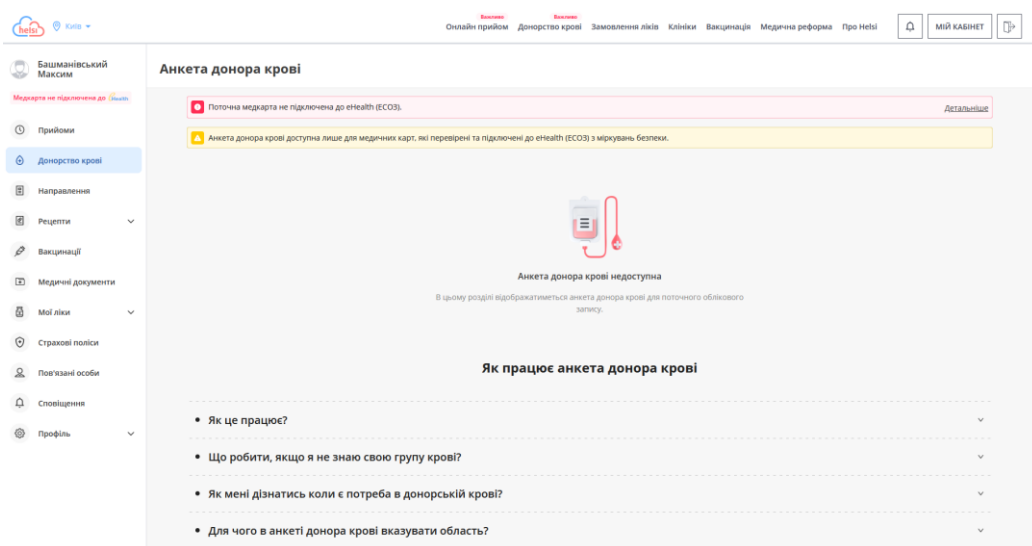


Рисунок 1.3 – Сторінка інформації про обліковий запис

Medics - це комплексна система управління медичним закладом, яка допомагає автоматизувати та оптимізувати всі аспекти роботи (рис. 1.4).

Можливості Medics[8]:

- Управління пацієнтами;
- Ведення медичної документації;
- Статистика та аналітика;
- Управління персоналом;
- Складський облік.

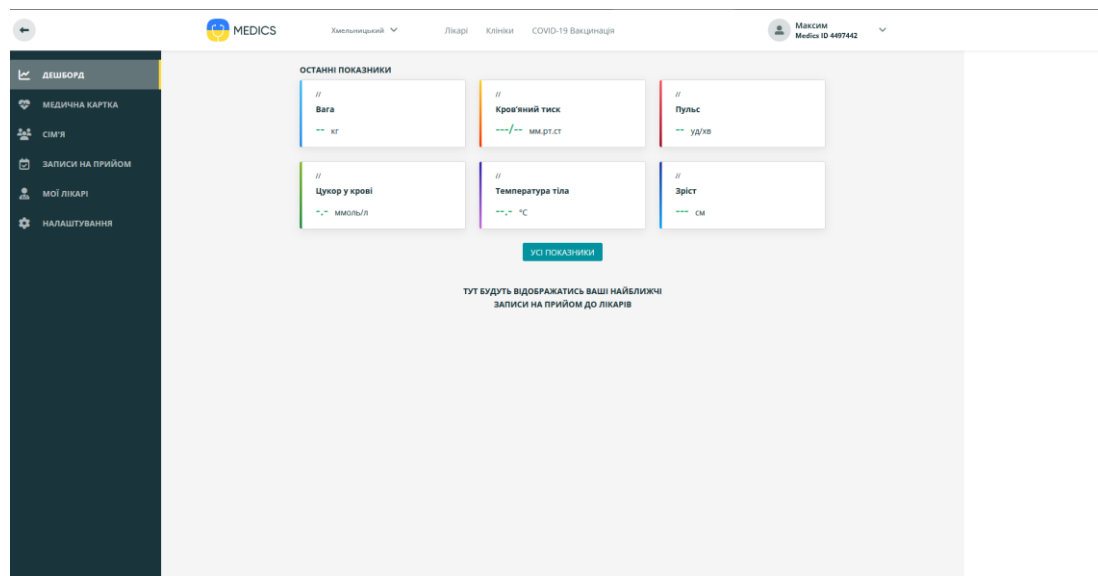


Рисунок 1.4 – головна сторінка вебсайту medics.ua

Головна сторінка вебсайту має вигляд кабінету користувача з його інформацією. Наявна медична картка, лікарі до яких був записаний користувач, активні записи на прийом, керування підопічними членами сім'ї.

Сторінка запису на прийом має фільтри пошуку лікарів, картки лікарів з загальною інформацією та час на який можна записатись до лікаря(рис. 1.5). Загалом додаток має мінімалістичний дизайн, що відображає необхідний функціонал.

аналізів. Це дозволяє лікарям швидко отримувати доступ до повної історії хвороби пацієнта, що сприяє ефективному лікуванню та моніторингу стану здоров'я.

Оберіть дату < 28/05/2024 >

Спеціальність < >

Кріс 1
Вибір запису

Кріс 2
Вибір відділення

Кріс 3
Вибір часу

Пошук

☒ Показати зайнятий час

Лікар-гастроентеролог

Матвеев Марина
Піпівник

08:00

08:20

08:40

09:00

09:20

09:40

10:00

10:20

10:40

11:00

11:20

11:40

12:00

12:20

12:40

13:00

13:20

13:40

14:00

Лікар-кардіолог

Нівець Ольга
Валентинівна

08:00

08:20

08:40

09:00

09:20

10:20

10:40

11:00

11:20

11:40

12:00

12:20

12:40

13:00

13:20

13:40

14:00

Лікар-онколог

Бакорук Ірина
Григорівна, Кабінет
№ 512а Онколог

08:00

08:20

08:40

10:00

10:20

10:40

11:00

11:20

11:40

12:00

12:20

12:40

13:00

13:20

13:40

14:00

Лікар-ревматолог

Тимощук Валентина
Сергіївна

08:00

08:20

08:40

09:00

09:20

09:40

10:00

10:20

10:40

11:00

11:20

11:40

12:00

12:20

12:40

13:00

13:20

13:40

14:00

Лікар-інфекціоніст

Шенгун Людмила
Миколаївна, Кабінет
№ 221

08:00

08:15

08:30

08:45

09:00

09:15

09:30

09:45

10:00

10:15

10:30

10:45

11:00

11:15

11:30

11:45

12:00

12:15

12:30

12:45

13:00

13:15

13:30

13:45

Рисунок 1.6 — Планування запису до лікаря системи ELEKS Portal

Програмні продукти, Helsi та Medics, представляють собою комплексні системи для управління медичними закладами, призначені для покращення якості медичного обслуговування та оптимізації робочих процесів. Обидва вони мають ряд спільних функціональних можливостей, таких як управління пацієнтами, ведення медичної документації, статистика та аналітика, управління персоналом та складський облік.

Helsi вирізняється низкою додаткових функцій, таких як облік вакцинації, страхових полісів, анкети донора крові. Наявна можливість запису до лікаря на онлайн прийом. Також варто зазначити швидкодію Helsi в порівнянні з Medics. Користувач системи не помітить процес завантаження вмісту сторінки вебдодатку Helsi.

Під час запису до лікаря в системі Medics наявна лише основна інформація, така як місце роботи, номер телефону, відгуки пацієнтів лікаря. Навпроти, Helsi має додаткову детальну інформацію про досвід та освіту лікаря.

ELEKS Portal є ще однією системою для управління медичними закладами, яка поєднує в собі численні функціональні можливості, подібні до Helsi та Medics. ELEKS Portal надає такі можливості, як реєстрація користувачів, захист особистої медичної інформації, запис на прийом до лікаря, ведення медичної

історії пацієнта, функціонал планування дій. ELEKS Portal вирізняється своїм інтуїтивно зрозумілим інтерфейсом та можливістю інтеграції з різними медичними системами та пристроями, що забезпечує безперебійну роботу медичного закладу та покращує якість обслуговування пацієнтів. Портал також підтримує багатомовність та надає гнучкі налаштування для різних типів користувачів.

В системі, що розробляється, планується реалізувати подібний функціонал ведення медичної інформації. Однак вебдодаток системи підтримки та планування роботи пансіонату має ряд особливостей реалізації функціоналу, не представленого в розглянутих аналогах, а саме:

- Відвідувачі пансіонату взаємодіють із закладом кожного дня протягом їх перебування на відпочинку, тому доцільно реалізувати планування дня відвідувача. Це передбачає вибір користувачем певних процедур, які він бажає відвідати протягом доби. Також система має мати функціонал автоматичного планування;
- За лікарями закріплені процедури, які вони проводять, тому відвідувач має запис саме на процедуру, а не безпосередньо до лікаря;
- Система оцінки задоволення закріплюється за певною процедурою, а не за лікарем;
- При записі на процедуру основна увага користувача має бути віддана опису процедури. Інформація про лікаря, що проводить процедуру, є другорядною;
- Особливістю системи, що реалізовується, є наявність модуля, що реалізує обмін повідомленнями між лікарем і відвідувачем. Користувач може задати певне питання щодо процедури у лікаря, що проводить цю процедуру.

Розробники даних медичних систем наголошують на безпеці своїх додатків, а саме шифруванню інформації та використанню захищених баз даних.

Загалом можна виділити необхідний для роботи системи підтримки рекреаційного центру функціонал, який реалізований в цих двох системах:

					ІПЗ.КР.Б-121-24-ІЗ	Арк.
						17

- реєстрація та вхід для пацієнтів, лікарів та адміністраторів;
- захист особистої медичної інформації;
- можливість пацієнтів записуватися на процедури;
- зберігання та оновлення інформації про пацієнтів.

Функціонал, що не наявний в цих двох системах, але вимагає реалізації в системі підтримки та планування роботи пансіонату:

- Планування дня відвідувача;
- Обмін повідомленнями з лікарем;
- Деталізація інформації про відвідувача;
- Наявність системи оповіщення пацієнта, щодо процедур, на які він має записи;

В таблиці 1.1 наведена загальна характеристика і порівняння досліджуваних систем.

Таблиця 1.1

Порівняльна таблиця

Параметр	Helsi	Medics	ELEKS Portal	Додаток що розробляється
Реєстрація користувачів	+	+	+	+
Захист особистої медичної інформації	+	+	+	+
Запис на прийом до лікаря	+	+	+	+
Ведення медичної історії пацієнта	+	+	+	+
Функціонал планування дій	-	-	+	+
Деталізація інформації про пацієнта	-	+	+	+
Система оцінки задоволення	+	+	-	+
Обмін повідомленнями з лікарем	-	-	-	+
Функціонал оповіщення пацієнтів	+	-	-	+

Наведені в таблиці параметри є загальними для 4 систем, назви в залежності від системи можуть різнитись.

1.3 Вибір архітектури веб-орієнтованої системи підтримки та планування роботи пансіонату

Вибір архітектури веб-орієнтованої системи медичного закладу має вирішальне значення для успішності розробки та ефективного функціонування програмного продукту. Серед сучасних архітектурних патернів можна виділити:

Монолітна архітектура: Монолітна архітектура є традиційним та простим підходом у розробці програмного забезпечення. У такій системі всі компоненти, такі як клієнтська частина, серверна логіка та база даних, розглядаються як одне ціле і розгортаються разом на одному сервері чи в одному контейнері. Основною перевагою монолітної архітектури є її простота. Завдяки тому, що всі компоненти знаходяться разом, розробка, тестування та впровадження програмного забезпечення можуть бути досить простими і зрозумілими. Цей підхід також може бути зручним для невеликих проєктів або стартапів з обмеженими ресурсами, оскільки не вимагає складних інфраструктурних рішень. Однак монолітна архітектура може мати обмежену масштабовність та гнучкість. При збільшенні обсягу програмного забезпечення стає складніше вести його розробку та підтримку. Крім того, великі монолітні системи можуть мати проблеми з швидкістю розгортання нових функцій та оновлень, оскільки будь-які зміни в одній частині системи можуть впливати на інші частини.

Мікросервісна архітектура: У цьому підході система розбивається на невеликі незалежні сервіси, які можуть бути розгорнуті та масштабовані окремо. Кожен сервіс виконує обмежену функціональність і взаємодіє з іншими сервісами через API. Завдяки використанню API для взаємодії між сервісами, мікросервіси можуть легко комунікувати один з одним. Це дозволяє створювати складні системи з різноманітними функціями, використовуючи найкращі практики розробки та інтеграції програмного забезпечення. Однак, робота з

мікросервісною архітектурою також має свої виклики. Управління багатьма сервісами може бути складним завданням, і необхідно враховувати аспекти, такі як моніторинг, логування, тестування та координація між сервісами. Крім того, збільшується складність розгортання та управління інфраструктурою.

Архітектура "Клієнт-сервер": Це розподілена архітектура, де клієнтська частина взаємодіє з сервером через мережу, щоб отримати доступ до ресурсів та послуг. Цей підхід дозволяє розділити функціональність між клієнтом та сервером, що сприяє зниженню навантаження на клієнтську сторону та полегшує розподілення ресурсів. На клієнтській стороні зазвичай розміщені користувачські інтерфейси (наприклад, вебдодатки або мобільні додатки), які взаємодіють з серверами через мережу (наприклад, Інтернет). Клієнти надсилають запити до сервера для отримання ресурсів або виконання певних операцій, наприклад, отримання медичної інформації, запис на прийом до лікаря, перегляд рецептів тощо. Крім того, на клієнтській стороні можуть виконуватися певні операції з обробки даних або відображення інформації користувачу.

Серверна частина відповідає за обробку запитів від клієнтів та надання відповідей. Вона містить бізнес-логіку, базу даних та інші компоненти, необхідні для обробки запитів. Сервери приймають запити від клієнтів, обробляють їх, звертаючись до бази даних чи інших джерел інформації, і надсилають відповіді назад клієнтам.

Зважаючи на поставлені задачі була обрана архітектура "Клієнт-сервер" для веб-орієнтованої системи медичного закладу. З огляду на специфіку медичної галузі, де швидкий та безпечний доступ до даних важливий, ця архітектурна модель дозволяє ефективно організувати обмін інформацією між клієнтами (пацієнтами) та сервером (медичним закладом). Вона сприяє забезпеченню безпеки та конфіденційності медичних даних, полегшує управління ресурсами та масштабування системи. Розподілення функціональності між клієнтом та сервером дозволяє оптимізувати використання ресурсів, забезпечує зниження навантаження на клієнтську сторону та покращує швидкодію системи. Такий підхід також сприяє

забезпеченню модульності та гнучкості системи, що важливо для подальшого розвитку та супроводження програмного продукту. Наприклад вибір клієнт-серверної архітектури дає можливість створення в майбутньому мобільного додатку, який буде використовувати сервер.



Рисунок 1.7 – діаграма, що показує роботу клієнт-серверної архітектури

1.4 Обґрунтування вибору інструментальних засобів та вимоги до апаратного забезпечення

Для обґрунтування вибору інструментальних засобів та визначення вимог до апаратного забезпечення були визначені функціональні вимоги та особливості системи, а також принципи, якими вона керується.

З огляду на те, що система є веб-орієнтованою, основною метою якої є надання послуг як медичному персоналу, так і відвідувачам, можна виділити наступні ланки, де потрібно виділити інструментальні засоби.

Для розробки серверної частини системи буде використаний фреймворк NestJS[2]. NestJS - це потужний фреймворк для розробки серверних додатків на Node.js, який пропонує модульну архітектуру та використовує концепції, взяті з Angular[3]. Він створений з урахуванням потреб сучасного розробника, який

шукає ефективні та легко збережені рішення для своїх проєктів. Однією з ключових переваг NestJS є його модульність. Фреймворк дозволяє створювати додатки з чіткою структурою, розділяючи їх на логічні модулі. Це сприяє збереженню чистоти коду та забезпечує зручне керування функціональністю додатку. Ще однією перевагою NestJS є його вбудований інжектор залежностей. Це дозволяє ефективно керувати залежностями в додатку, що робить код більш читабельним та підтримуваним.

Також важливо відзначити наявність middleware та обробників запитів в NestJS. Ці функції дозволяють легко обробляти HTTP-запити, валідувати дані, автентифікувати користувачів та виконувати інші операції, що робить процес розробки ефективнішим та швидшим.

Враховуючи особливості нашого медичного додатку, який потребує надійності, масштабовності та чистоти коду, NestJS є ідеальним вибором. Він дозволяє створити структурований та ефективний серверний додаток, який відповідає потребам сучасної розробки програмного забезпечення.

Для зберігання даних в роботі буде використана система управління базами даних PostgreSQL[4]. Це потужна та надійна реляційна база даних, яка підтримує широкий спектр функцій, таких як транзакції, підтримка структури даних JSON та XML, реплікація даних та багато інших. Вона ідеально підходить для зберігання медичної інформації та забезпечує високу надійність та швидкодію.

Для клієнтського додатку буде використаний фреймворк Next.js[5]. Це фреймворк для розробки вебдодатків на основі React, який надає широкий спектр можливостей для зручного та ефективного створення клієнтських додатків.

Однією з ключових переваг Next.js є підтримка рендерингу на стороні сервера (SSR) та генерації статичних сайтів (SSG). Це дозволяє зменшити час завантаження сторінок, поліпшити індексацію вебсайту пошуковими системами та покращити користувацький досвід.

Крім того, Next.js має потужну систему маршрутизації, яка дозволяє зручно керувати URL-адресами та переходами між сторінками. Це важливо для створення зручного та логічного користувацького інтерфейсу.

Next.js також підтримує модель сторінок з вкладеними компонентами, що дозволяє створювати багаторівневу архітектуру додатків та зручно організовувати код.

Враховуючи потреби системи, яка потребує швидкого завантаження сторінок, зручної навігації та ефективного відображення даних, Next.js є ідеальним вибором. Він дозволяє створювати масштабовані, швидкі та зручні клієнтські додатки, які відповідають сучасним стандартам розробки вебдодатків.

Створення систем на цих фреймворках передбачає використання мови програмування TypeScript[6]. Це мова програмування, яка є розширенням JavaScript, призначена для розробки великих та складних програмних проєктів. Основні переваги TypeScript полягають в тому, що ця мова програмування надає статичну типізацію, покращує читабельність коду та допомагає виявляти помилки на етапі розробки.

Однією з ключових переваг TypeScript є можливість визначення типів даних для змінних, параметрів функцій, об'єктів та інших елементів програми.

Це дозволяє підвищити надійність програмного коду та запобігти багато типових помилок, що можуть виникнути при роботі з JavaScript.

Крім того, TypeScript надає розширену підтримку сучасних функцій JavaScript, таких як стрілкові функції, асинхронні функції та інші. Вона також підтримує використання класів, інтерфейсів та інших конструкцій, що спрощує організацію коду та підвищує його читабельність.

У випадку великих програмних проєктів, таких як медична система, яка має складну структуру та великий обсяг коду, використання TypeScript дозволяє покращити стабільність, надійність та підтримку системи. Враховуючи це, використання TypeScript являє собою раціональний вибір для розробки як клієнтської, так і серверної частин програмного продукту.

Під час розробки серверної частини вебдодатку постає необхідність у тестуванні впроваджених функціональних можливостей. Для цього буде використана програма — Postman[7].

Postman - це платформа для тестування та розробки програмного забезпечення (API), яка дозволяє розробникам створювати, тестувати, документувати та співпрацювати над API. Основні переваги Postman полягають у його зручному та інтуїтивно зрозумілому інтерфейсі, можливості автоматизації тестування API, створення колекцій запитів, генерації документації API та багато іншого.

Однією з головних переваг Postman є можливість створювати та виконувати різні типи запитів (GET, POST, PUT, DELETE тощо) до API та перевіряти їх відповіді. Це дозволяє розробникам ефективно тестувати різноманітні функції та можливості свого API перед реальним впровадженням.

