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

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
“__” _____2024 р.

Дипломний проєкт

на здобуття ступеня бакалавра
за освітньо-професійною програмою “Інженерія програмного забезпечення комп’ютерних систем”
спеціальності 121 “Інженерія програмного забезпечення”

на тему: Система для спільного використання моделей штучного інтелекту для діагностування захворювань (серверна частина)

Виконав : студент __4__ курсу, групи _ІП-04_
(шифр групи)
_____Храпко Василь Андрійович_____
(прізвище, ім’я, по батькові) (підпис)

Керівник _____асистент Ковальчук О. М._____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) доцент, к.т.н., Волокита А. М._____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент доцент кафедри ІСТ, доц., к.т.н., Шимкович В. М._____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

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

Київ – 2024 р.

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

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
“__” _____2024 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Храпка Василя Андрійовича

1. Тема проєкту *Система для спільного використання моделей штучного інтелекту для діагностування захворювань (серверна частина)*

керівник проєкту Ковальчук Олександр Миронович, асистент
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від 27 травня 2024 року №2112-с

2. Термін здачі студентом закінченого проєкту *6 червня 2024 р.*

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

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

Розділ 1. Аналіз предметної області та огляд існуючих систем-аналогів.

Розділ 2. Огляд підходів та технологій для розробки системи.

Розділ 3. Розробка системи.

Розділ 4. Розгортання та аналіз розробленої системи

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

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

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Волокита А. М.		

7. Дата видачі завдання «17» жовтня 2023 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми роботи*	*04.12.2023-08.12.2023*	
2.	*Вивчення та аналіз завдання*	*02.01.2024-26.01.2024*	
3.	*Розробка архітектури та загальної структури системи*	*05.02.2024-23.02.2024*	
4.	*Розробка структур окремих частин системи*	*04.03.2024-19.03.2024*	
5.	*Програмна реалізація системи*	*01.04.2024-19.04.2024*	
6.	*Оформлення пояснювальної записки*	*01.05.2024-24.05.2024*	
7.	*Захист програмного продукту*	*27.05.2024*	
8.	*Передзахист*	*06.06.2024*	
9.	*Захист*	*20.06.2024*	

Студент-дипломник _____ Василь ХРАПКО
(підпис)

Керівник проєкту _____ Олександр КОВАЛЬЧУК
(підпис)

АНОТАЦІЯ

У бакалаврському дипломному проєкті реалізовано серверну частину системи для спільного використання моделей штучного інтелекту для діагностування захворювань за зображеннями.

Система забезпечує можливість користувачам з роллю пацієнта виконувати діагностику шляхом завантаження зображення для аналізу доступними моделями штучного інтелекту та переглядати результати діагностики. Також система надає користувачам з роллю адміністратора доступ до керування моделями штучного інтелекту: завантаження моделей штучного інтелекту, керування версіями моделей, активації та деактивації використання моделей.

Для взаємодії клієнтських застосунків із сервером розроблено серверний API використовуючи мову програмування TypeScript, серверну платформу виконання JavaScript-коду Node.js та фреймворк Nest.js. Для виконання діагностики захворювань за зображеннями використано моделі штучного інтелекту, розроблені за допомогою бібліотеки машинного навчання TensorFlow, та бібліотеку tensorflow для роботи з моделями. Діагностика виконується в серверному додатку, розробленому на мові програмування Python та фреймворку FastAPI. Комунікація сервера для обробки запитів і серверів для виконання діагностики реалізована за допомогою брокера повідомлень RabbitMQ. Для збереження даних використовується реляційна база даних PostgreSQL, для файлів – об'єктне сховище MinIO.

Ключові слова: моделі штучного інтелекту, діагностування захворювань за зображеннями, TensorFlow, TypeScript, Node.js, Nest.js, Python, FastAPI, RabbitMQ, PostgreSQL, MinIO.