Крім того, Postman надає можливість створення та організації колекцій запитів, які можуть бути легко використані для автоматизації тестування, навчання нових розробників або навіть для створення документації API. Це дозволяє зберігати та керувати набором тестових запитів та забезпечує їх легку доступність для всього розробницького колективу.

З врахуванням великої кількості функцій та можливостей, а також його простоти використання, Postman є незамінним інструментом для тестування та розробки API, зокрема для медичної системи, де важлива відповідність стандартам безпеки та точність передачі даних. Використання Postman допомагає забезпечити високу якість програмного забезпечення та зручність у взаємодії з API.

Користувач системи взаємодіє із системою шляхом використання користувацького інтерфейсу, який розгорнутий у вузлі веб-браузера. Вузол це - девайс користувача. Веб-браузер повинен правильно відображати сторінки, які використовують сучасні версії HTML (наприклад, HTML5), CSS і JavaScript. Підтримувати сучасні протоколи шифрування, такі як HTTPS, для забезпечення

безпеки під час передачі даних. Мати здатність адаптувати вебсайти до різних розмірів екранів (респонсивний дизайн).

Девайс користувача це — пристрій, який підтримує такі операційні системи як: Windows, macOS, Linux, Android, iOS. Для коректної роботи з іншим вузлом системи, пристрій користувача має підтримувати стабільний тип підключення до мережі, наприклад: Wi-Fi, Ethernet, Мобільна мережа (3G, 4G, 5G). Пристрій має мати актуальні на 2024 рік характеристики. Актуальність пристрою користувача визначається середніми даними характеристик станом на 2024 рік.

Висновки до першого розділу

У першому підрозділі був розглянутий перелік завдань для реалізації системи підтримки та планування роботи пансіонату, який обґрунтовується необхідністю вдосконалення управління та надання якісних послуг у зв'язку з швидким розвитком технологій та змінами вимог до рекреаційних закладів.

У другому підрозділі були розглянуті функціональні аналоги продукту що розробляється, проведений аналіз функціоналу. Визначені особливості системи що створюється, побудована порівняльна таблиця систем.

У третьому підрозділі було визначено клієнт-серверну архітектуру веб-орієнтованої системи, також були проаналізовані сучасні приклади архітектур веб-орієнтованих систем.

У четвертому підрозділі було проведено обґрунтування вибору інструментальних засобів та вимог до апаратного забезпечення. Вибір ґрунтується на потребах та особливостях розроблюваної системи.

РОЗДІЛ 2. ПРОЕКТУВАННЯ ВЕБ-ОРІЄНТОВАНОЇ СИСТЕМИ ПІДТРИМКИ ТА ПЛАНУВАННЯ РОБОТИ ПАНСІОНАТУ

2.1 Визначення варіантів використання та об'єктно-орієнтованої структури системи

Створення інноваційної системи для пансіонату потребує ретельного аналізу та визначення можливих варіантів її використання. В розділі — «Постановка задачі» були сформовані основні функціональні вимоги системи, які дозволять оптимізувати роботу пансіонату та покращити якість надання рекреаційних послуг.

Важливим етапом є опис користувачів системи. Ретельне визначення цих груп користувачів допомагає зрозуміти їхні потреби та очікування від системи, що в свою чергу дозволяє краще врахувати їхні потреби при проектуванні та розробці програмного забезпечення. Користувачі можуть бути розділені на дві основні групи: пацієнти — планують та відвідують процедури, оцінюють якість проведення процедури, та медичний персонал, який має інструменти для ефективного управління робочими процесами, а саме ведення медичних звітів та організований робочий графік проведення процедур. Медичні звіти містять необхідну інформацію про пацієнта та результат проходження процедури. Планування відвідування процедур можливе протягом кожного дня, враховується оптимізація роботи медичного персоналу та комфорт пацієнтів. Так як при плануванні та проведенні процедур можуть виникати непередбачувані ситуації, система має додаткову роль — адміністратор. В межі відповідальності цієї ролі входить управління ролями в системі. Для наглядної взаємодії між елементами додатку було створено Діаграму варіантів використання (рис 2.1).

Опис діаграми варіантів використання системи наведено в таблицях 2.1 — 2.8.

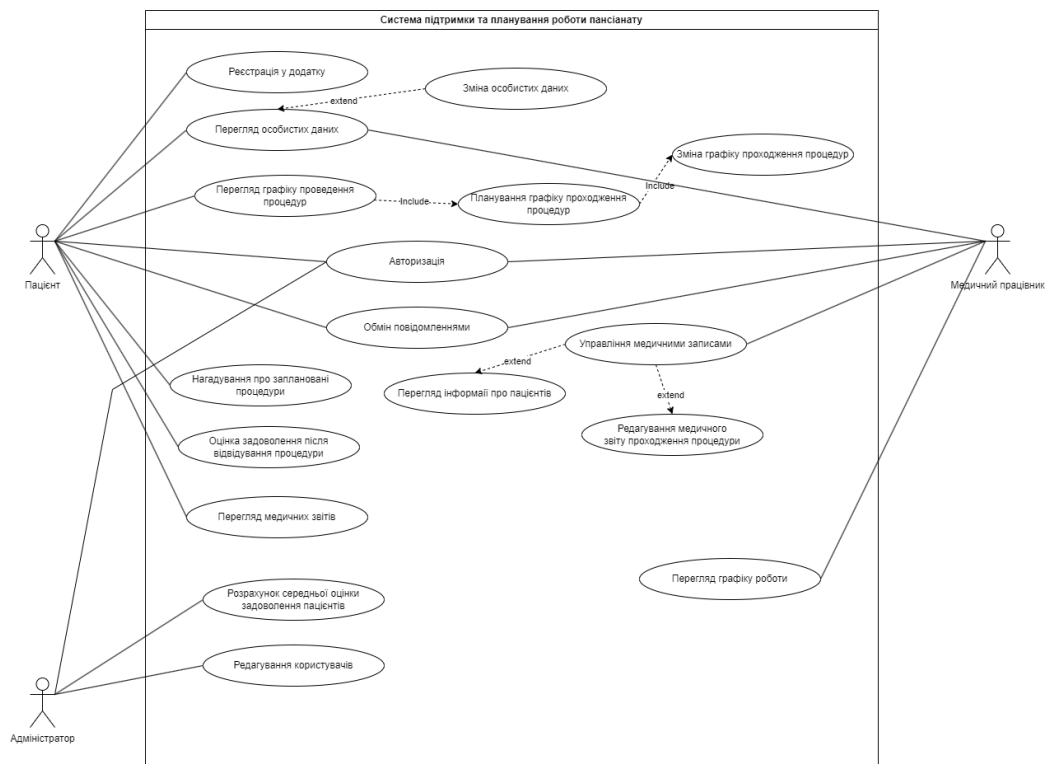


Рисунок 2.1 — діаграма варіантів використання

Таблиця 2.1

Опис прецеденту «Авторизація/Реєстрація»

Назва варіанту використання	Авторизація/Реєстрація
Основні актори	Пацієнт, медичний працівник, адміністратор
Короткий опис	Пацієнти, медичний персонал та адміністратори можуть авторизуватися у системі, використовуючи свої облікові дані, або здійснити реєстрацію як нові користувачі. У разі авторизації, користувачі мають доступ до особистого кабінету з можливістю перегляду та редагування власної інформації, а також виконання інших функцій, що відповідають їхнім ролям в системі.

Таблиця 2.2

Опис прецеденту «Обмін повідомленнями»

Назва варіанту використання	Обмін повідомленнями
Основні актори	Пацієнт, медичний працівник

Продовження таблиці 2.2

Короткий опис	Пацієнти можуть надсилати повідомлення медичному персоналу з питань їхнього стану здоров'я, запитань про процедури або будь-яких інших питань, пов'язаних з їхнім перебуванням у пансіонаті. Вони також можуть отримувати відповіді від медичного персоналу.
---------------	--

Таблиця 2.3

Опис прецеденту «Оцінка задоволення після відвідування процедури»

Назва варіанту використання	Оцінка задоволення після відвідування процедури
Основні актори	Пацієнт
Короткий опис	Після завершення процедури пацієнт має можливість виставити оцінку за певним критеріям, які можуть включати такі аспекти, як професіоналізм медичного персоналу, комфортність процедури, якість обладнання тощо.

Таблиця 2.4

Опис прецеденту «Управління медичними записами»

Назва варіанту використання	Управління медичними записами
Основні актори	Пацієнт, медичний працівник
Короткий опис	Цей прецедент містить: перегляд інформації про пацієнтів, редагування медичного звіту проходження процедури, перегляд медичних звітів. Редагування опису власних процедур.

Таблиця 2.5

Опис прецеденту «Перегляд графіку проведення процедур»

Назва варіанту використання	Перегляд графіку проведення процедур
Основні актори	Пацієнт

Продовження таблиці 2.5

Короткий опис	Цей прецедент містить: перегляд графіку проведення процедур, планування графіку проходження процедур — пацієнт може побудувати власний графік опираючись на можливість проведення процедури та власний час, зміна графіку проходження процедур — відміна проходження процедури, зміна часу проходження(якщо є така можливість). Найважчий функціонал автоматичного планування дня пацієнта
---------------	--

Таблиця 2.6

Опис прецеденту «Перегляд особистих даних»

Назва варіанту використання	Перегляд особистих даних
Основні актори	Пацієнт, медичний працівник
Короткий опис	Пацієнти можуть переглядати свої особисті дані. Медичні працівники можуть переглянути інформацію про спеціалізацію та кваліфікацію.

Таблиця 2.7

Опис прецеденту «Використання штучного інтелекту»

Назва варіанту використання	Аналітика та звіти
Основні актори	Пацієнт
Короткий опис	Пацієнти можуть отримувати інформацію про процедури, дізнаватись деталі їх проходження.

Таблица 2.8

Опис прецеденту «Редагування користувачів»

Назва варіанту використання	Редагування користувачів
Основні актори	Адміністратор
Короткий опис	Ця функція надає можливість адміністратору керувати доступом користувачів до різних функціональних можливостей системи та забезпечує безпеку та контроль над активністю користувачів.

Для розробки системи підтримки та планування роботи пансіонату було обрано об'єктно-орієнтований підхід, який забезпечує модульність, гнучкість і масштабованість додатку. Система розробляється з використанням принципів чистої архітектури, де логіка додатку і бізнес-логіка відокремлені від інфраструктурних та інтерфейсних деталей.

Чисту архітектуру (Clean Architecture) розробив і популяризував Роберт С. Мартін, також відомий як Uncle Bob, один із провідних експертів у галузі розробки програмного забезпечення[10]. Він представив концепцію чистої архітектури у своїй книзі "Clean Architecture: A Craftsman's Guide to Software Structure and Design", яка була опублікована у 2017 році[11]. Мартін відомий також як автор книг "Clean Code" та "The Clean Coder", які мають значний вплив на професійну розробку програмного забезпечення.

Метою чистої архітектури є створення програмного забезпечення, яке є легким для розуміння, тестування та розширення. Вона пропонує організацію коду таким чином, щоб логіка додатку була відокремлена від деталей реалізації, таких як бази даних, фреймворки та інші сторонні компоненти. Основним принципом чистої архітектури є поділ системи на шари з чіткими межами, де внутрішні шари не залежать від зовнішніх. Це забезпечує незалежність від інфраструктури, що дозволяє легко замінювати або оновлювати будь-який компонент системи без впливу на інші частини. Такий підхід сприяє довговічності та гнучкості програмного забезпечення, зменшуючи витрати на підтримку та розвиток у довгостроковій перспективі.

Нижче на рисунку 2.2 відображена діаграма, яка описує реалізацію чистої архітектури. Архітектура зберігає низьку когезію між шарами та високу когезію всередині кожного шару, що полегшує тестування і повторне використання компонентів. Це також дозволяє розробникам зосередитися на бізнес-логіці, не турбуючись про технічні деталі реалізації, що покращує продуктивність команди і якість кінцевого продукту.

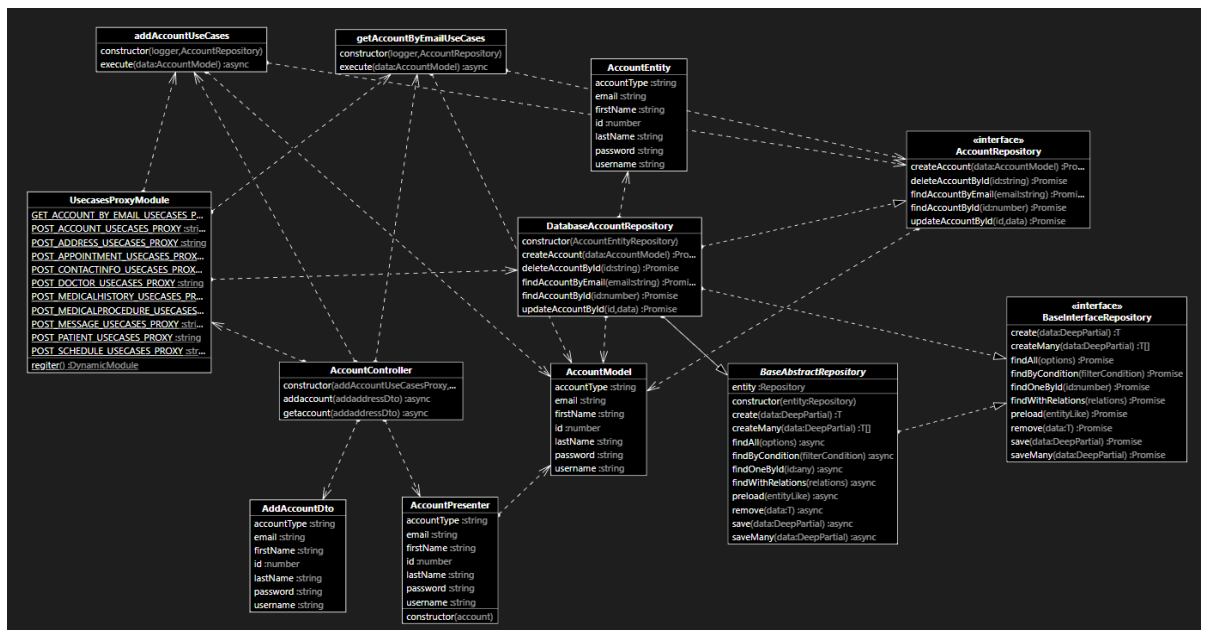


Рисунок 2.3 - Діаграма класів сутності account

На шарі domain визначений клас AccountModel який описує модель сутності, має відповідні поля. Також на цьому шарі визначений інтерфейс репозиторію, який визначає методи, які класи шару infrastructure мають реалізувати та виконувати. Так як цей інтерфейс використовує клас моделі, цей клас та інтерфейс поєднані зв'язком — залежність.

Шар infrastructure має клас сутності. Сутність є частиною інфраструктури лише з тієї причини, що використовує зовнішній модуль. У цій області був використаний патерн проектування - Repository (Репозитарій). Патерн Repository використовується для інкапсуляції логіки доступу до даних та надання єдиного інтерфейсу для роботи з даними. Він надає абстракцію рівня доступу до даних та приховує деталі реалізації бази даних чи іншого джерела даних. Патерн репозиторій складається з таких класів як:

- BaseAbstractRepository – загальний абстрактний клас, який забезпечить спільні операції з базою даних для своїх підкласів. Цей клас має інтерфейс в якому визначені методи взаємодії з базою даних. Інтерфейс та клас мають зв'язок — реалізація;
- DatabaseAccountRepository - є конкретною реалізацією репозиторію для AccountEntity. Це частина патерну що абстрагує доступ до даних за

набором інтерфейсів. Це дозволяє іншій частині програми взаємодіяти з базою даних, не знаючи особливостей коду доступу до бази даних. Клас розширює BaseAbstractRepository, що означає, що він успадковує загальні операції з базою даних від нього. Він також реалізує AccountRepository, який є інтерфейсом, який визначає методи, які має мати репозиторій для облікових записів.

Також шар infrastructure має клас UsecasesProху - цей клас є сполучною ланкою між usecases та infrastructure, будуть впровадженні різні служби, необхідні для usecases. Таким чином, у майбутньому буде легко змінити сервіс. Цей клас має залежності з класом іншого шару та контролером.

Контролер можна назвати маршрутом, до якого буде звертатись клієнтський додаток та отримувати певну відповідь. Контролер не може існувати без DTO (об'єкт передачі даних) є вхідними даними для контролера, це дані, що відображаються на дисплеї, і дані, перевірені перед використанням. Між контролером та DTO визначена залежність. Presenter – це вихідні дані контролера, які теж мають залежність від контролера.

Зовнішній шар — usecases. В ньому визначені всі можливі варіанти використання. На діаграмі представлений два варіанти використання — addAccountUseCases, getAccountByUseCases Цей клас має залежності з багатьма класами системи.

Під час реалізації об'єктно-орієнтованої моделі був використаний патерн проектування - Dependency Injection (DI). Це шаблон проектування, який широко використовується у фреймворку NestJS. Це техніка, за допомогою якої один об'єкт забезпечує залежності іншого об'єкта. Цей шаблон використовується для підвищення ефективності та модульності програми. Наприклад у конструкторі DatabaseAccountRepository декоратор @InjectRepository(AccountEntity) використовується для введення репозиторію AccountEntity. Це яскравий приклад ін'єкції залежностей. DatabaseAccountRepository не потрібно знати, як створити репозиторій AccountEntity. Він надається (впроваджується) під час виконання фреймворком NestJS. Ін'єкція залежностей дозволяє відокремити створення

об'єкта від його використання, що, у свою чергу, полегшує перемикання залежностей і зменшує код у конструкторах. Це також дуже корисно для модульного тестування, дозволяючи тестувати клас сам по собі.

Діаграма, що показує ініціалізацію класу з Dependency Injection і без (рис. 2.4).

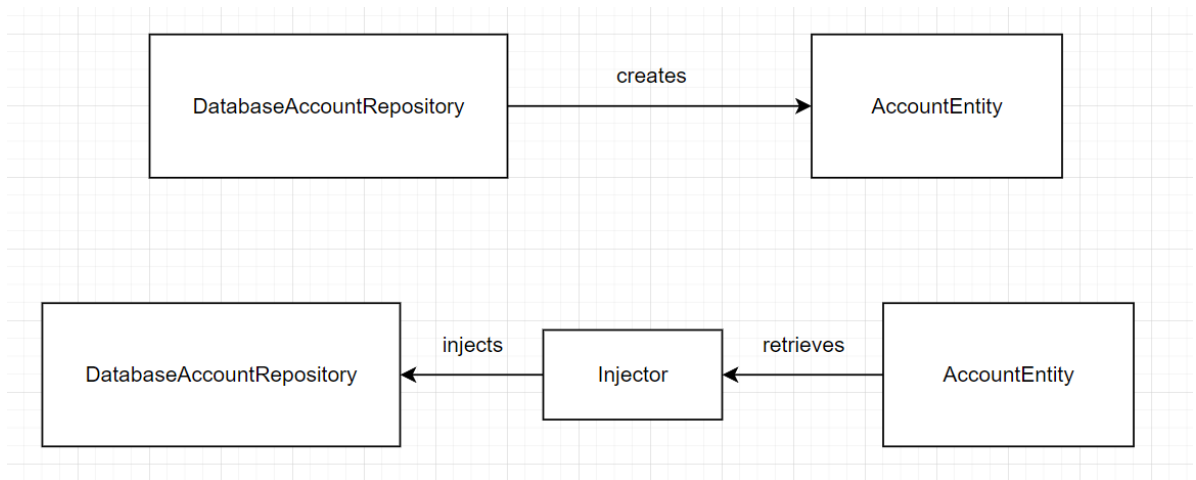


Рисунок 2.4 — Діаграма, що відображає патерн проектування DI

Діаграма компонентів (рис. 2.5) ілюструє архітектуру системи підтримки та планування роботи пансіонату, розбиваючи її на кілька ключових модулів: Domain, UseCases, Infrastructure, Gateways, API, Web.