ANNOTATION

In this project for a Bachelor's Degree, server-side of system for common application of artificial intelligence models for making the diagnosis of diseases by the images has been implemented.

The system provides an opportunity for users (acting as patients) to perform diagnostics by means of image uploading for analysis, applying available models of artificial intelligence and to review diagnostics' findings. Furthermore, the system gives an access to operate the models of artificial intelligence for users performing administrator role, that is uploading of artificial intelligence models, operating of models' version, activation and deactivation of models' application.

For the interaction of the clients' applications with server, server API has been developed using TypeScript programming language, Node.js server platform for running JavaScript-code as well as Nest.js. framework. To diagnose diseases using images, models of artificial intelligence have been employed, devised with the help of TensorFlow computer-assisted learning library and tensorflow library to work with models. The diagnostics is performed in server application, developed on the basis of Python programming language and FastAPI framework. Communication between requests' processing server and servers for diagnosing has been implemented with the help of RabbitMQ messages broker. To preserve the data, PostgreSQL relational database is used, whereas MinIO object warehouse is employed for files.

Key words: models of artificial intelligence, diagnosis of diseases by means of image uploading, TensorFlow, TypeScript, Node.js, Nest.js, Python, FastAPI, RabbitMQ, PostgreSQL, MinIO.

справки	Формат	*Значення*	*Найменування*	К-сть. листів	№ екземпляра	*Додаток*
			Документація загальна			
			Знову розроблена			
	А4	***ІАЛЦ.467200.002 ТЗ***	Система для спільного	4		
			використання моделей			
			штучного інтелекту			
			для діагностування			
			захворювань			
			(серверна частина)			
			Технічне завдання			
	А4	***ІАЛЦ.467200.003 ПЗ***	Система для спільного	95		
			використання моделей			
			штучного інтелекту			
			для діагностування			
			захворювань			
			(серверна частина)			
			Пояснювальна записка			
	А4	***ІАЛЦ.467200.004 Д1***	Схема компонентів системи	1		
			(структурна схема)			
	А4	***ІАЛЦ.4672008.005 Д2***	Діаграма бази даних	1		
			(функціональна схема)			
	А4	***ІАЛЦ.467200.006 Д3***	Алгоритм дій програмного	1		
			забезпечення			
			(принципова схема)			
	А4	***ІАЛЦ.467200.007 Д4***	Текст програмного коду	43		

					ІАЛЦ.467200.001 ОА
Зм	***Лист***	***№ докум.***	***Підп***	***Дата***	
Розроб.		Храпко В. А.			**Система для спільного використання моделей штучного інтелекту для діагностування захворювань (серверна частина)** Опис альбому
Перев.		Ковальчук О. М.			

Літ.	Аркуш	Аркушів
	1	1

КПІ ім. Ігоря Сікорського, ФІОТ, ІП-04

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Система для спільного використання моделей штучного інтелекту для діагностування захворювань (серверна частина)*»

Київ – 2024

ЗМІСТ

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання поширюється на розробку серверної частини системи для спільного використання моделей штучного інтелекту для діагностування захворювань за зображеннями, підтримку та покращення розробленої системи.

Область застосування системи – забезпечення доступу до попередньої медичної діагностики всім користувачам-пацієнтам незалежно від місця їх перебування для створення доступного та ефективного інструменту попереднього діагностування хвороб.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки даної системи є завдання для виконання роботи кваліфікаційно-освітнього рівня «бакалавр інженерії програмного забезпечення», який був затверджений факультетом “Інформатики та обчислювальної техніки” кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний інститут ім. Ігоря Сікорського».

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою та призначенням даного проєкту є розробка серверної частини системи для спільного використання моделей штучного інтелекту для діагностування захворювань за зображеннями для забезпечення можливості пацієнтів виконувати попередню діагностику хвороб, що покращить доступність аналізу для виявлення хвороб.