Domain відповідає за бізнес-логіку та основні моделі даних, які є незалежними від зовнішніх залежностей. UseCases містить бізнес-логіку, яка залежить від домену і виконує конкретні випадки використання. Infrastructure забезпечує всі технічні деталі та залежності, такі як бази даних, фреймворки та зовнішні служби, необхідні для функціонування системи. Gateways виступає як проміжний шар, що ізолює бізнес-логіку від деталей реалізації управління та інтерфейсу.

API відповідає за ініціалізацію додатку, всіх налаштувань та обробку HTTP-запитів. Web представляє інтерфейс користувача, який реалізовано як React додаток. Web, включає компоненти сторінок, повторно використовувані компоненти, серверні API маршрути та middleware для обробки запитів.

Взаємодія між компонентами демонструє, як Domain взаємодіє з UseCases, які у свою чергу взаємодіють з Gateways та API. Infrastructure надає необхідні

сервіси та ресурси для Domain, UseCases і API. Web є Next.js додатком, який працює для обробки запитів і надання даних користувачам через API. Ця структура забезпечує модульність, масштабованість та ефективне управління бізнес-логікою та інтерфейсом користувача.

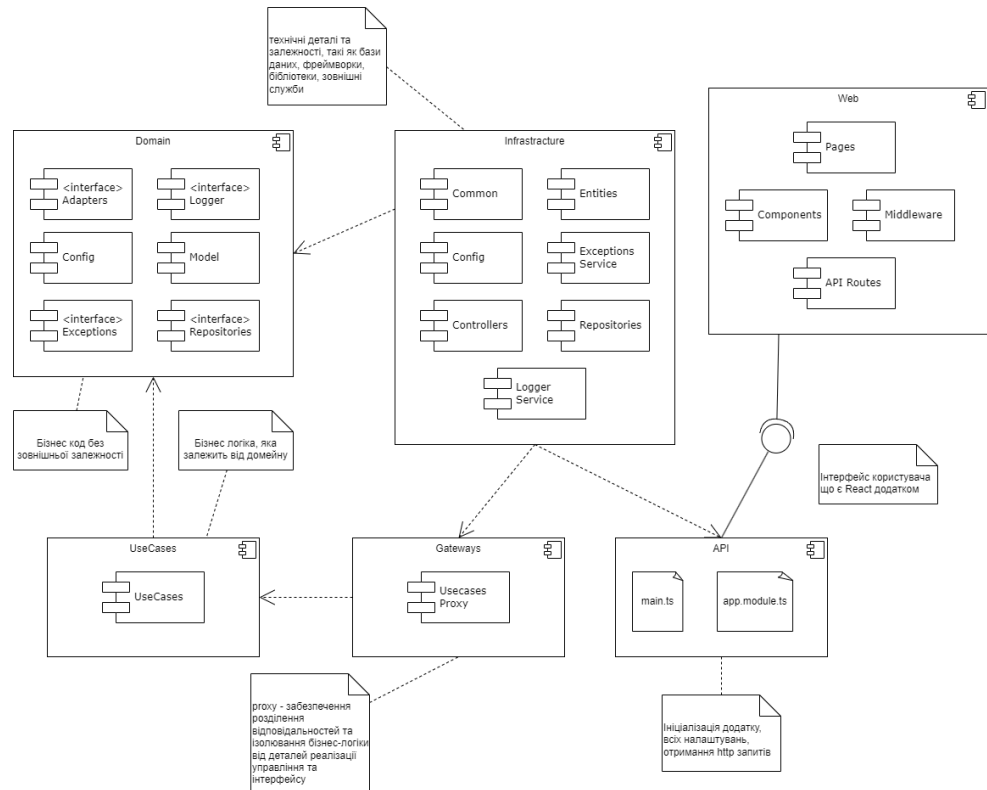


Рисунок 2.5 — Діаграма компонентів системи

2.2 Розробка бази даних системи

При розробці системи підтримки та планування роботи пансіонату було обрано PostgreSQL як основну систему управління базами даних (СУБД). PostgreSQL є потужною, відкритою та надійною реляційною СУБД, яка відома своєю стабільністю, масштабованістю та багатofункціональністю. Вибір PostgreSQL був обумовлений наступними причинами:

- Відкритий вихідний код: PostgreSQL є програмним забезпеченням з відкритим вихідним кодом, що дозволяє зменшити витрати на ліцензії та надає можливість кастомізації під конкретні потреби проекту.
- Масштабованість і продуктивність: PostgreSQL підтримує масштабування як вертикальне (збільшення потужності одного сервера), так і

горизонтальне (розподілення навантаження між кількома серверами), що дозволяє обробляти великі обсяги даних та забезпечувати високу продуктивність.

- Підтримка стандартів SQL: PostgreSQL дотримується стандартів SQL, що забезпечує сумісність з іншими системами та інструментами, а також дозволяє легко інтегрувати базу даних з іншими компонентами системи.
- Розширюваність: PostgreSQL надає можливість створення власних функцій, типів даних, індексів та інших об'єктів бази даних, що дозволяє адаптувати систему під специфічні вимоги проекту.
- Безпека: PostgreSQL має вбудовані механізми аутентифікації та авторизації, що забезпечує високий рівень безпеки даних. Підтримка шифрування та управління правами доступу дозволяє захистити конфіденційну інформацію.
- Спільнота та підтримка: Велика та активна спільнота розробників PostgreSQL забезпечує регулярні оновлення, виправлення помилок та підтримку користувачів, що робить цю СУБД надійним вибором для довгострокових проектів.

Для реалізації функціональності системи підтримки та планування роботи пансіонату було створено кілька таблиць у базі даних. Нижче наведено опис кожної таблиці та її атрибутів.

Таблиця 2.9

Таблиця account, що зберігає інформацію про облікові записи користувачів.

Поле	Тип даних	Опис
id	integer	Унікальний ідентифікатор
updated_date	timestamp without time zone	Дата останнього оновлення
create_date	timestamp without time zone	Дата створення облікового запису
password	text	Хешований пароль користувача
last_login	timestamp without time zone	Дата останнього входу в систему

Таблиця 2.10

Таблиця doctor зберігає інформацію про лікарів.

Поле	Тип даних	Опис
id	integer	Унікальний ідентифікатор лікаря
specialty	character varying	Спеціальність лікаря
qualification	character varying	Кваліфікація лікаря
account_id	integer	Ідентифікатор облікового запису користувача

Таблиця 2.11

Таблиця message зберігає повідомлення між користувачами.

Поле	Тип даних	Опис
id	integer	Унікальний ідентифікатор повідомлення
specialty	character varying	Роль відправника повідомлення
messageContent	bytea	Вміст повідомлення
timestamp	timestamp without time zone	Час відправлення повідомлення
sender_id	integer	Ідентифікатор відправника
receiver_id	integer	Ідентифікатор одержувача

Таблиця 2.12

Таблиця patient зберігає інформацію про пацієнтів.

Поле	Тип даних	Опис
id	integer	Унікальний ідентифікатор пацієнта
account_id	integer	Ідентифікатор облікового запису користувача
recovery_status	boolean	Статус одужання пацієнта, за замовчуванням false
additional_info	bytea	Додаткова інформація про пацієнта/

Таблиця 2.13

Таблиця patient_procedure зберігає інформацію про записи на процедури.

Поле	Тип даних	Опис
id	integer	Унікальний ідентифікатор процедури пацієнта
patient_id	integer	Ідентифікатор пацієнта
doctor_id	integer	Ідентифікатор лікаря
procedure_id	integer	Ідентифікатор процедури
procedure_date	date	Дата проведення процедури
created_date	timestamp without time zone	Дата створення запису
updated_date	timestamp without time zone	Дата останнього оновлення запису
appointment_time	time without time zone	Час призначення процедури
rating	integer	Оцінка процедури пацієнтом
report	bytea	Звіт про проведену процедуру

Таблиця 2.14

Таблиця procedure зберігає інформацію про медичні процедури.

Поле	Тип даних	Опис
id	integer	Унікальний ідентифікатор процедури
doctor_id	integer	Ідентифікатор лікаря
procedure_name	character varying	Назва процедури
procedure_description	character varying	Опис процедури
average_rating	double precision	Середня оцінка процедури
deleted	boolean	Статус існування процедури

Для забезпечення безпеки конфіденційних даних у системі підтримки та планування роботи пансіонату використовується шифрування. Шифрування застосовується для захисту інформації, що зберігається у полях типу bytea. Це включає такі поля, як additional_info у таблиці patient та report у таблиці patient_procedure, а також messageContent у таблиці message.

Всі дані, що зберігаються у полях типу bytea, шифруються перед збереженням у базі даних. Це забезпечує додатковий рівень безпеки, оскільки

навіть у випадку компрометації бази даних, конфіденційна інформація залишається захищеною.

Для шифрування даних використовується функція `pgp_sym_encrypt` (рис. 2.6), яка забезпечує симетричне шифрування з використанням вказаного ключа.

Приклад реалізації шифрування у програмному коді наведено нижче:

```
1 async encryptReport(report: any, encryptionKey: string): Promise<any> {  
2   const encryptedContent = await this.patientProcedureEntityRepository.query(  
3     `SELECT pgp_sym_encrypt($1, $2) AS encrypted_report`,  
4     [report, encryptionKey],  
5   );  
6   return encryptedContent[0].encrypted_report;  
7 }
```

Рисунок 2.6 – Використання симетричного шифрування

На діаграмі нижче (рис. 2.7) представлена структура бази даних системи підтримки та планування роботи пансіонату. База даних складається з таблиць, які з'єднані між собою різними типами зв'язків. Основні зв'язки між таблицями включають первинні та зовнішні ключі, що забезпечує цілісність даних і правильне відображення взаємодії між сутностями.

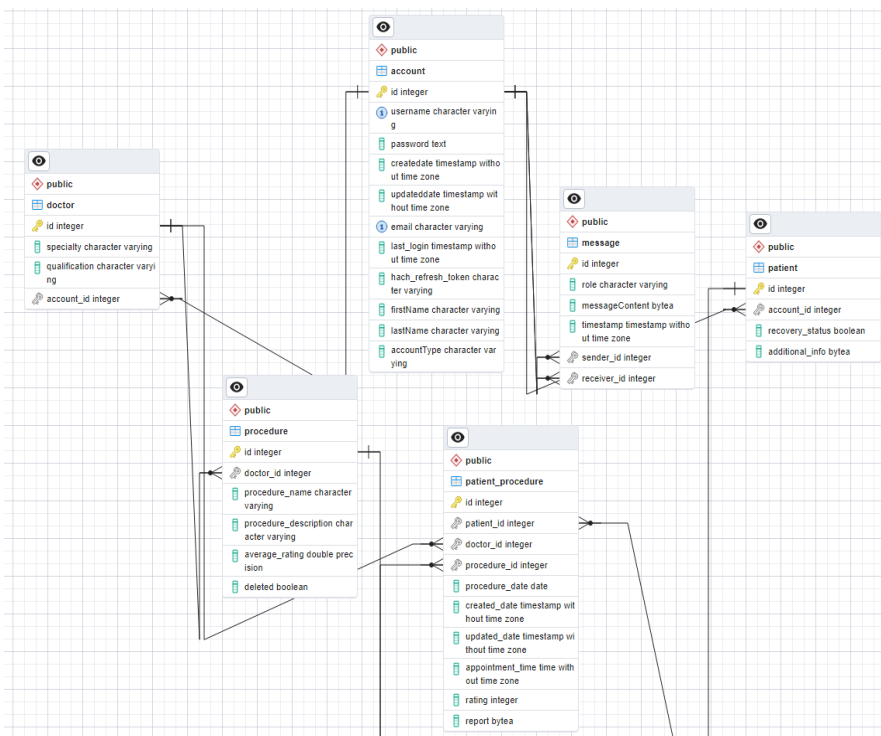


Рисунок 2.7 — Діаграма структури бази даних

Використання зв'язків між таблицями бази даних забезпечує цілісність даних та логічну структуру зберігання інформації. Кожен зв'язок виконує конкретну роль у моделюванні реальних взаємодій між сутностями системи. Наприклад, зв'язок між таблицями `account` і `patient` дозволяє зберігати персональні дані пацієнтів разом з їх обліковими записами, а зв'язок між таблицями `doctor` і `procedure` дозволяє визначати, які процедури може проводити конкретний лікар.

Такі зв'язки також спрощують доступ до даних та їх обробку. Наприклад, для отримання списку процедур, які пройшов конкретний пацієнт, можна зробити запит до таблиці `patient_procedure` з фільтрацією за `patient_id`. Це забезпечує ефективність та зручність роботи з даними в системі.

Для реалізації функціоналу, пов'язаного з повідомленнями, використовується БД - MongoDB, яка є нереляційною базою даних[12]. MongoDB забезпечує гнучке зберігання документів у форматі JSON, що робить її ідеальною для динамічних та швидкозмінних даних, таких як підписки на повідомлення. У MongoDB дані зберігаються у вигляді колекцій, що є аналогом таблиць у реляційних базах даних. Кожен документ у колекції є аналогом рядка таблиці і зберігається у форматі JSON. Це дозволяє зберігати складні структури даних у простій та гнучкій формі.

Для зберігання підписок на push-повідомлення використовується колекція `subscriptions` (рис. 2.8). Кожен документ у цій колекції містить інформацію про одну підписку, включаючи кінцеву точку для надсилання повідомлень та ключі для шифрування.

```
_id: ObjectId('66676484b3e634b3ce88a5bd')
endpoint: "https://updates.push.services.mozilla.com/wpush/v2/gAAAAABmZ2SE5VjfQQW..."
keys: Object
  auth: "S-LUvRgLuYcsZyPRLJt2yg"
  p256dh: "BCW-vLECGW8vseg4SDav09_39XXN2ZITSuNad19hFo46j7xgMbIw6j7dbPQtwZ4gntcRny..."
patientId: 34
```

Рисунок 2.8 — Приклад структури документу в колекції `subscriptions`

Структура документа в колекції subscriptions

Поле	Опис
_id	Унікальний ідентифікатор документа
endpoint	URL-адреса, на яку надсилається push-повідомлення. Це ключове поле, яке вказує на кінцеву точку підписки.
keys.p256dh	Публічний ключ для шифрування повідомлень. Використовується для забезпечення конфіденційності переданих даних.
keys.auth	Ключ аутентифікації, який забезпечує цілісність та правомірність повідомлення.

2.3 Проектування та реалізація алгоритмів роботи системи

У сучасних програмних системах важливо не лише забезпечити їх функціональність, але й спроектувати ефективні алгоритми роботи, які б забезпечували стабільну та надійну роботу додатку. Один з важливих аспектів роботи моєї системи — це процес запису користувача на процедуру. Далі представлено опис алгоритму роботи системи на основі діаграми активності (рис. 2.9).

Процес починається з того, що користувач здійснює вхід на веб-сайт. Він проходить етап аутентифікації. На цьому етапі система перевіряє, чи є користувач зареєстрованим. Якщо користувач не зареєстрований, система пропонує йому зареєструватися. Після успішної реєстрації користувач проходить аутентифікацію. Далі користувач обирає потрібну процедуру з доступного списку. Також пацієнт обирає бажану дату та час для запису на процедуру.

Система перевіряє, чи існує вже запис на іншу процедуру з ідентичною датою та часом. Якщо такий запис існує, користувач отримує повідомлення про помилку через дублювання записів.

Цей алгоритм забезпечує, що користувач не зможе записатися на дві процедури одночасно, забезпечуючи правильне планування процедур та уникнення дублювань. В результаті система гарантує стабільність та надійність процесу запису, що підвищує загальну ефективність роботи додатку.

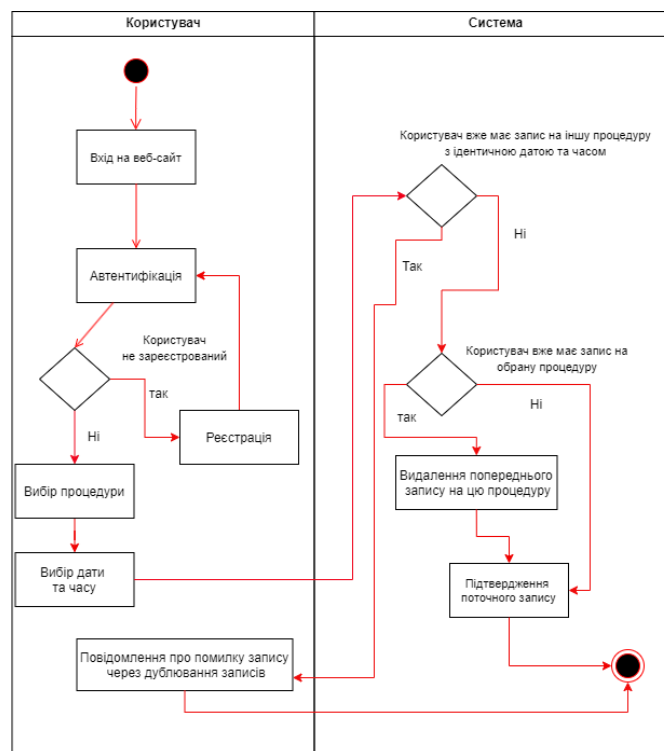


Рисунок 2.9 — Діаграма активності запису пацієнта на процедуру

Далі наведено опис реалізації цього алгоритму у коді. Метод `update` починається з перевірки, чи є тип облікового запису пацієнтом (рис. 2.10). Якщо це не так, виконується відповідна обробка.

отримання цієї інформації використовуються відповідні методи репозиторію. Це дозволяє забезпечити точне планування та уникнути конфліктів у розкладі процедур.

Асинхронне завантаження даних гарантує швидке і ефективне отримання необхідної інформації, зменшуючи затримки у виконанні операцій. Завдяки цьому підходу, система може динамічно реагувати на зміни у розкладі та забезпечувати актуальність даних для користувачів.



```

1 let inactiveProcedures = [];
2 let overlappingProcedures = [];
3
4 if (existingProcedure) {
5   [inactiveProcedures, overlappingProcedures] = await Promise.all([
6     this.patientProcedureRepository.getInactiveProceduresByDoctorAndTime(
7       existingProcedure.doctorId,
8       existingProcedure.appointmentTime,
9       existingProcedure.procedureDate,
10    ),
11    this.patientProcedureRepository.getProceduresByDoctorAndTime(
12      currentProcedure.doctorId,
13      currentProcedure.appointmentTime,
14      procedureDate,
15    ),
16   ]);
17 } else {
18   overlappingProcedures =
19     await this.patientProcedureRepository.getProceduresByDoctorAndTime(
20       currentProcedure.doctorId,
21       currentProcedure.appointmentTime,
22       procedureDate,
23     );
24 }

```

Рисунок 2.13 — Отримання неактивних та перекриваючих процедур

Якщо є існуюча процедура, вона оновлюється: встановлюється новий ідентифікатор пацієнта, створюється звіт, оновлюється дата (рис. 2.14). Загалом процедура стає активною для запису іншим користувачем, пацієнт з ідентифікатором 28 — це адміністратор, всі процедури доступні до запису мають цей ідентифікатор. Також неактивні процедури що були заблоковані через існуючу процедуру, стають активними.

```

1  try {
2      if (existingProcedure) {
3          existingProcedure.updatedDate = new Date();
4          existingProcedure.patientId = 28;
5          existingProcedure.report = { report: 'звіт' };
6          existingProcedure.rating = 0;
7          await this.patientProcedureRepository.updatePatientProcedure(
8              existingProcedure.id,
9              existingProcedure,
10             );
11
12         await Promise.all(
13             inactiveProcedures.map(async (procedure) => {
14                 if (procedure.patientId !== patient.id && procedure.id !== +id) {
15                     procedure.patientId = 28;
16                     procedure.report = { report: 'звіт' };
17                     procedure.updatedDate = new Date();
18                     await this.patientProcedureRepository.updatePatientProcedure(
19                         procedure.id,
20                         procedure,
21                     );
22                 }
23             }),
24         );
25     }

```

Рисунок 2.14 — Оновлення поточної та неактивних процедур

Поточна процедура також оновлюється з новими даними пацієнта та звітом(рис. 2.15).

```

1  currentProcedure.updatedDate = new Date();
2  currentProcedure.patientId = patient.id;
3  currentProcedure.report = { report: 'звіт' };
4
5  const result =
6      await this.patientProcedureRepository.updatePatientProcedure(
7          +id,
8          currentProcedure,
9      );

```

Рисунок 2.15 — Оновлення поточної процедури

Перебиваючі процедури створюються для поточної процедури, змінюючи ідентифікатори пацієнтів. Ідентифікатор 29 встановлюється для процедур які є неактивними (рис. 2.16).

```

1  await Promise.all(
2      overlappingProcedures.map(async (procedure) => {
3          if (procedure.patientId === 28 && procedure.id !== +id) {
4              procedure.patientId = 29;
5              procedure.report = { report: 'звіт' };
6              procedure.updatedDate = new Date();
7              await this.patientProcedureRepository.updatePatientProcedure(
8                  procedure.id,
9                  procedure,
10             );
11         }
12     })),
13 );

```

Рисунок 2.16 — Оновлення перекриваючих процедур

Кінцевий етап процесу включає логування успішного виконання операції та обробку можливих винятків, таких як дублювання записів (рис. 2.17). Логування дозволяє зберегти інформацію про кожну успішну операцію, що забезпечує відстеження історії дій і полегшує діагностику у разі виникнення проблем. Якщо виникає конфлікт, система обробляє його за допомогою відповідного винятку, що дозволяє користувачам отримати інформативне повідомлення про помилку та вжити необхідних заходів для її усунення.

Для запобігання одночасного запису пацієнтів на дві різні процедури в один і той же час, був створений індекс в базі даних, який дозволяє існування як активних, так і неактивних записів (рис. 2.18). Це рішення забезпечує унікальність записів у системі, запобігаючи можливим конфліктам і забезпечуючи коректне функціонування системи запису пацієнтів на процедури. Індекс працює таким чином, що він перевіряє наявність активного запису для конкретного пацієнта в заданий час перед дозволом створення нового запису, що значно підвищує надійність системи.

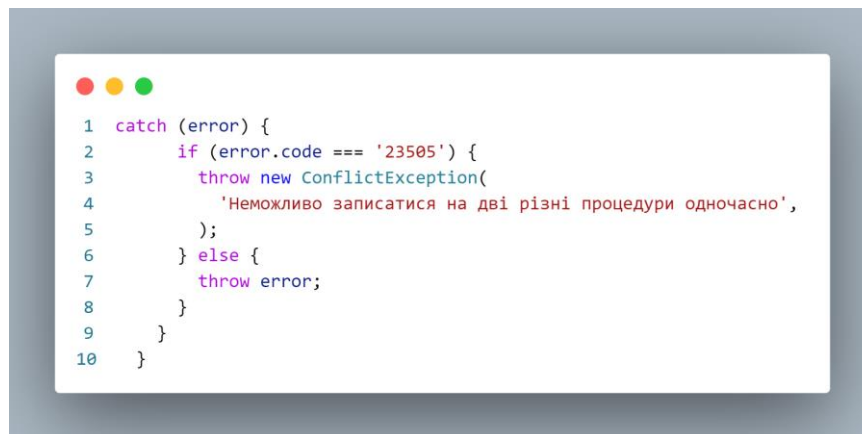


Рисунок 2.17 — Виключення для конфліктних записів

```

CREATE UNIQUE INDEX IF NOT EXISTS unique_patient_procedure
ON public.patient_procedure USING btree
(patient_id ASC NULLS LAST, procedure_date ASC NULLS LAST, appointment_time ASC NULLS LAST)
TABLESPACE pg_default
WHERE patient_id <> ALL (ARRAY[28, 29]);

```

Рисунок 2.18 — Створений індекс

Діаграма станів (рис. 2.19), яка описує процес запису пацієнта на процедуру, проходження процедури, та отримання звіту в медичній системі.

- Очікування запису пацієнта:
 - **Стан:** Лікар очікує запису пацієнта на процедуру.
 - **Перехід:** Якщо пацієнт записується на процедуру, система переходить до наступного стану. Якщо пацієнт скасовує запис, цикл завершується.
- Пацієнт записався на процедуру:
 - **Стан:** Після того як пацієнт записався на процедуру, графік лікаря оновлюється.
 - **Перехід:** Пацієнт відвідує лікаря для проходження процедури.
- Пацієнт пройшов процедуру:
 - **Стан:** Пацієнт проходить процедуру у лікаря. Цей стан поділяється на кілька підстанів:
 - Заповнення звіту проходження процедури;
 - Оцінка задоволення проходження процедури пацієнтом.
- Пацієнт переглянув звіт проходження процедури:

- **Стан:** Пацієнт переглядає звіт про проходження процедури;
- **Перехід:** Після перегляду звіту пацієнт виконує необхідні заходи відповідно до рекомендацій лікаря;

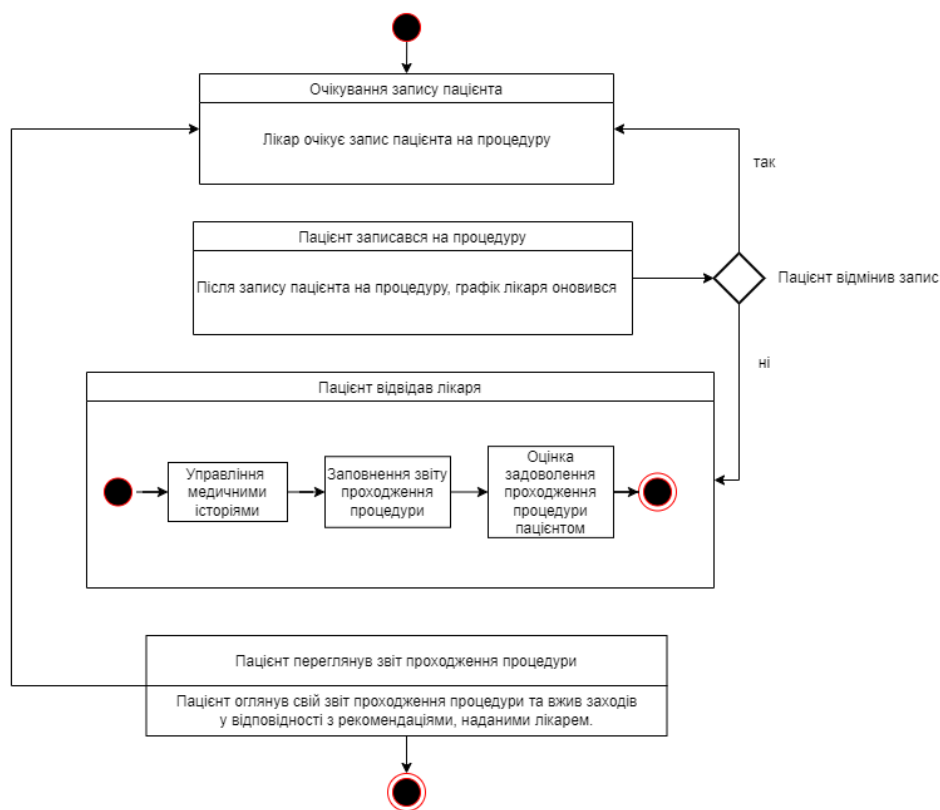


Рисунок 2.19 — Діаграма станів запису пацієнта на процедуру

Тепер розглянемо етапи, показані на діаграмі, більш детально, описуючи код, який реалізує ці процеси. Ми детально розберемо деякі з етапів, щоб зрозуміти, як саме система виконує ці операції на рівні коду.

Спочатку розглянемо процес скасування запису на процедуру, який складається з кількох важливих кроків: перевірка наявності запису, перевірка прав доступу, оновлення статусу процедури та обробка неактивних процедур.

Давайте детально опишемо кожен з цих кроків.

Перевірка наявності запису на процедуру.

Перше, що потрібно зробити, це переконатися, що запис на процедуру існує. Для цього використовується метод `getPatientProcedureById`, який отримує запис за його ідентифікатором:

```
1 async cancelProcedure(id: string, account: any) {
2     const patientProcedure =
3         await this.patientProcedureRepository.getPatientProcedureById(+id);
4     if (!patientProcedure) {
5         throw new ForbiddenException('Запис не найден');
6     }
7 }
```

Рисунок 2.20 — перевірка існування запису на процедуру

Якщо запис не знайдено, в системі спрацьовує виняток `ForbiddenException` з повідомленням "Запис не знайдений" (рис. 2.20). Це важливий крок для забезпечення цілісності даних і уникнення непередбачуваних помилок.

Перевірка прав доступу.

Наступний крок — це перевірка, чи дійсно пацієнт, який намагається скасувати запис, має на це право (рис. 2.21). Спочатку отримуємо інформацію про пацієнта за ідентифікатором облікового запису:

```
1  const patient = await this.patientRepository.getPatientByAccountId(  
2      account.id,  
3  );  
4  if (!patient) {  
5      throw new ForbiddenException('Пацієнт не знайдений');  
6  }
```

Рисунок 2.21 — Перевірка існування пацієнта

Оновлення статусу процедури.

Після перевірки прав доступу система оновлює статус процедури, встановлюючи новий ідентифікатор пацієнта на 28 (адміністратор), оновлює дату та додає звіт. Запис на процедуру стає доступним для інших користувачів системи, тобто пацієнтів(рис. 2.22).

```

1 patientProcedure.updatedDate = new Date();
2 patientProcedure.report = { report: 'Звіт' };
3 patientProcedure.patientId = 28;
4 await this.patientProcedureRepository.updatePatientProcedure(
5     +id,
6     patientProcedure,
7 );

```

Рисунок 2.22 — Скасування запису на процедуру

Обробка неактивних процедур.

Після скасування процедури система перевіряє, чи є неактивні процедури, які можуть бути розблоковані(рис. 2.23). Отримуємо список таких процедур, потім оновлюємо кожну з цих процедур, якщо вони не належать поточному пацієнту і не є тією ж самою процедурою:

```

1 const inactiveProcedures =
2     await this.patientProcedureRepository.getInactiveProceduresByDoctorAndTime(
3         patientProcedure.doctorId,
4         patientProcedure.appointmentTime,
5         patientProcedure.procedureDate,
6     );
7
8 for (const procedure of inactiveProcedures) {
9     if (procedure.patientId !== patient.id && procedure.id !== +id) {
10        procedure.patientId = 28;
11        procedure.report = { report: 'Звіт' };
12        procedure.updatedDate = new Date();
13        await this.patientProcedureRepository.updatePatientProcedure(
14            procedure.id,
15            procedure,
16        );
17    }
18 }

```

Рисунок 2.23 — Обробка неактивних записів

Функція reportProcedure призначена для оновлення звіту про процедуру пацієнта. Ця функція дозволяє лікарю додати або оновити звіт про проведену процедуру. Функція reportProcedure починається з отримання запису про процедуру за її ідентифікатором через метод getPatientProcedureById. Якщо

запис не знайдено, кидається `ForbiddenException` з повідомленням "Пацієнт не знайдений".

Далі отримуються лікарі, пов'язані з обліковим записом, за допомогою методу `findByAccountId`. Якщо лікарів не знайдено, спрацьовує `ForbiddenException` з повідомленням "Лікар не знайдений". Перевіряється, чи є серед цих лікарів той, який здійснює процедуру, за допомогою методу `some`. Якщо лікар не авторизований, кидається `ForbiddenException` з повідомленням "Доступ заборонено".

Після авторизації лікаря, функція оновлює звіт про процедуру, змінює дату оновлення і зберігає зміни в базі даних через метод `updatePatientProcedure`. В кінці логується успішне виконання операції з повідомленням "Запис на процедуру був оновлений".

```
async reportProcedure(id: string, report: object, account: any) {  
    const patientProcedure =  
        await this.patientProcedureRepository.getPatientProcedureById(+id);  
    if (!patientProcedure) {  
        throw new ForbiddenException('Пацієнт не знайдений');  
    }  
    const doctors = await this.doctorRepository.findByAccountId(account.id);  
    if (!doctors || doctors.length === 0) {  
        throw new ForbiddenException('Лікар не знайдений');  
    }  
  
    const isAuthorizedDoctor = doctors.some(  
        (doctor: any) => doctor.id === patientProcedure.doctorId,  
    );  
    if (!isAuthorizedDoctor) {  
        throw new ForbiddenException('Доступ заборонено');  
    }  
  
    patientProcedure.report = report;  
    patientProcedure.updatedDate = new Date();  
    const result = await this.patientProcedureRepository.updatePatientProcedure(  
        +id,  
        patientProcedure,  
    );  
  
    return result;  
}
```

Шифрування звітів проходження процедур є важливим аспектом забезпечення конфіденційності та безпеки медичної інформації пацієнтів. В сучасних медичних системах збереження та передача чутливих даних повинні здійснюватися із дотриманням найвищих стандартів безпеки. Для досягнення цього використовується симетричне шифрування, яке забезпечує захист даних від несанкціонованого доступу.

Функції `encryptReport` та `decryptReport` реалізують процес шифрування та дешифрування звітів про проходження процедур (рис. 2.24). Функція `encryptReport` приймає звіт та ключ шифрування, використовує SQL-запит до бази даних для виконання функції `pgp_sym_encrypt`, яка шифрує вміст звіту з використанням наданого ключа. Результат шифрування повертається як зашифрований звіт.



```

1  async encryptReport(report: any, encryptionKey: string): Promise<any> {
2      const encryptedContent = await this.patientProcedureEntityRepository.query(
3          `SELECT pgp_sym_encrypt($1, $2) AS encrypted_report`,
4          [report, encryptionKey],
5      );
6      return encryptedContent[0].encrypted_report;
7  }
8  async decryptReport(
9      encryptedReport: any,
10     encryptionKey: string,
11 ): Promise<any> {
12     const decryptedReport = await this.patientProcedureEntityRepository.query(
13         `SELECT pgp_sym_decrypt($1, $2) AS decrypted_report`,
14         [encryptedReport, encryptionKey],
15     );
16     return JSON.parse(decryptedReport[0].decrypted_report);
17 }

```

Рисунок 2.24 — Функції для шифрування та розшифрування звітів

Функція `dailyScheduleProcedure` відповідає за автоматичне планування процедур на кілька днів вперед. Це реалізовано як Cron операція, яка виконується щодня опівночі завдяки декоратору `@Cron(CronExpression.EVERY_DAY_AT_MIDNIGHT)`. Основна мета цієї функції — забезпечити наявність запланованих процедур у системі на визначений період, щоб лікарі та пацієнти мали доступ до розкладу процедур.

Функція починається з визначення фіксованого набору часових інтервалів для запису процедур: з 9:00 до 17:00 з перервою на обід. Далі отримується список всіх процедур із репозиторію процедур за допомогою методу `getProcedures`. Кількість записів для кожної процедури встановлено у 8, а кількість днів для планування — у 4.

Для кожного дня в інтервалі `daysToSchedule` створюється дата процедури, починаючи з поточного дня. Після цього для кожної процедури і для кожного визначеного часового інтервалу перевіряється наявність існуючих записів за допомогою методу `getExistenceProcedure`. Якщо записів немає, створюється новий запис для процедури з основними параметрами: ідентифікатор лікаря, пацієнта (28 — адміністратор), дата і час процедури, звіт та рейтинг. Ці записи зберігаються в базі даних за допомогою методу `createPatientProcedure`.

```
async dailyScheduleProcedure() {  
  try {  
    const appointmentTime = ['9:00','10:00','11:00','12:00','13:00',  
      '15:00','16:00','17:00',  
    ];  
    const procedures = await this.procedureRepository.getProcedures();  
    const appointmentsPerProcedure = 8;  
    const daysToSchedule = 4;  
  
    for (let dayOffset = 0; dayOffset < daysToSchedule; dayOffset++) {  
      const procedureDate = new Date();  
      procedureDate.setDate(procedureDate.getDate() + dayOffset);  
  
      const createProceduresPromises = [];  
  
      for (const procedure of procedures) {  
        for (let i = 0; i < appointmentsPerProcedure; i++) {  
          const appointmentTimeSlot = appointmentTime[i];  
  
          const createProcedurePromise = (async () => {  
            const existingProcedures =  
              await this.patientProcedureRepository.getExistenceProcedure(  
                procedure,  
                procedureDate,  
                appointmentTimeSlot,  
              );  

```

```

    if (!existingProcedures) {
        const patientProcedure = new PatientProcedureModel();
        patientProcedure.doctorId = procedure.doctorId;
        patientProcedure.patientId = 28;
        patientProcedure.procedureId = procedure.id;
        patientProcedure.procedureDate = procedureDate;
        patientProcedure.createdDate = new Date();
        patientProcedure.updatedDate = new Date();
        patientProcedure.appointmentTime = appointmentTimeSlot;
        patientProcedure.report = { report: 'Звіт' };
        patientProcedure.rating = 0;
        await this.patientProcedureRepository.createPatientProcedure(
            patientProcedure,
        );
    }
    })();
    createProceduresPromises.push(createProcedurePromise);
}
}
await Promise.all(createProceduresPromises);
}
}

```

Функція `handleCronSchedule` відповідає за виклик функції `dailyScheduleProcedure` за розкладом, використовуючи Cron операцію[13]. Декоратор `@Cron(CronExpression.EVERY_DAY_AT_MIDNIGHT)` забезпечує виконання цієї функції щодня опівночі. Під час виконання функції логуються повідомлення про початок та завершення операції, а також обробляються можливі помилки.

```

@Cron(CronExpression.EVERY_DAY_AT_MIDNIGHT)
async handleCronSchedule() {
    try {
        this.logger.log(
            'DailyProceduresSchedulerStrategy',
            'Daily schedule procedure',
        );
        this.patientProcedureUsecaseProxy.getInstance().dailyScheduleProcedure();
    } catch (error) {
        this.logger.error('DailyProceduresScheduler', error);
    }
}

```

```
}
```

Функція `handleCronDelete` налаштована на виконання щодня о 23:00 за допомогою декоратора `@Cron(CronExpression.EVERY_DAY_AT_11PM)`. Її мета — викликати метод `scheduleDeleteProcedure`, який видаляє не заплановані процедури на поточний день.

```
@Cron(CronExpression.EVERY_DAY_AT_11PM)
```

```
async handleCronDelete() {  
  try {  
    this.logger.log(  
      'proceduresDeleteSchedulerStrategy',  
      'schedule procedure deleted',  
    );  
    this.patientProcedureUsecaseProxy.getInstance().scheduleDeleteProcedure();  
  } catch (error) {  
    this.logger.error('proceduresDeleteScheduler', error);  
  }  
}
```

Метод `scheduleDeleteProcedure` видаляє всі процедури, які не були заплановані на поточний день.

```
async scheduleDeleteProcedure() {  
  try {  
    const today = new Date();  
    const startOfDay = new Date(today.setHours(0, 0, 0, 0));  
    const endOfDay = new Date(today.setHours(23, 59, 59, 999));  
  
    await this.patientProcedureRepository.deleteSchedule(  
      startOfDay,  
      endOfDay,  
    );  
  } catch (error) {  
    this.logger.error(  
      'proceduresDeleteSchedulerStrategy',  
      'Error deleting procedures',  
    );  
  }  
}
```

Для реалізації функціоналу надсилання щоденних повідомлень у системі підтримки та планування роботи пансіонату використовується Next.js, який

надає можливість писати серверний код, а також такі бібліотеки, як axios, dayjs, cron, і web-push. Цей функціонал дозволяє автоматично надсилати користувачам нагадування про заплановані процедури.