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки даного дипломного проєкту є офіційні документації, публікації, науково-технічна література та статті в мережі Інтернет на тему

дипломного проєкту, а також на теми розробки серверних додатків, проєктування та розробка баз даних.

5 ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

Розроблена система має виконувати такі вимоги:

- Забезпечення можливості користувачам з роллю пацієнтів виконання діагностики хвороб шляхом завантаження зображення для аналізу доступними моделями штучного інтелекту
- Забезпечення доступу користувачів з роллю пацієнтів до власних результатів діагностики.
- Забезпечення користувачам з роллю адміністраторів можливості керування моделями штучного інтелекту: завантаження моделей штучного інтелекту, керування версіями моделей, активації та деактивації використання моделей
- Забезпечення цілісності даних в базі даних.
- Надання зрозумілої та повної документації методів взаємодії клієнтів з серверним API.

5.2. Вимоги до програмного забезпечення

- ОС Windows, Mac чи Linux.
- IDE Visual Studio версії 1.17 або вище.
- Програмне забезпечення для системи контейнеризації Docker 20 версії або вище.

5.3. Вимоги до апаратної частини

- ЦП не менше ніж з 2 ядрами.
- RAM не менше ніж 2 ГБ.
- ROM не менше ніж 12 ГБ.

6 ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	04.12.2023-08.12.2023
Вивчення та аналіз завдання	02.01.2024-26.01.2024
Розробка архітектури та загальної структури системи	05.02.2024-23.02.2024
Розробка структур окремих частин системи	04.03.2024-19.03.2024
Програмна реалізація системи	01.04.2024-19.04.2024
Виправлення помилок	22.04.2024-30.04.2024
Оформлення пояснювальної записки	01.05.2024-24.05.2024

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Система для спільного використання моделей штучного інтелекту для діагностування захворювань (серверна частина)*»

Київ – 2024

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

ШІ	Штучний інтелект
AI	(Artificial Intelligence) Штучний інтелект
ПЗ	Програмне забезпечення
CAD	(Computer-Aided Diagnostics) Комп’ютерна медична діагностика
КТ	Комп'ютерна томографія
BE	(Back end) Серверна частина програмного забезпечення
API	(Application programming interface) Програмний інтерфейс взаємодії
БД	База даних
СУБД	Система управління базами даних
SQL	(Structured Query Language) Декларативна мова програмування для взаємодії користувача з реляційними базами даних
NoSQL	(non SQL) База даних, що забезпечує механізм зберігання та видобування даних відмінний від підходу таблиць-відношень в реляційних базах даних.
HTTP	(HyperText Transfer Protocol) Протокол передачі гіпертекстових документів в комп’ютерних мережах
HTTPS	(HyperText Transfer Protocol Secure) Безпечний протокол передачі гіпертекстових документів в комп’ютерних мережах
CORS	(Cross-Origin Resource Sharing) Механізм спільного використання ресурсів з різних джерел
ACID	(Atomicity, Consistency, Isolation, Durability) Набір властивостей, що гарантують надійну роботу транзакцій бази даних (атомарність, узгодженість, ізольованість, довговічність)

REST	(Representational State Transfer) Набір підходів до будування архітектури взаємодії з сервером в комп’ютерних мережах.
TS	(TypeScript) Мова програмування TypeScript.
CPU	(Central processing unit) Центральний процесор
RAM	(Random Access Memory) Оперативна пам'ять
IP	(Internet Protocol) Протокол мережевого рівня для передавання датаграм між мережами
TCP	(Transmission Control Protocol) Протокол транспортного рівня призначений для керування передаванням даних у комп'ютерних мережах.
CRUD	(Create, Read, Update, Delete) Назва чотирьох базових операцій для роботи зі сховищем даних

ВСТУП

Сфера медицини завжди відігравала ключову роль в житті людини. В останні роки, коли світ зіткнувся з пандемією коронавірусної хвороби та російською агресією проти України, питання діагностування хвороб без можливості безпосередньої взаємодії з лікарями стало надзвичайно актуальною. Наведені події продемонстрували, що існує потреба в забезпеченні доступу до швидкої діагностики в умовах обмеження доступу пацієнтів до медичних закладів, перебування у віддалених чи небезпечних районах. Це спонукає до створення нових способів проведення попереднього аналізу хвороб.