Функція `getTodayPatientProcedures` робить HTTP-запит до бекенду для отримання списку процедур, запланованих для пацієнта на сьогодні. Якщо запит успішний, функція повертає список процедур; у разі помилки (наприклад, якщо користувач не авторизований) повертається порожній список або кидається виключення.

```
const getTodayPatientProcedures = async (
  patientId: string
): Promise<ProcedureForPatient[]> => {
  try {
    const response = await axios.get<{ data: ProcedureForPatient[] }>(
      `${process.env.NEXT_PUBLIC_BACKEND_API_URL}/patientProcedure/getTodayPatientProcedures/${patientId}`
    )
    return response.data.data
  } catch (error: any) {
    if (error.response?.status === 403) {
      return []
    }
    throw new Error(error.response?.data.message || 'Error fetching procedures')
  }
}
```

Функція `sendDailyNotifications` відповідає за надсилання щоденних нагадувань користувачам про заплановані процедури. Вона отримує всі підписки з MongoDB, перевіряє їх валідність, та надсилає повідомлення користувачам, у яких є заплановані процедури на сьогодні. Якщо підписка недійсна або її термін дії закінчився, вона видаляється з бази даних.

```
const sendDailyNotifications = async () => {
  const notificationPromises = allSubscriptions.map(async subscription => {
    if (
      subscription.endpoint &&
      subscription.keys &&
      subscription.keys.p256dh &&
      subscription.keys.auth
    ) {
```

```

const pushSubscription = {
  endpoint: subscription.endpoint,
  keys: {
    p256dh: subscription.keys.p256dh,
    auth: subscription.keys.auth
  }
}

const userProcedures = await getTodayPatientProcedures(
  subscription.patientId
)

if (userProcedures.length > 0) {
  const appointmentTimes = userProcedures
    .map(procedure => procedure.appointmentTime)
    .join(', ')

  const message = {
    title: 'Нагадування',
    body: `Ви маєте ${userProcedures.length} запланованих процедур на сьогодні!
Час записів: ${appointmentTimes}.`
  }

  await Promise.all(notificationPromises)
}

```

Функція middleware відповідає за ініціалізацію Cron-завдань при першому запиті до сервера. Це гарантує, що завдання будуть запуснені автоматично, як тільки сервер почне обробляти запити. Ініціалізація відбувається шляхом відправки запиту до спеціального API /api/initCron, який запускає функцію для планування щоденних повідомлень.

```

if (!cronInitialized) {

  const host = request.headers.get('host')
  if (host) {
    const cronUrl = `http://${host}/api/initCron`
    try {
      const response = await fetch(cronUrl, {
        method: 'GET',
        headers: {
          'Content-Type': 'application/json'
        }
      })
      if (!response.ok) {
        const text = await response.text()
        console.error('Error response:', text)
      }
    }
  }
}

```

```

        throw new Error('Failed to initialize cron job')
    }
    const data = await response.json()
    cronInitialized = true
  } catch (error) {
    console.error(error)
  }
}
}

```

Ця функція також керує авторизацією користувачів, перевіряючи наявність токена оновлення (refreshToken). Якщо користувач намагається отримати доступ до сторінок авторизації (/auth), але вже має токен, його перенаправляють на домашню сторінку. Якщо користувач намагається отримати доступ до захищених сторінок без токена, його перенаправляють на сторінку авторизації.

```

const { url, cookies } = request

const refreshToken = cookies.get(EnumTokens.REFRESH)?.value
const isAuthPage = url.includes('/auth')
if (isAuthPage && refreshToken) {

    return NextResponse.redirect(new URL(DASHBOARD_PAGES.HOME, url))
}

if (isAuthPage) {
    return NextResponse.next()
}

if (!refreshToken) {
    return NextResponse.redirect(new URL('/auth', request.url))
}

return NextResponse.next()

```

Висновки до другого розділу

Під час проектування веб-орієнтованої системи підтримки та планування роботи пансіонату були визначені варіанти використання та проаналізовані всі можливі сценарії взаємодії користувачів із системою. Це дозволило створити модель, яка чітко відображає взаємодію між різними компонентами системи.

Основні цілі проекту включали оптимізацію роботи пансіонату та покращення якості надання рекреаційних послуг. Система була розроблена як веб додаток з інтуїтивно зрозумілим інтерфейсом, що полегшує користувачам взаємодію з системою.

Було спроектовано структуру бази даних, яка забезпечує ефективне зберігання та обробку даних про пацієнтів, медичний персонал, процедури та записи на процедури. На основі розроблених сутностей були створені таблиці у базі даних, а також діаграма, яка візуалізує ці зв'язки. Особливу увагу приділено забезпеченню безпеки конфіденційних даних за допомогою шифрування, що забезпечує захист інформації навіть у випадку компрометації бази даних.

Основні етапи проектування та реалізації системи включали аналіз функціональних вимог, розробку алгоритмів для управління записами на процедури, обробки медичних звітів та забезпечення безперебійної роботи системи. Було реалізовано функціонал для автоматичного планування процедур, обробки повідомлень між пацієнтами та медичним персоналом, а також забезпечення щоденних нагадувань про заплановані процедури.

У розділі було представлено детальний опис алгоритмів та методів, використаних для реалізації цього функціоналу, що забезпечує стабільну та надійну роботу системи.

WebServer: Виконується в середовищі Node.js і використовує Express.js для обробки серверних запитів. Веб-сервер містить три основні шари:

- Domain: Включає бізнес-логіку та моделі даних.
- Infrastructure: Забезпечує технічні деталі, такі як доступ до бази даних, зовнішні сервіси та інші компоненти.
- Usecases: Включає конкретні сценарії використання бізнес-логіки.

DataBaseServer: Система використовує дві бази даних для зберігання даних:

- PostgreSQL: Використовується для зберігання структурованих даних, таких як інформація про пацієнтів, лікарів, процедури та записи на процедури.
- MongoDB: Використовується для зберігання неструктурованих даних, таких як підписки на push-повідомлення та інша гнучка інформація.

Взаємодія між WebServer та DataBaseServer здійснюється за допомогою драйверів баз даних. Веб-сервер обробляє запити від клієнтів, взаємодіє з базами даних для отримання та зберігання інформації, а також надає відповіді клієнтам через HTTP-запити.

Ця структура забезпечує модульність, масштабованість та ефективне управління бізнес-логікою та інтерфейсом користувача, дозволяючи системі забезпечувати стабільну та надійну роботу.

Для забезпечення безперебійної роботи та легкої підтримки системи підтримки та планування роботи пансіонату використовується сучасний процес розгортання з використанням хостингу Railway та CI/CD (Continuous Integration/Continuous Deployment). Цей процес включає декілька ключових етапів, кожен з яких забезпечує надійність, продуктивність та безпеку додатку.

Railway є платформою для хостингу, яка забезпечує легкий та швидкий процес розгортання додатків. Railway автоматично управляє інфраструктурою, дозволяючи розробникам зосередитися на написанні коду. Основні переваги використання Railway включають:

- **Автоматичне масштабування:** Railway автоматично масштабує ресурси в залежності від навантаження на додаток.
- **Простота налаштування:** Інтеграція з GitHub та іншими репозиторіями забезпечує простий процес налаштування та розгортання.
- **Моніторинг та логування:** Railway надає інструменти для моніторингу продуктивності та логування, що дозволяє швидко виявляти та вирішувати проблеми.

Для автоматизації процесу розгортання використовується CI/CD з GitLab Runner. Цей процес включає наступні етапи:

Тестування додатку.

Кожен коміт у репозиторії GitLab запускає автоматичні тести, що перевіряють роботу додатку. Це забезпечує високу якість коду та запобігає помилкам на ранніх стадіях розробки.

Збірка додатку.

Після успішного проходження тестів, GitLab Runner створює збірку додатку. Цей процес включає компіляцію коду, збирання всіх необхідних залежностей та підготовку додатку до розгортання.

Збірка додатку гарантує, що всі зміни в коді були коректно інтегровані та готові до розгортання на продакшн.

Автоматизоване розгортання:

Після верифікації у тестовому середовищі, збірка автоматично розгортається на продакшн середовище.

GitLab Runner виконує розгортання на продакшн за допомогою інструментів управління конфігураціями, забезпечуючи безперервність бізнес-процесів[14].

Лістинг конфігурації CI/CD:

```
image: node:alpine

stages:
  - install_dependencies
  - lint
  - test
```

```

- build
- deploy

cache:
  paths:
    - node_modules/

install_dependencies:
  stage: install_dependencies
  script:
    - npm install
  artifacts:
    paths:
      - node_modules/

lint:
  stage: lint
  script:
    - npm run lint

test:
  stage: test
  script:
    - npm test
  coverage: '/All lines: (\d+\.\d+)%/'
  artifacts:
    paths:
      - coverage/

build:
  stage: build
  script:
    - npm run build

deploy:
  stage: deploy
  image: ubuntu
  only:
    - pushes
    - main
  script:
    - apt-get update && apt-get install -y curl
    - curl -fsSL https://railway.app/install.sh | sh
    - railway up --service=$RAILWAY_SERVICE_NAME -d

```

Інфраструктура розгортання додатку використовує Railway (рис. 3.2) як платформу для хостингу бекенду, фронтенду та бази даних.

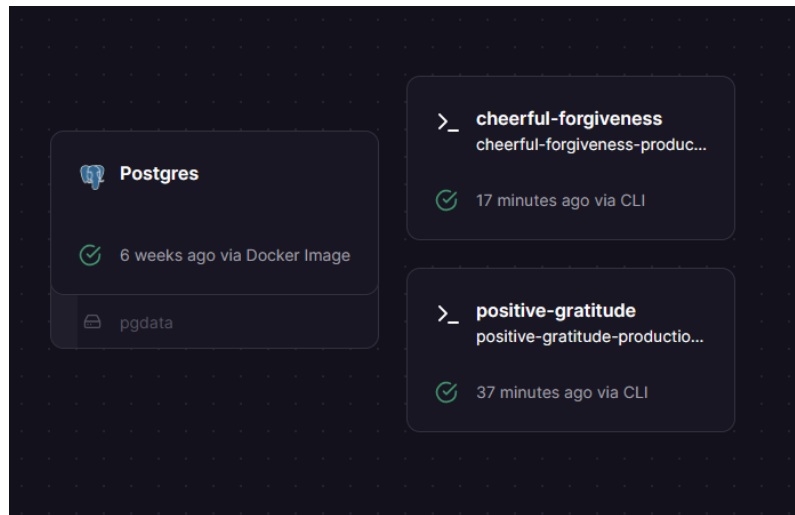


Рисунок 3.2 — Розгортання додатку на Railway

3.2 Структура інтерфейсу вебдодатку та порядок роботи

При відкритті вебдодатку користувач переходить на сторінку реєстрації, де він може зареєструватися в системі (рис 3.3).

Зареєструватися

Нікнейм:

Введіть нікнейм

Пароль:

Введіть пароль

Підтвердіть пароль:

Ім'я:

Прізвище:

Email:

[Зареєструватися](#)

[Увійти](#)

Рисунок 3.3 — Форма реєстрації

Для автентифікації користувач має перейти на форму автентифікації (рис. 3.4).

Увійти

Нікнейм:

Введіть ваш нікнейм

Будь-ласка введіть ваш нікнейм

Пароль:

Введіть ваш пароль

Будь-ласка введіть ваш пароль

Увійти

[Ще не зареєстровані? Зареєструватися](#)

Рисунок 3.4 — Форма автентифікації

Для форм автентифікації та реєстрації доступна валідація полів. У разі помилки відповідне поле підсвічується іншим кольором та з'являється повідомлення про помилку.

Після реєстрації, система перенаправляє користувача на форму автентифікації, де користувач входить в систему використовуючи свої облікові дані. Спочатку розглянемо роботу веб-додатку у ролі звичайного користувача, тобто пацієнта.

Головна сторінка веб-додатку для звичайного користувача відображає основну інформацію про додаток(рис. 3.5).



Рисунок 3.5 — Головна сторінка вебдодатку

На сторінці планування для пацієнта відображаються доступні для запису процедури на сьогодні та декілька днів вперед. Ця сторінка дозволяє пацієнту

ознайомитися з пропонованими процедурами, вибрати зручну дату і записатися на потрібну процедуру (рис. 3.6).

Верхня частина сторінки містить повідомлення про час оновлення записів на процедури: "Записи на процедури оновлюються кожен день о 00:00". Також виводиться поточний час, який допомагає пацієнту орієнтуватися в реальному часі.

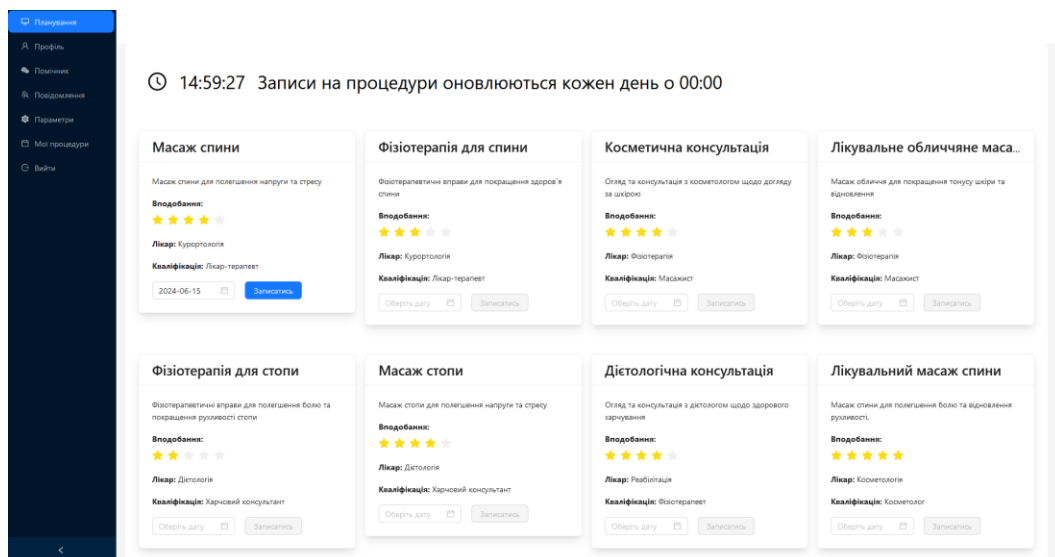
Процедури представлені у вигляді карток, кожна з яких містить наступну інформацію:

- назва процедури;
- опис процедури: короткий опис мети або особливостей процедури, наприклад, "Масаж спини для полегшення напруги та стресу";
- середня оцінка від користувачів: відображається у вигляді зірок, що показують середню оцінку процедури від інших пацієнтів;
- інформація про лікаря: спеціалізація та кваліфікація лікаря.

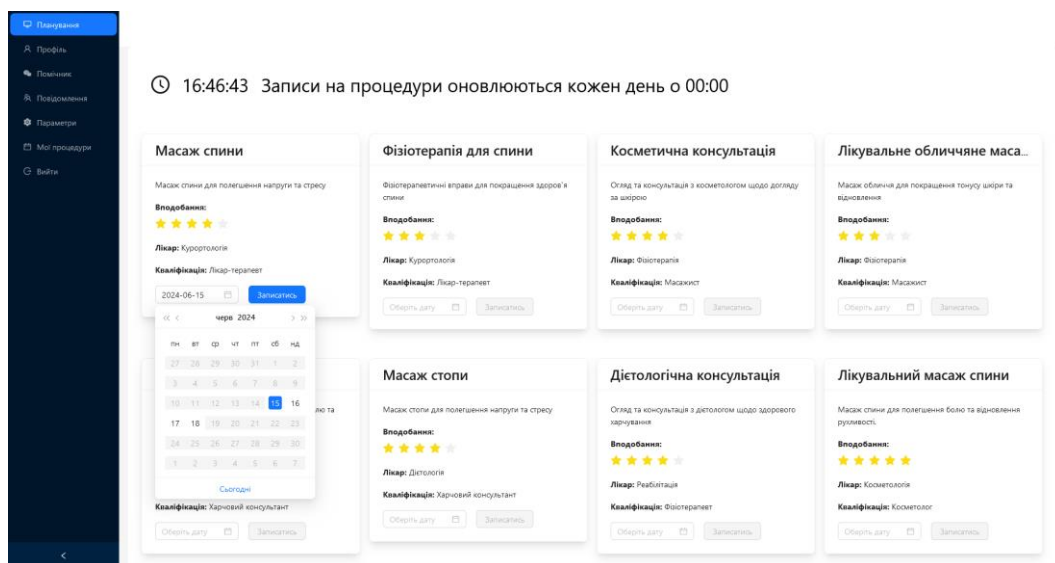
На кожній картці є кнопки "Обрати дату" і "Записатись", які дозволяють пацієнту вибрати зручний час для процедури і записатися на неї. Коли користувач обирає дату для запису, кнопка "Записатись" стає активною.

Ліва частина сторінки містить навігаційне меню, яке дозволяє користувачеві переходити між різними розділами додатку:

- планування — поточна сторінка для вибору процедур;
- профіль — перегляд та редагування особистої інформації;
- помічник — розділ з підказками та рекомендаціями;
- повідомлення — перегляд та надсилання повідомлень;
- параметри — налаштування системи;
- мої процедури — перегляд запланованих та минулих процедур;
- вийти — вихід з системи.



Коли користувач натискає кнопку "Записатись", відкривається календар, який дозволяє користувачу обрати потрібну йому дату запису (рис. 3.7). Система планує записи на 3 три дні, тому в календарі доступні відповідні дати.



Вікно запису на процедури, показує доступні на сьогоднішній день процедури (рис. 3.8). Вікно містить перелік можливих часів для запису. Часові слоти, що є заблокованими, не можуть бути обрані, оскільки час їхньої доступності вже минув. Активний часовий слот, що можна обрати для запису, виділено та він доступний для натискання.

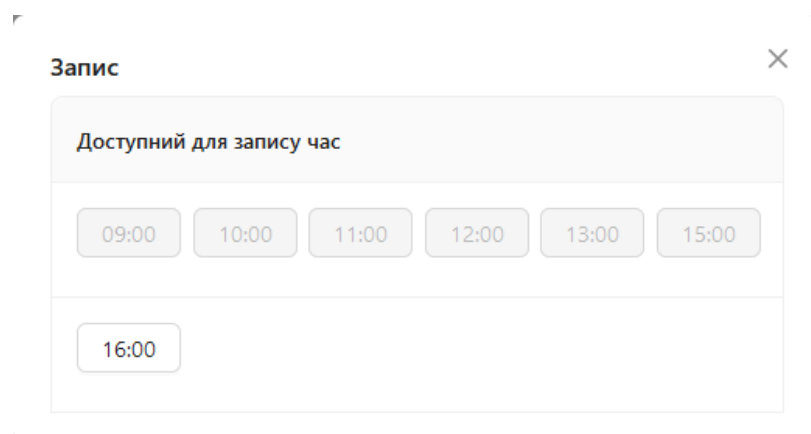


Рисунок 3.8 — Вибір часу для запису на процедуру

Після вибору дати запису користувачем, з'являється попереджувальне повідомлення(рис. 3.9). У цьому вікні йдеться про те, що якщо користувач уже записаний на інший час у той же день, його попередній запис буде відмінено. Повідомлення запитує користувача, чи він упевнений у своєму рішенні продовжити. У вікні є дві кнопки: "Скасувати" для відміни дії та повернення до попереднього екрану без зміни запису, і "Записатись" для підтвердження вибраного часу і заміни попереднього запису на новий, якщо він існує.

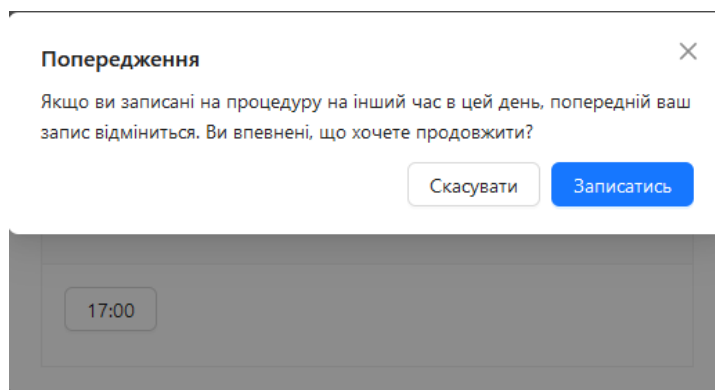
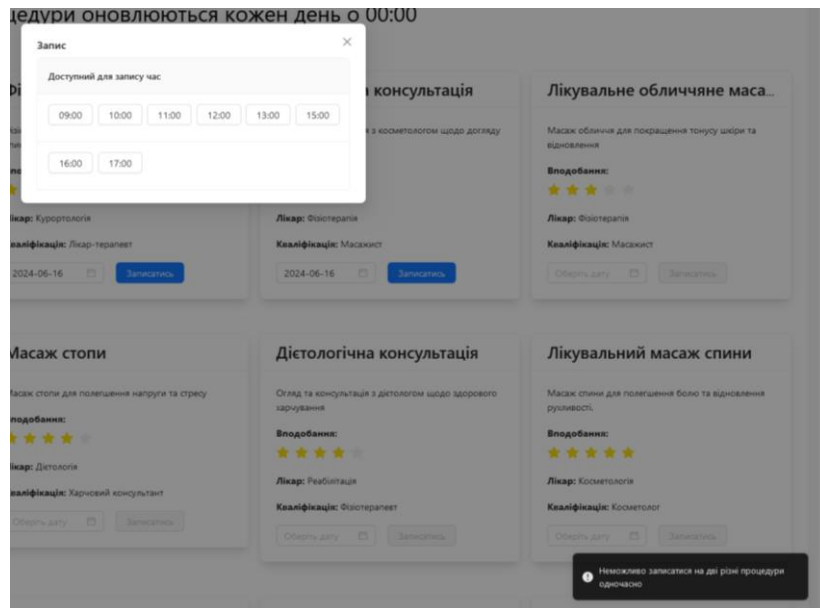
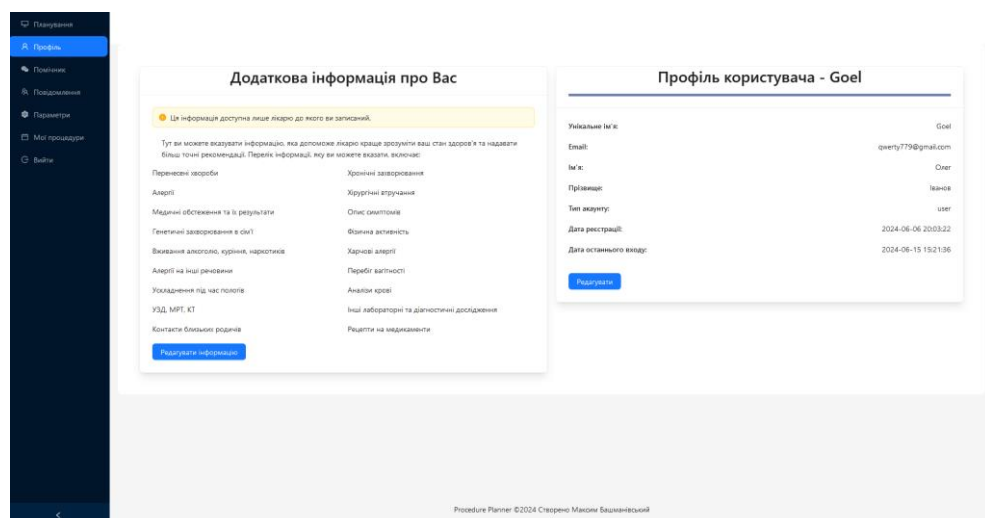


Рисунок 3.9 — Підтвердження запису

Користувач може мати запис на певний час на процедуру і намагатись записатись на той же час на іншу процедуру. В такому випадку після підтвердження запису пацієнт отримає повідомлення про помилку запису(рис. 3.10). В такій ситуації користувач має перейти на сторінку "Мої процедури" і відмінити вже існуючий запис.



Далі розглянемо сторінку профілю користувача. Сторінка профілю користувача містить детальну інформацію про зареєстрованого користувача та додаткову інформацію про пацієнта (рис. 3.11).



Для зручного введення та редагування особистої інформації пацієнта використовується редактор тексту (рис. 3.12). Він надає користувачам

Редагування інформації про пацієнта

H₁ H₂ Sans Serif ↕ Small ↕
B I U ☒ ☞ ⌵ ⏎ 🔍 ✖

Перенесені хвороби: Гостра респираторна вірусна інфекція (ГРВІ), грип.

Хронічні захворювання: Артеріальна гіпертензія, астма.

Алергії: Алергія на пилок, алергія на пил.

Хірургічні втручання: Апендектомія (видалення апендикса) у 2018 році.

Медичні обстеження та їх результати:

- Станний аналіз крові (2023 рік) - показники в нормі.
- Кардіограма (2022 рік) - без патологій.

Опис симптомів: Часті головні болі, слабкість в ногах після фізичних навантажень.

Генетичні захворювання в сім'ї: Діабет у матері, артеріальна гіпертензія у батька.

Фізична активність: Заняття плаванням 2 рази на тиждень, щоденні прогулянки.

Вживання алкоголю, куріння, наркотиків: Не вживаю алкоголь, не кую, не вживаю наркотики.

Харчові алергії: Алергія на арахіс, морепродукти.

Алергії на інші речовини: Алергія на деякі види миючих засобів.

Перебіг вагітності: Два вагітності, без ускладнень.

Онвити інформацію

Редагування особистої інформації (рис. 3.13). Ця форма призначена для редагування профілю користувача в веб-додатку. Вона дозволяє користувачу оновлювати свої особисті дані, такі як електронна адреса, ім'я та прізвище.

Профіль користувача - Goel

Email

Ім'я

Прізвище

СкасуватиЗберегти

Для управління записами на процедури була розроблена сторінка - "Мої процедури" (рис. 3.14). На цій сторінці користувач може управляти своїми

У верхній частині сторінки розташовані сьогоднішні процедури, які користувач може запланувати за допомогою функціоналу планування дня.

При створенні розкладу система враховує попередні записи пацієнта, надаючи пріоритет процедурам, які пацієнт проходив раніше. Це дозволяє забезпечити безперервність лікування та підтримку здоров'я, надаючи пацієнту процедури, з якими він вже знайомий і які можуть бути найбільш корисними для нього.

					<i>ПЗ.КР.Б-121-24-ПЗ</i>	Арк.
						71

×

Повторення процедур: Рекомендується повторювати масаж стопи двічі на тиждень для досягнення стійкого позитивного ефекту.

Домашній догляд: Рекомендується робити теплі ванночки для стоп з морською сіллю або трав'яними настоянками щовечора для зменшення набряків та розслаблення м'язів.

Фізична активність: Виконувати легкі фізичні вправи для стоп, такі як перекошування тенісного м'яча стопою або підйом на носки для підтримання м'язового тонуусу та покращення кровообігу.

Гідратація: Пити достатню кількість води протягом дня для покращення обміну речовин та зменшення набряклості.

Зручне взуття: Носити комфортне взуття з ортопедичною успішкою для зменшення навантаження на стопи та попередження розвитку плоскостопості.

У розділі майбутніх процедур відображаються записи на процедури, які заплановані на найближчі дні.

Минулі процедури



Таким чином, сторінка управління записами забезпечує зручний інтерфейс для керування всіма аспектами процедур, від планування до оцінки їх якості, що

сприяє підвищенню ефективності медичного обслуговування та задоволеності користувачів.

Сторінка "Повідомлення" забезпечує зручну комунікацію між користувачами (рис. 3.17). Тут користувачі можуть відправляти один одному короткі повідомлення, відповідати на них та видаляти.

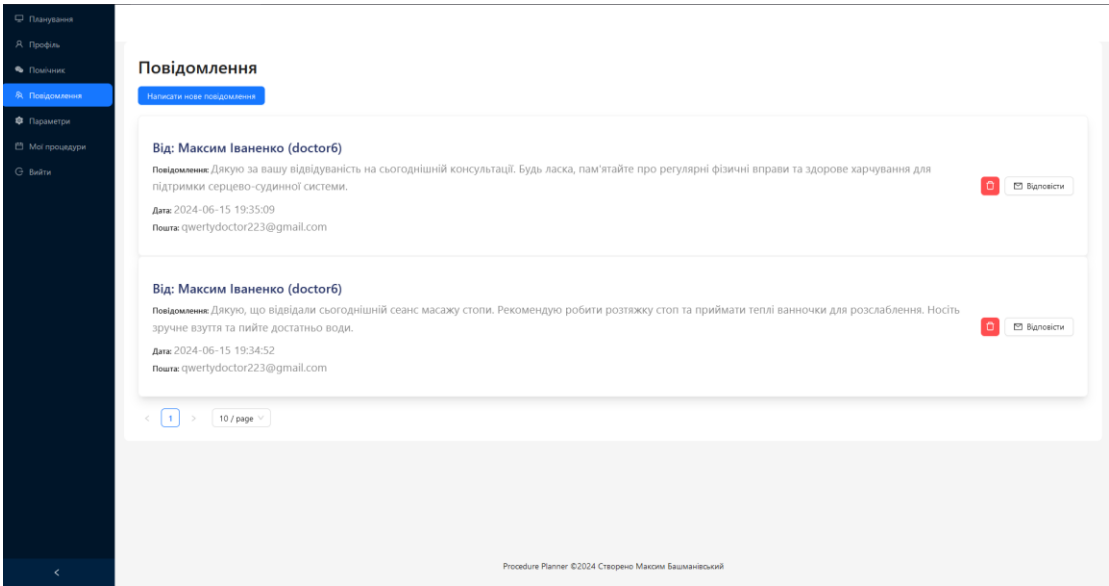


Рисунок 3.17 — Сторінка "Повідомлення"

На цій сторінці представлений список отриманих повідомлень, який включає відправника, вміст повідомлення, дату та час відправлення, а також електронну адресу відправника. Для кожного повідомлення доступні кнопки для видалення повідомлення та для відповіді на нього.

Функція надсилання нового повідомлення реалізована через кнопку "Написати нове повідомлення". При натисканні на цю кнопку відкривається форма для введення електронної адреси отримувача та тексту повідомлення.

Користувачі можуть легко відповісти на отримане повідомлення, натиснувши кнопку "Відповісти". Це автоматично заповнює форму з електронною адресою відправника, що значно спрощує процес відповіді.

Для видалення повідомлення користувачі можуть скористатися кнопкою "Видалити". Це дозволяє зберігати порядок у списку повідомлень та видаляти непотрібні чи вже неактуальні повідомлення.

Сторінка "Повідомлення" є важливою частиною додатку, забезпечуючи ефективний та простий спосіб комунікації між користувачами, сприяючи підтримці зв'язку та обміну важливою інформацією.

Сторінка "Параметри" надає користувачам можливість налаштувати отримання сповіщень про їхні процедури(рис. 3.18). Сповіщення надсилаються кожного дня о 8:30 ранку пацієнтам, які підписалися на ці сповіщення. У повідомленнях міститься розклад процедур на поточний день, що допомагає пацієнтам організувати свій час та не пропустити жодної процедури.

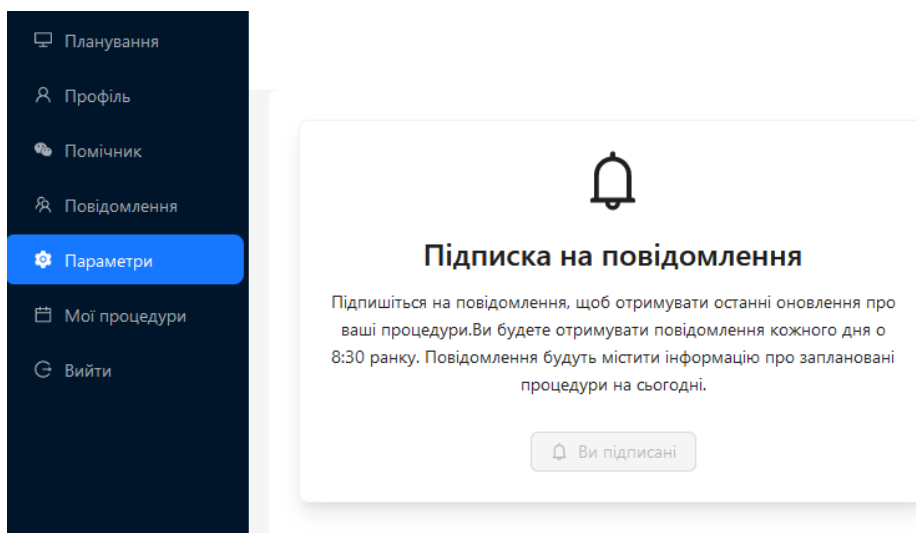


Рисунок 3.18 — Сторінка "Параметри"

На цій сторінці користувачі можуть легко підписатися на отримання сповіщень, натиснувши відповідну кнопку. Якщо користувач вже підписаний, система відображає відповідний статус.

Для реалізації цієї функції використовуються web push повідомлення. Web push повідомлення — це сповіщення, які надсилаються через веб-браузери навіть тоді, коли користувач не активний на сайті. Вони дозволяють інформувати користувачів у реальному часі про важливі події чи оновлення.

Після підписки на сповіщення, браузер користувача отримує маркер, який дозволяє серверу надсилати push повідомлення. Ці повідомлення з'являються у вигляді невеликих спливаючих вікон у куточку екрану, навіть якщо користувач не знаходиться на сайті, що робить їх дуже зручними для оперативного інформування.

Завдяки такій системі сповіщень пацієнти завжди в курсі свого розкладу процедур, що підвищує їхню організованість та знижує ризик пропуску важливих заходів.

На сторінці "Помічник" представлений інтерактивний чат з штучним інтелектом на базі ChatGPT 3.5(рис. 3.19). Цей інструмент дозволяє користувачам отримувати детальну інформацію про медичні процедури, задавати питання та отримувати відповіді в режимі реального часу. Інтерфейс чату простий у використанні, користувачі можуть вводити свої запити в текстове поле та натискати кнопку відправки.

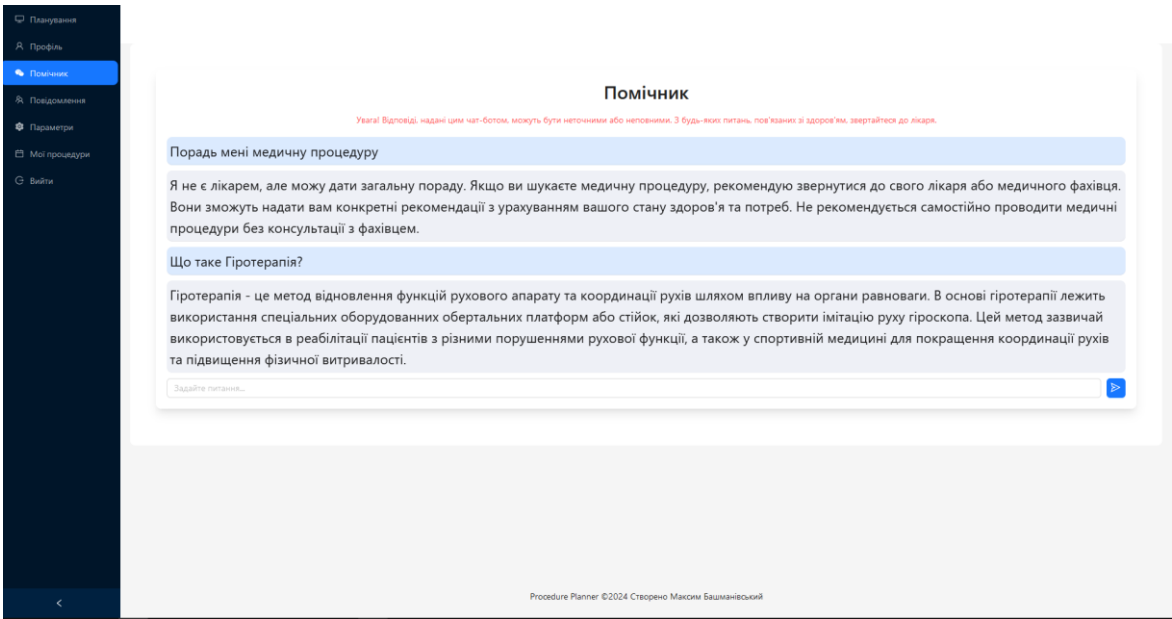


Рисунок 3.19 — Сторінка "Помічник"

Помічник надає корисні поради, пояснення та рекомендації щодо різних медичних процедур. Наприклад, можна дізнатися про новітні методи лікування, реабілітаційні процедури, а також отримати загальні поради щодо здорового способу життя. Важливо зазначити, що відповіді, надані помічником, мають інформативний характер і не замінюють консультацію з кваліфікованим медичним фахівцем.

Функціонал чату робить його незамінним інструментом для швидкого отримання інформації та підвищення обізнаності користувачів щодо їхнього здоров'я та доступних медичних послуг.

Наступна сторінка - "Процедури", доступна виключно для користувачів з роллю лікаря(рис 3.20). Ця сторінка надає можливість лікарям визначати, редагувати та видаляти процедури, які вони будуть проводити.

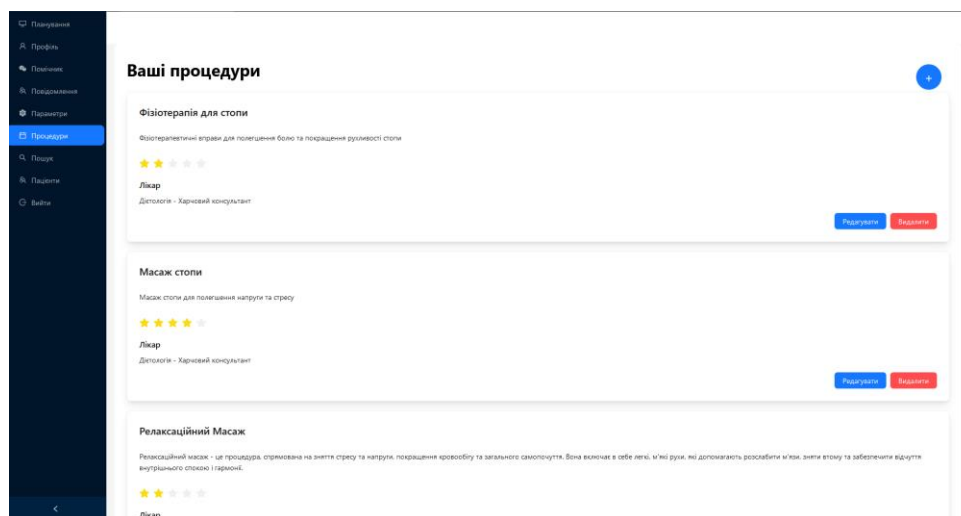


Рисунок 3.20 — Сторінка "Процедури"

На сторінці представлений перелік процедур, де кожна процедура має картку з назвою, описом, рейтингом, та інформацією про лікаря, який її проводить. Лікар може редагувати або видаляти процедури за допомогою відповідних кнопок.

При редагуванні лікар може оновлювати інформацію про процедуру, що дозволяє зберігати актуальні дані для пацієнтів. У разі видалення процедури, вона стає неактивною, і записи на наступні дні вже плануються без цієї процедури. Це забезпечує актуальність розкладу та дозволяє уникнути можливих непорозумінь.