Таким чином, метою проєкту є створення серверної частини системи для використання моделей штучного інтелекту для діагностування захворювань. Така система дозволить швидко отримати попередній висновок щодо ймовірності виникнення певних хвороб без необхідності звернення до лікарів.

Описуючи потенційні сфери застосування системи, що розроблюється, можна виділити наступні пункти:

- Забезпечення доступу до попередньої діагностики захворювань для пацієнтів, у яких немає можливості відвідати медичний заклад для виконання аналізу захворювань традиційними методами.
- Можливість швидкого виконання попереднього діагностування зменшить навантаженість медичного персоналу, що збільшить ефективність медичного обслуговування.
- Надання додаткової інформації лікарям при визначенні діагнозів для збільшення точності обстежень.
- Впровадження додаткових способів навчання для студентів-медиків та покращення їхньої кваліфікації через інтеграцію системи в освітній процес.

- Спосіб цифровізації сфери охорони здоров’я для підвищення якості медичних послуг через можливість розширення системи та додавання користувачів-лікарів з доступом до перегляду історії попередніх діагностик пацієнтів з використанням моделей штучного інтелекту та надання кваліфікованих діагнозів та плану лікування без безпосередньої взаємодії з пацієнтами.

Підсумовуючи наведені тези, система для спільного використання моделей штучного інтелекту для діагностування захворювань буде сприяти спрощенню доступу пацієнтів до виконання попереднього діагностування захворювань, підвищенню якості та швидкості медичного обслуговування та цифровізації сфери охорони здоров’я.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОГЛЯД ІСНУЮЧИХ СИСТЕМ-АНАЛОГІВ

1.1 Аналіз предметної області

Сфера медицини є критичною сферою в сучасному суспільстві, яка постійно розвивається і безпосередньо впливає на тривалість та якість життя людей. Важливим аспектом медичної сфери є діагностування захворювань [1] – процес виявлення хвороб, розладів, порушення здоров'я. Точність проведення діагностування впливає на весь процес подальшого лікування, визначає його ефективність та результати. Процес проведення діагностування захворювань постійно розвивається, щоб забезпечити більшу ймовірність виявлення захворювань та потенційних наслідків повною мірою, відстеження нових захворювань та їхніх ускладнень. Тому важливим завданням сфери медичного діагностування хвороб є впровадження нових методів для збільшення ефективності та точності діагностування. Ще одним завданням є оптимізація та автоматизація процесів виявлення хвороб, щоб надати кваліфікованим фахівцям інструменти для зменшення навантаження для виявлення захворювань для випадків, коли це можливо виконати та зменшити ризики традиційних ручних методів діагностування, що схильні до надання хибних результатів [2]. Одним з напрямків розвитку сфери медичного діагностування є впровадження сучасних технологій для вирішення зазначених задач.

Одним з підходів до використання технологій у діагностуванні хвороб є комп'ютерне діагностування (Computer-Aided Diagnosis, CAD) – використання систем, що надають засоби для виявлення захворювань за зображеннями [3]. Одним з основних напрямків CAD є використання методів прогнозування за допомогою штучного інтелекту для виявлення захворювань. Зазвичай роль

засобів CAD у діагностуванні захворювань полягає у допомозі попереднього визначення захворювань, у той час як медичний фахівець на основі рішень, отриманих з використанням засобів CAD, та інших результатів діагностування, визначає остаточний діагноз.