Для додавання нової медичної процедури використовується форма(рис. 3.20). Важливим елементом форми є вибір акаунту лікаря, оскільки один обліковий запис може мати кілька акаунтів лікарів. Таким чином, процедура створюється для конкретного лікаря, що забезпечує точність і правильне розподілення медичних завдань.

Форма складається з кількох обов'язкових полів:

- виберіть акаунт лікаря - випадаючий список, де користувач має обрати конкретний акаунт лікаря для створення процедури;

- назва процедури - текстове поле для введення назви медичної процедури;
- опис процедури - текстове поле для введення детального опису процедури.

Рисунок 3.21 — Форма створення процедури

Після заповнення всіх необхідних полів, користувач може натиснути кнопку "Додати", щоб зберегти нову процедуру, або "Відмінити", щоб скасувати створення процедури.

Наступна сторінка, яку ми розглянемо, це сторінка "Пошук"(рис. 3.22). Вона призначена для лікарів, щоб вони могли ефективно шукати процедури за різними параметрами. Інтерфейс сторінки дозволяє вводити ключові слова, такі як назва процедури, опис процедури, ім'я чи прізвище пацієнта, або ж електронну адресу. Це забезпечує швидкий і зручний доступ до потрібної інформації.

На цій сторінці лікарі можуть бачити список процедур, які відповідають заданим критеріям пошуку. Кожна процедура представлена у вигляді картки, яка містить наступну інформацію:

- назва процедури;
- ім'я та прізвище пацієнта;
- електронна адреса пацієнта.

Також для кожної процедури передбачена можливість редагування звіту, що дозволяє лікарям зручно оновлювати інформацію про проведені процедури.

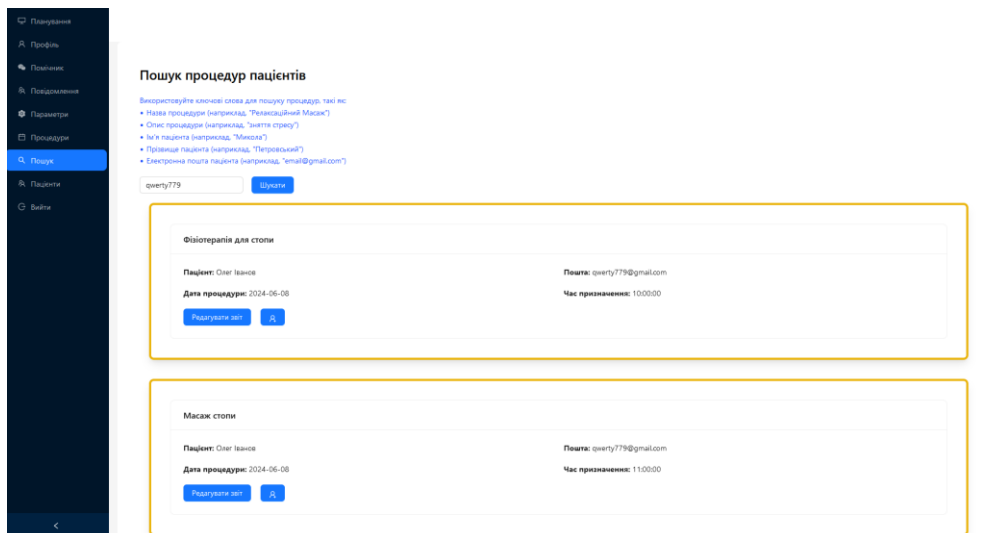


Рисунок 3.22 — Сторінка "Пошук"

Функція пагінації дозволяє лікарям переглядати результати пошуку, розділені на сторінки, що значно покращує зручність користування системою при великій кількості записів.

Наступна сторінка — "Пацієнти"(рис. 3.23). Вона дозволяє лікарям переглядати записи процедур пацієнтів, розділені на майбутні, минулі та сьогоднішні.

На цій сторінці відображаються всі заплановані майбутні процедури з можливістю редагування звітів та перегляду інформації про пацієнта(рис 3.23). Всі проведені процедури представлені з можливістю перегляду та редагування звітів.

Також відображаються всі процедури, заплановані на поточний день. Додатково є функція пагінації для зручного перегляду великої кількості записів та можливість перегляду особистої інформації пацієнтів. Ця сторінка забезпечує лікарям зручний доступ до необхідних даних для ефективного ведення медичної інформації.

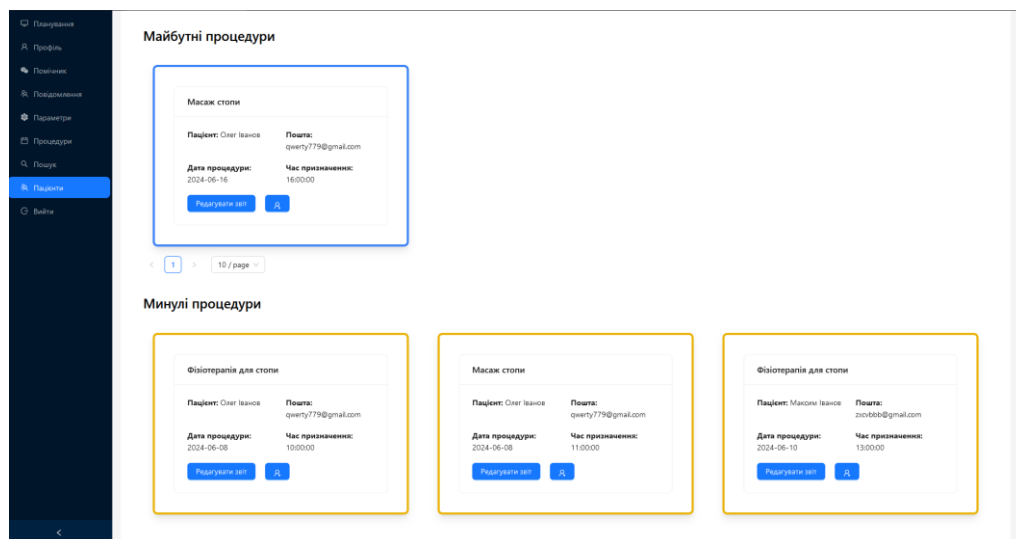


Рисунок 3.23 — Сторінка "Пацієнти"

На сторінці профілю лікаря відображається інформація про його акаунти лікарів(рис. 3.24).

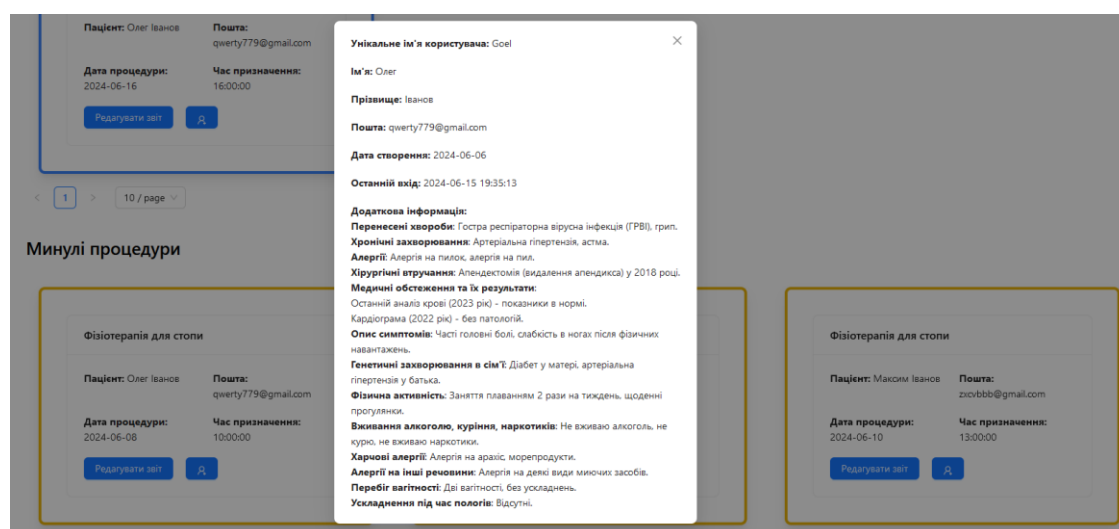


Рисунок 3.24 — Перегляд особистої інформації про пацієнта

Лікар може мати кілька акаунтів, кожен з яких містить деталі про кваліфікацію та спеціальність (рис. 3.25). Лікар має можливість оновлювати цю інформацію, натискаючи кнопку "Редагувати".

Поля кваліфікації та спеціальності є обов'язковими для заповнення, оскільки це основна інформація про лікаря, яка важлива для пацієнтів та інших медичних працівників. Цей розділ забезпечує легкий доступ до актуалізації професійних даних лікаря.

3.3 Тестування роботи програмного комплексу

Для забезпечення надійності та коректної роботи серверної частини додатку було створено юніт-тести для сутності акаунту. На прикладі коду, що перевіряє стратегію валідації JWT(рис 3.27), видно, як імітується поведінка репозиторію користувачів та сервісу шифрування.

У тесті перевіряється ситуація, коли користувач не знайдений у базі даних, і відповідно повертається значення null.

Використовуючи Jest[15] для мокування методів та перевірки результатів, забезпечується коректність логіки аутентифікації та підвищується стабільність всього додатку.



```
1 describe('Validation jwt strategy', () => {
2   it('should return null because user not found', async () => {
3     (adminUserRepo.getUserByUsername as jest.Mock).mockReturnValue(
4       Promise.resolve(null),
5     );
6     (bcryptService.compare as jest.Mock).mockReturnValue(
7       Promise.resolve(false),
8     );
9
10    expect(
11      await loginUseCases.validateUserForJWTStrategy('username'),
12    ).toEqual(null);
13  });
14 }
```

Рисунок 3.27 — Перевірка стратегії валідації JWT

Інтерфейс системи було розроблено з урахуванням різноманітних розмірів екрану, що дозволяє користувачам комфортно користуватися додатком як на комп'ютерах, так і на мобільних телефонах(рис. 3.28). Навігаційне меню та основні функції автоматично підлаштовуються під розмір екрану, забезпечуючи зручний доступ до всіх розділів і функцій системи.

На мобільних пристроях користувачі можуть швидко і легко переглядати свій розклад процедур, записуватись на нові процедури, читати повідомлення від лікарів та налаштовувати параметри свого облікового запису.

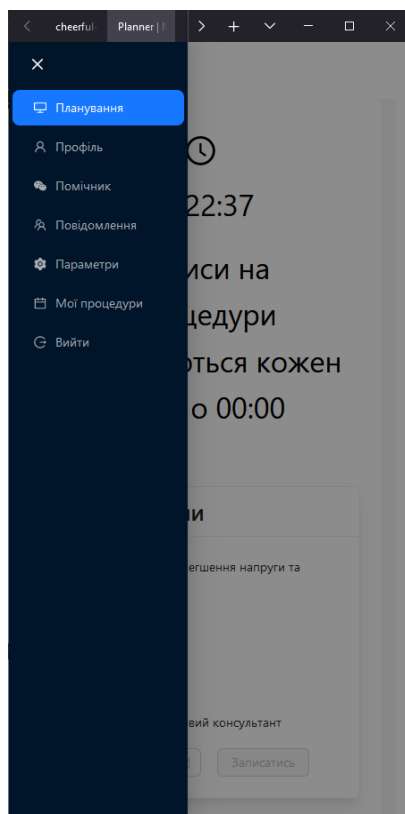


Рисунок 3.28 — Вигляд вебдодатку на мобільному пристрої

Завдяки адаптивному дизайну, всі елементи інтерфейсу залишаються чіткими та доступними, що робить користування додатком інтуїтивно зрозумілим і зручним, незалежно від пристрою. Мобільний додаток було протестовано мануально, щоб забезпечити його безперебійну роботу на різних платформах та пристроях. У процесі тестування перевірялися функціональність навігаційного меню, коректність відображення розкладу процедур, можливість запису на нові процедури, отримання та читання повідомлень від лікарів, а також налаштування параметрів облікового запису. Всі тести пройшли успішно, що підтверджує високу якість реалізації адаптивного дизайну та функціональності додатку.

Висновки до третього розділу

У даному розділі було детально розглянуто процес встановлення та налаштування параметрів системи, що включає порядок розгортання додатку та взаємодію його компонентів. Цей аналіз дозволив чітко визначити архітектуру

рішення та продемонструвати, як різні компоненти співпрацюють між собою для забезпечення належної роботи додатку.

Описана діаграма розгортання дала змогу візуально ілюструвати ці взаємозв'язки та процеси, що відбуваються в системі. Було представлено структуру інтерфейсу додатку, яка включає основні функції для користувачів, такі як реєстрація, створення заявок на обмін та додавання відгуків. Також було розглянуто можливості адміністратора, зокрема додавання, редагування та видалення активів, а також управління обмінами користувачів.

Розділ також висвітлив процес тестування додатку, де були наведені конкретні приклади застосування методів тестування, що забезпечує високу якість та надійність роботи системи. Завдяки детальному аналізу та впровадженню юніт-тестів для сутності акаунту серверної частини, було досягнуто стабільності та коректності логіки аутентифікації.

ВИСНОВКИ

Під час виконання кваліфікаційної роботи було спроектовано та реалізовано систему підтримки та планування роботи пансіонату. Спершу було проведено детальний аналіз предметної області та порівняння різних аналогів програмного продукту, що дозволило визначити основні вимоги до веб-додатку. На основі цього аналізу були сформовані завдання для створення веб-додатку, визначені ключові функціональні можливості та етапи їх реалізації.

У процесі проектування веб-додатку було виконано аналіз варіантів використання та розглянуто можливі сценарії взаємодії користувачів із системою. Спроектовано структуру бази даних для ефективного та безпечного зберігання та обробки даних про пацієнтів та лікарів. На основі розроблених сутностей створені таблиці у базі даних, а також діаграма бази даних для їх візуалізації.

Також була представлена структура інтерфейсу додатку системи підтримки та планування роботи пансіонату. Описано основні функції користувачів, такі як реєстрація, записи на процедури, налаштування надсилання сповіщень, а також можливості адміністратора, включаючи додавання, редагування та видалення лікарів, а також управління ролями користувачів. Користувачі можуть легко планувати свої процедури, отримувати необхідну інформацію та сповіщення, а також комунікувати з лікарями через додаток.

Було висвітлено процес тестування додатку, де наведено конкретні приклади застосування методів тестування. Це забезпечило високу якість та надійність роботи системи. Завдяки детальному аналізу та впровадженню юніт-тестів для сутності акаунту серверної частини досягнуто стабільності та коректності логіки аутентифікації.

Узагальнюючи, виконана робота демонструє системний та структурний підхід до розробки програмного забезпечення, де кожен етап має стратегічне значення для досягнення високих стандартів якості та ефективності.

СПИСОК ЛІТЕРАТУРИ

1. Башманівський М. О., Власенко О. В. Дослідження впровадження Dependency Injection в NestJS. Тези доповідей 14-ї міжнародної наукової конференції студентів та молодих вчених – "Сучасні Інформаційні Технології 2024", 16-17 травня 2024 року. [Електронний ресурс]. Режим доступу до ресурсу: http://ics_conf.tilda.ws/ukr
2. Nest.js [Електронний ресурс]. Режим доступу до ресурсу: <https://docs.nestjs.com/>
3. Angular [Електронний ресурс]. Режим доступу до ресурсу: <https://angular.dev/>
4. PostgreSQL [Електронний ресурс]. Режим доступу до ресурсу: <https://www.postgresql.org/>
5. NextJS [Електронний ресурс]. Режим доступу до ресурсу: <https://nextjs.org/>
6. TypeScript [Електронний ресурс]. Режим доступу до ресурсу: <https://www.typescriptlang.org/docs/>
7. Helsi [Електронний ресурс]. Режим доступу до ресурсу: <https://helsi.me/>
8. Medics [Електронний ресурс]. Режим доступу до ресурсу: <https://medics.ua/>
9. ELEKS Portal [Електронний ресурс]. Режим доступу до ресурсу: <https://portal-doctor.eleks.com/web/ml2zhytomyr/registration.html>
10. Роберт Сесіл Мартін [Електронний ресурс]. Режим доступу до ресурсу: <https://www.linkedin.com/in/robert-martin-7395b0/>
11. Clean Architecture A Craftsman Guide to Software Structure and Design [Електронний ресурс]. Режим доступу до ресурсу: <https://github.com/GunterMueller/Books-3/blob/master/Clean%20Architecture%20A%20Craftsman%20Guide%20to%20Software%20Structure%20and%20Design.pdf>
12. MongoDB [Електронний ресурс]. Режим доступу до ресурсу: <https://www.mongodb.com/>

13. Task Scheduling [Электронный ресурс]. Режим доступа до ресурсу:
<https://docs.nestjs.com/techniques/task-scheduling>

14. GitLab Runner [Электронный ресурс]. Режим доступа до ресурсу:
<https://docs.gitlab.com/runner/>

15. Jest [Электронный ресурс]. Режим доступа до ресурсу:
<https://jestjs.io/uk/>

16. TanStack Query [Электронный ресурс]. Режим доступа до ресурсу:
<https://tanstack.com/query/latest/docs/framework/react/overview>

17. Moment.js [Электронный ресурс]. Режим доступа до ресурсу:
<https://momentjs.com/docs/>

18. Ant Design 5.0 [Электронный ресурс]. Режим доступа до ресурсу:
<https://ant.design/>

19. Push API [Электронный ресурс]. Режим доступа до ресурсу:
https://developer.mozilla.org/en-US/docs/Web/API/Push_API

20. Sonner [Электронный ресурс]. Режим доступа до ресурсу:
<https://sonner.emilkowal.ski/>

21. Dayjs [Электронный ресурс]. Режим доступа до ресурсу:
<https://day.js.org/>

22. Axios [Электронный ресурс]. Режим доступа до ресурсу: <https://axios-http.com/uk/docs/intro>

23. React Hook Form [Электронный ресурс]. Режим доступа до ресурсу:
<https://react-hook-form.com/>

24. TypeORM [Электронный ресурс]. Режим доступа до ресурсу:
<https://typeorm.io/>

25. Bcrypt [Электронный ресурс]. Режим доступа до ресурсу:
<https://www.npmjs.com/package/bcrypt>

ДОДАТКИ

					ІІЗ.КР.Б-121-24-ІІЗ	Арк.
						87

```

//patientProcedure.usecases.ts
import { PatientProcedureRepository } from 'src/domain/repositories/patientProcedure.repository';
import {
  ConflictException,
  ForbiddenException,
  Logger,
  NotFoundException,
} from '@nestjs/common';
import { ProcedureRepository } from 'src/domain/repositories/procedure.repository';
import { PatientProcedureModel } from 'src/domain/model/patientProcedureModel';
import { DoctorRepository } from 'src/domain/repositories/doctor.repository.interface';
import { PatientRepository } from 'src/domain/repositories/patient.repository.interface';

export class addPatientProcedureUseCases {
  constructor(
    private readonly logger: Logger,
    private readonly procedureRepository: ProcedureRepository,
    private readonly doctorRepository: DoctorRepository,
    private readonly patientRepository: PatientRepository,
    private readonly patientProcedureRepository: PatientProcedureRepository,
  ) {}

  private ensureIsAdmin(accountType: string) {
    if (accountType !== 'admin') {
      throw new ForbiddenException(
        'Permission denied. Only admin can execute this operation.',
      );
    }
  }

  private ensureIsDoctor(accountType: string) {
    if (accountType !== 'doctor' && accountType !== 'admin') {
      throw new ForbiddenException(
        'Permission denied. Only doctors or admin can execute this operation.',
      );
    }
  }

  private ensureIsPatient(accountType: string) {
    if (accountType !== 'user') {
      throw new ForbiddenException(
        'Permission denied. Only user can execute this operation.',
      );
    }
  }
}

```

```

async add(
    data: PatientProcedureModel,
    account: any,
): Promise<PatientProcedureModel> {
    this.ensureIsAdmin(account.accountType);

    const procedure = await this.procedureRepository.getDoctorByProcedureId(
        data.procedureId,
    );

    if (!procedure || procedure.deleted) {
        throw new ForbiddenException('Procedure not found or has been deleted');
    }

    const patientProcedure = new PatientProcedureModel();
    patientProcedure.doctorId = procedure.doctor.id;
    patientProcedure.patientId = 28;
    patientProcedure.procedureId = data.procedureId;
    patientProcedure.procedureDate = new Date();
    patientProcedure.createdDate = new Date();
    patientProcedure.updatedDate = new Date();
    patientProcedure.appointmentTime = data.appointmentTime;
    patientProcedure.report = { report: 'report' };
    patientProcedure.rating = 0;

    const result =
        await this.patientProcedureRepository.createPatientProcedure(
            patientProcedure,
        );

    this.logger.log(
        'addPatientProcedureUseCases execute',
        'New patient procedure has been inserted',
    );

    return result;
}

async update(account: any, id: string): Promise<PatientProcedureModel> {
    this.ensureIsPatient(account.accountType);

    const patient = await this.patientRepository.getPatientByAccountId(
        account.id,

```

```

);
if (!patient) {
    throw new ForbiddenException('Пацієнт не знайдений');
}

const currentProcedure =
    await this.patientProcedureRepository.getPatientProcedureById(+id);
if (!currentProcedure) {
    throw new NotFoundException('Поточна процедура не знайдена');
}

const procedureDate = new Date(currentProcedure.procedureDate)
    .toISOString()
    .split('T')[0];

const existingProcedure =
    await this.patientProcedureRepository.getExistenceProcTodayByPatientId(
        patient.id,
        procedureDate,
        currentProcedure.procedureId,
    );

let inactiveProcedures = [];
let overlappingProcedures = [];

if (existingProcedure) {
    [inactiveProcedures, overlappingProcedures] = await Promise.all([
        this.patientProcedureRepository.getInactiveProceduresByDoctorAndTime(
            existingProcedure.doctorId,
            existingProcedure.appointmentTime,
            existingProcedure.procedureDate,
        ),
        this.patientProcedureRepository.getProceduresByDoctorAndTime(
            currentProcedure.doctorId,
            currentProcedure.appointmentTime,
            procedureDate,
        ),
    ]);
} else {
    overlappingProcedures =
        await this.patientProcedureRepository.getProceduresByDoctorAndTime(
            currentProcedure.doctorId,
            currentProcedure.appointmentTime,
            procedureDate,

```

```

    );
}

try {
    if (existingProcedure) {
        existingProcedure.updatedDate = new Date();
        existingProcedure.patientId = 28;
        existingProcedure.report = { report: '3BIT' };
        existingProcedure.rating = 0;
        await this.patientProcedureRepository.updatePatientProcedure(
            existingProcedure.id,
            existingProcedure,
        );

        await Promise.all(
            inactiveProcedures.map(async (procedure) => {
                if (procedure.patientId !== patient.id && procedure.id !== +id) {
                    procedure.patientId = 28;
                    procedure.report = { report: '3BIT' };
                    procedure.updatedDate = new Date();
                    await this.patientProcedureRepository.updatePatientProcedure(
                        procedure.id,
                        procedure,
                    );
                }
            })
        );
    }

    currentProcedure.updatedDate = new Date();
    currentProcedure.patientId = patient.id;
    currentProcedure.report = { report: '3BIT' };

    const result =
        await this.patientProcedureRepository.updatePatientProcedure(
            +id,
            currentProcedure,
        );

    await Promise.all(
        overlappingProcedures.map(async (procedure) => {
            if (procedure.patientId === 28 && procedure.id !== +id) {
                procedure.patientId = 29;
                procedure.report = { report: '3BIT' };
            }
        })
    );
}

```

```
procedure.updatedDate = new Date();
await this.patientProcedureRepository.updatePatientProcedure(
    procedure.id,
    procedure,
);
}
}),
);
```

```
this.logger.log(
    'updatePatientProcedureUseCases виконано',
    'Запис на процедуру був оновлений',
);

return result;
} catch (error) {
    this.logger.error('Error occurred:', error);
    if (error.code === '23505') {
        throw new ConflictException(
            'Неможливо записатися на дві різні процедури одночасно',
        );
    } else {
        throw error;
    }
}
}
```

```
async delete(id: string, account: { accountType: string }) {
  this.ensureIsAdmin(account.accountType);
  await this.patientProcedureRepository.deletePatientProcedure(+id);
  this.logger.log(
    'deletePatientProcedureUseCases execute',
    'Patient procedure have been deleted',
  );
}
```

```
async getById(id: string, account: any) {
  const patient = await this.patientRepository.getPatientByAccountId(
    account.id,
  );
  const patientProcedure =
    await this.patientProcedureRepository.getPatientProcedureById(+id);
  if (!patientProcedure) {
    throw new ForbiddenException('Patient procedure not found');
  }
}
```

```

    }
    if (patient.id !== patientProcedure.patientId) {
        throw new ForbiddenException('Permission denied');
    }
    this.logger.log(
        'getPatientProcedureUseCases execute',
        'Patient procedure have been found',
    );

    return patientProcedure;
}

async dailyScheduleProcedure() {
    try {
        const appointmentTime = [
            '9:00',
            '10:00',
            '11:00',
            '12:00',
            '13:00',
            '15:00',
            '16:00',
            '17:00',
        ];

        const procedures = await this.procedureRepository.getProcedures();
        const appointmentsPerProcedure = 8;
        const daysToSchedule = 4;

        for (let dayOffset = 0; dayOffset < daysToSchedule; dayOffset++) {
            const procedureDate = new Date();
            procedureDate.setDate(procedureDate.getDate() + dayOffset);

            const createProceduresPromises = [];

            for (const procedure of procedures) {
                for (let i = 0; i < appointmentsPerProcedure; i++) {
                    const appointmentTimeSlot = appointmentTime[i];

                    const createProcedurePromise = (async () => {
                        const existingProcedures =
                            await this.patientProcedureRepository.getExistenseProcedure(
                                procedure,
                                procedureDate,
                                appointmentTimeSlot,

```

```

    );

    if (!existingProcedures) {
        const patientProcedure = new PatientProcedureModel();
        patientProcedure.doctorId = procedure.doctorId;
        patientProcedure.patientId = 28;
        patientProcedure.procedureId = procedure.id;
        patientProcedure.procedureDate = procedureDate;
        patientProcedure.createdDate = new Date();
        patientProcedure.updatedDate = new Date();
        patientProcedure.appointmentTime = appointmentTimeSlot;
        patientProcedure.report = { report: 'Звіт' };
        patientProcedure.rating = 0;
        await this.patientProcedureRepository.createPatientProcedure(
            patientProcedure,
        );
    }
    })();

    createProceduresPromises.push(createProcedurePromise);
}
}
await Promise.all(createProceduresPromises);
}

this.logger.log(
    'DailyProceduresSchedulerStrategy',
    'Процедури були заплановані на 4 дні вперед',
);
} catch (error) {
    this.logger.error(
        'DailyProceduresSchedulerStrategy',
        'Error scheduling procedures',
    );
}
}

async scheduleDeleteProcedure() {
    try {
        const today = new Date();
        const startOfDay = new Date(today.setHours(0, 0, 0, 0));
        const endOfDay = new Date(today.setHours(23, 59, 59, 999));

        await this.patientProcedureRepository.deleteSchedule(

```

```

        startOfDay,
        endOfDay,
    );
} catch (error) {
    this.logger.error(
        'proceduresDeleteSchedulerStrategy',
        'Error deleting procedures',
    );
}
}

async getAll(account: any) {
    this.ensureIsAdmin(account.accountType);
    const patientProcedures =
        await this.patientProcedureRepository.getAllProcedures();
    this.logger.log(
        'getPatientProceduresUseCases execute',
        'All patient procedures have been fetched',
    );
    return patientProcedures;
}

async getMyProcedures(account: any) {
    const patient = await this.patientRepository.getPatientByAccountId(
        account.id,
    );
    if (!patient) {
        throw new ForbiddenException('Patient not found');
    }
    const patientProcedures =
        await this.patientProcedureRepository.getPatientProcedures(patient.id);
    if (!patientProcedures.length) {
        throw new ForbiddenException('Patient procedures not found');
    }
    if (patient.id !== patientProcedures[0].patientId) {
        throw new ForbiddenException('Permission denied');
    }
    this.logger.log(
        'getMyProceduresUseCases execute',
        'All patient procedures have been fetched',
    );
    return patientProcedures;
}

```

```

async getDoctorProcedures(
  account: any,
  category: 'past' | 'today' | 'future',
  page: number,
  pageSize: number,
) {
  this.ensureIsDoctor(account.accountType);
  const doctors = await this.doctorRepository.findByAccountId(account.id);

  if (!doctors || doctors.length === 0) {
    throw new ForbiddenException('Doctor not found');
  }

  let procedures = [];
  let total = 0;

  for (const doctor of doctors) {
    let result;
    if (category === 'past') {
      result =
        await this.patientProcedureRepository.getPastProceduresByDoctorId(
          +doctor.id,
          page,
          pageSize,
        );
    } else if (category === 'today') {
      result =
        await this.patientProcedureRepository.getTodayProceduresByDoctorId(
          +doctor.id,
          page,
          pageSize,
        );
    } else if (category === 'future') {
      result =
        await this.patientProcedureRepository.getFutureProceduresByDoctorId(
          +doctor.id,
          page,
          pageSize,
        );
    }

    if (result) {
      procedures = procedures.concat(result.data);
      total += result.total;
    }
  }
}

```

```

    }
}

this.logger.log(
    'getDoctorProceduresUseCases execute',
    'All doctor procedures have been fetched',
);

return { doctorProcedures: procedures, total };
}

async getReport(id: string, account: any) {
    const patientProcedure =
        await this.patientProcedureRepository.getPatientProcedureById(+id);
    if (!patientProcedure) {
        throw new ForbiddenException('Patient procedure not found');
    }
    if (account.accountType === 'doctor') {
        const doctor = await this.doctorRepository.findByAccountId(account.id);
        if (!doctor) {
            throw new ForbiddenException('Doctor not found or permission denied');
        }
    }
    if (account.accountType === 'user') {
        const patient = await this.patientRepository.getPatientByAccountId(
            account.id,
        );
        if (!patient) {
            throw new ForbiddenException('Patient not found or permission denied');
        }
        if (patient.id !== patientProcedure.patientId) {
            throw new ForbiddenException('Permission denied');
        }
    }

    this.logger.log(
        'getReportUseCases execute',
        'Patient procedure report has been fetched',
    );
    return patientProcedure.report;
}

async searchProcedures(
    keyword: string,

```

```

    account: any,
  ): Promise<{ data: any[]; total: number }> {
    this.ensureIsDoctor(account.accountType);

    let result = [];
    let total = 0;

    if (account.accountType === 'user') {
      const patient = await this.patientRepository.getPatientByAccountId(
        account.id,
      );
      if (!patient) throw new ForbiddenException('Patient not found');

      const searchResult =
        await this.patientProcedureRepository.searchProceduresByKeyword(
          keyword,
          +patient.id,
          'patient',
        );
      result = searchResult[0];
      total = searchResult[1];
    } else if (account.accountType === 'doctor') {
      const doctors = await this.doctorRepository.findByAccountId(account.id);
      if (!doctors || doctors.length === 0)
        throw new ForbiddenException('Doctor not found');

      for (const doctor of doctors) {
        const searchResult =
          await this.patientProcedureRepository.searchProceduresByKeyword(
            keyword,
            +doctor.id,
            'doctor',
          );
        result = result.concat(searchResult[0]);
        total += searchResult[1];
      }
    } else {
      throw new ForbiddenException('Invalid account type');
    }

    this.logger.log(
      'searchProceduresUseCases execute',
      'Procedures have been fetched by keyword',
    );
  }

```

```

    return { data: result, total };
}

async rateProcedure(id: string, rating: number, account: any) {
    const patientProcedure =
        await this.patientProcedureRepository.getPatientProcedureById(+id);
    if (!patientProcedure) {
        throw new ForbiddenException('Patient procedure not found');
    }
    const patient = await this.patientRepository.getPatientByAccountId(
        account.id,
    );
    if (!patient) {
        throw new ForbiddenException('Patient not found');
    }
    if (patient.id !== patientProcedure.patientId) {
        throw new ForbiddenException('Permission denied');
    }
    patientProcedure.rating = rating;
    patientProcedure.updatedDate = new Date();
    const result = await this.patientProcedureRepository.updatePatientProcedure(
        +id,
        patientProcedure,
    );
    this.logger.log(
        'rateProcedureUseCases execute',
        'Patient procedure have been rated',
    );
    return result;
}

async reportProcedure(id: string, report: object, account: any) {
    const patientProcedure =
        await this.patientProcedureRepository.getPatientProcedureById(+id);
    if (!patientProcedure) {
        throw new ForbiddenException('Пацієнт не знайдений');
    }
    const doctors = await this.doctorRepository.findByAccountId(account.id);
    if (!doctors || doctors.length === 0) {
        throw new ForbiddenException('Лікар не знайдений');
    }

    const isAuthorizedDoctor = doctors.some(

```

```

        (doctor: any) => doctor.id === patientProcedure.doctorId,
    );
    if (!isAuthorizedDoctor) {
        throw new ForbiddenException('Доступ заборонено');
    }

    patientProcedure.report = report;
    patientProcedure.updatedDate = new Date();
    const result = await this.patientProcedureRepository.updatePatientProcedure(
        +id,
        patientProcedure,
    );
    this.logger.log(
        'reportProcedureUseCases виконано',
        'Запис на процедуру був оновлений',
    );
    return result;
}

async getProceduresByDate(procedureId: number, date: string) {
    const patientProcedures =
        await this.patientProcedureRepository.getPatientProceduresTimesByDate(
            date,
            procedureId,
        );
    console.log('patientProcedures', patientProcedures);

    if (!patientProcedures.length) {
        throw new ForbiddenException('Procedures not found');
    }

    const timesWithoutSeconds = patientProcedures.map((procedure) => ({
        id: procedure.id,
        appointmentTime: procedure.appointmentTime.slice(0, -3),
        appointmentDate: procedure.appointmentDate,
    }));
    this.logger.log(
        'getProceduresByDate execute',
        'Procedures times have been fetched',
    );

    return timesWithoutSeconds;
}

```

```

async cancelProcedure(id: string, account: any) {
  const patientProcedure =
    await this.patientProcedureRepository.getPatientProcedureById(+id);
  if (!patientProcedure) {
    throw new ForbiddenException('Запис не знайдений');
  }

  const patient = await this.patientRepository.getPatientByAccountId(
    account.id,
  );
  if (!patient) {
    throw new ForbiddenException('Пацієнт не знайдений');
  }

  if (patient.id !== patientProcedure.patientId) {
    throw new ForbiddenException('Доступ заборонено');
  }

  patientProcedure.updatedDate = new Date();
  patientProcedure.report = { report: 'Звіт' };
  patientProcedure.patientId = 28;
  await this.patientProcedureRepository.updatePatientProcedure(
    +id,
    patientProcedure,
  );

  const inactiveProcedures =
    await this.patientProcedureRepository.getInactiveProceduresByDoctorAndTime(
      patientProcedure.doctorId,
      patientProcedure.appointmentTime,
      patientProcedure.procedureDate,
    );

  for (const procedure of inactiveProcedures) {
    if (procedure.patientId !== patient.id && procedure.id !== +id) {
      procedure.patientId = 28;
      procedure.report = { report: 'Звіт' };
      procedure.updatedDate = new Date();
      await this.patientProcedureRepository.updatePatientProcedure(
        procedure.id,
        procedure,
      );
    }
  }
}

```

```

this.logger.log(
    'cancelProcedureUseCases виконано',
    'Запис на процедуру був скасований',
);
}

async scheduleMyDay(account: any) {
    this.ensureIsPatient(account.accountType);

    const appointmentTimes = [
        '9:00',
        '10:00',
        '11:00',
        '12:00',
        '13:00',
        '15:00',
        '16:00',
        '17:00',
    ];

    const patient = await this.patientRepository.getPatientByAccountId(
        account.id,
    );
    if (!patient) {
        throw new ForbiddenException('Patient not found');
    }

    const today = new Date();
    const todayStr = today.toISOString().split('T')[0];

    let availableProcedures =
        await this.patientProcedureRepository.getAvailableProceduresByDay(
            todayStr,
        );
    availableProcedures.sort((a, b) => b.rating - a.rating);

    const userPreferences =
        await this.patientProcedureRepository.getUserPreferences(patient.id);
    const scheduledProcedures = [];

    for (const time of appointmentTimes) {
        if (availableProcedures.length === 0) break;
    }
}

```

```

let preferredProcedures = availableProcedures.filter((proc) =>
    userPreferences.includes(proc.procedure.id),
);

if (preferredProcedures.length === 0) {
    preferredProcedures = availableProcedures;
}

const procedureToSchedule = preferredProcedures.shift();

if (!procedureToSchedule) {
    continue;
}

scheduledProcedures.push({
    id: procedureToSchedule.id,
    patientId: patient.id,
    procedureId: procedureToSchedule.procedure.id,
    appointmentTime: time,
    procedureDate: today,
    doctorId: procedureToSchedule.doctorId,
});

availableProcedures = availableProcedures.filter(
    (proc) => proc.procedure.id !== procedureToSchedule.procedure.id,
);
}

for (const proc of scheduledProcedures) {
    const existingProcedure =
        await this.patientProcedureRepository.getExistenceProcedure(
            28,
            proc.procedureDate,
            proc.appointmentTime,
        );

    if (existingProcedure) {
        const patientProcedure = new PatientProcedureModel();
        patientProcedure.id = proc.id;
        patientProcedure.doctorId = proc.doctorId;
        patientProcedure.patientId = proc.patientId;
        patientProcedure.procedureId = proc.procedureId;
        patientProcedure.procedureDate = proc.procedureDate;
        patientProcedure.createdDate = proc.createdDate;
    }
}

```

```

patientProcedure.updatedDate = new Date();
patientProcedure.appointmentTime = proc.appointmentTime;
patientProcedure.report = { report: 'Ваш звіт' };
patientProcedure.rating = 0;

await this.patientProcedureRepository.updatePatientProcedure(
    patientProcedure.id,
    patientProcedure,
);
} else {
    console.log(
        `Procedure already exists for patient ${proc.patientId} at ${proc.appointmentTime}`,
    );
}
}

this.logger.log(
    'scheduleMyDay',
    `User ${patient.id} scheduled for procedures`,
);
}

async getPatientInfoByProcedureId(patientProcedureId: number, account: any) {
    const patientProcedureDoctor =
        await this.patientProcedureRepository.getPatientProcedureById(
            patientProcedureId,
        );
    if (!patientProcedureDoctor) {
        throw new NotFoundException('Doctor for the patient procedure not found');
    }

    const doctors = await this.doctorRepository.findByAccountId(account.id);
    if (!doctors || doctors.length === 0) {
        throw new NotFoundException('Doctor not found');
    }

    const hasPermission = doctors.some(
        (doctor) => doctor.id === patientProcedureDoctor.doctorId,
    );
    if (!hasPermission) {
        throw new ForbiddenException('Permission denied');
    }

    const patientInfo = await this.patientRepository.getPatientById(

```

```

        +patientProcedureDoctor.patientId,
    );
    if (!patientInfo) {
        throw new NotFoundException('Patient not found');
    }

    this.logger.log(
        'getPatientInfoByProcedureId',
        `Patient ${patientInfo.id} info has been fetched`,
    );

    return patientInfo;
}
async getTodayPatientProcedures(patientId: number) {
    const patientProcedures =
        await this.patientProcedureRepository.getTodayPatientProcedures(
            patientId,
        );
    return patientProcedures;
}
}

```