Також важливим завданням є надання доступу до використання засобів CAD. Оскільки використання моделей машинного навчання вимагає наявності технічних знань, для забезпечення можливості використання інструментів користувачам, необхідним є розроблення інтерфейсів, які спростять взаємодію з такими інструментами. Розробка серверного додатка, що буде взаємодіяти з інструментами CAD та надавати API для інтеграції з клієнтськими системами (наприклад, WEB-додатки, мобільні додатки), значно спрощує взаємодію з моделями штучного інтелекту для діагностування, робить систему гнучкою та доступною для великої кількості користувачів.

Відповідно, предметною областю даного проєкту є серверна система, що буде забезпечувати доступ користувачів до виконання діагностування захворювань за допомогою засобів комп'ютерної діагностики (CAD). Результати діагностування такими засобами не є офіційними діагнозами, вони лише забезпечують можливість виконання домедичного діагностування для виявлення захворювань у випадках, коли існують обмеження можливості виконання повноцінного діагностування медичними фахівцями.

1.2 Огляд існуючих систем-аналогів

У цьому пункті буде розглянуто наявні системи-аналоги, які надають можливість виконання діагностики захворювань за зображеннями використовуючи моделі ШІ. Буде виконано аналіз кожної системи та розглянуто процес виконання діагностики, використовуючи доступ користувача до платформи. Для виконання огляду було обрано систему

діагностування, розроблену командою викладачів Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського», - med.comsys.kpi.ua, систему для діагностування Qure.ai та програмне забезпечення для діагностування шкіри за допомогою ШІ Triage.

1.2.1 med.comsys.kpi.ua

med.comsys.kpi.ua [4] – це система діагностування розроблена командою викладачів Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського». Система є безкоштовною, але облікові дані для входу в систему можна отримати лише шляхом надсилання заявки на реєстрацію на пошту адміністрації системи. У системі доступні акаунти лише для ролі лікарів, відсутня можливість реєстрації для ролі пацієнт. Система надає можливість завантажувати зображення для діагностування моделями ШІ. Доступний вибір лише з запропонованих варіантів типів дослідження (наразі в системі доступний широкий вибір типів досліджень – дослідження різних аномалій шкіри, легень).

При цьому перевагою цієї системи є можливість виконувати дослідження одразу для декількох типів захворювань. Після завантаження фотографії та вибору типу діагностики можна переглянути результати обробки у вигляді ймовірності ураження обраними типами захворювань. Також присутня можливість переглядати результати попередніх діагностик. З недоліків, хоча ця система реалізовує основні можливості аналізу фотографій, відсутня можливість власноруч гнучко обирати типи діагностик та моделі ШІ, що будуть використані для діагностики, присутня лише можливість обирати з запропонованих системою. Також відсутня версійність аналізів, щоб мати змогу виконати повторну діагностику виконаних раніше досліджень та порівняти результати попередніх та наступних версій використаних моделей ШІ.

1.2.2 Qure.ai

Qure.ai [5] – це система, що надає доступ до аналізу зображень з використанням технологій на базі ШІ. Основною ціллю даної платформи є забезпечення можливості раннього виявлення хвороб для пацієнтів та надання можливості більш точної діагностики хвороб пацієнтів для лікарів. Qure.ai складається з підсистем, що надають доступ до діагностики різних типів зображень [6]:

- qXR – спеціалізується на аналізі зображень грудної клітини для виявлення аномалій та захворювань, пов'язаних з легенями, діафрагми, кістках грудної клітини та серця.
- qER – спеціалізується на аналізі зображень при скринінгу КТ та виявляє наявність проблем пошкодження черепа, крововиливів, інфарктів та інших дефектів.

Qure.ai надає безкоштовний пробний період для ознайомлення з системою, проте подальше використання платформи є платним. Реєстрація доступна на вебплатформі та у мобільних додатках. З рис. 1.1 видно, що користувач має доступ до завантаження зображень для аналізу за допомогою ШІ, проте відсутня можливість обирати моделі ШІ, що будуть використовуватись, а також типи досліджень для завантаженої фотографії (тип дослідження визначається автоматично для завантаженого файлу).

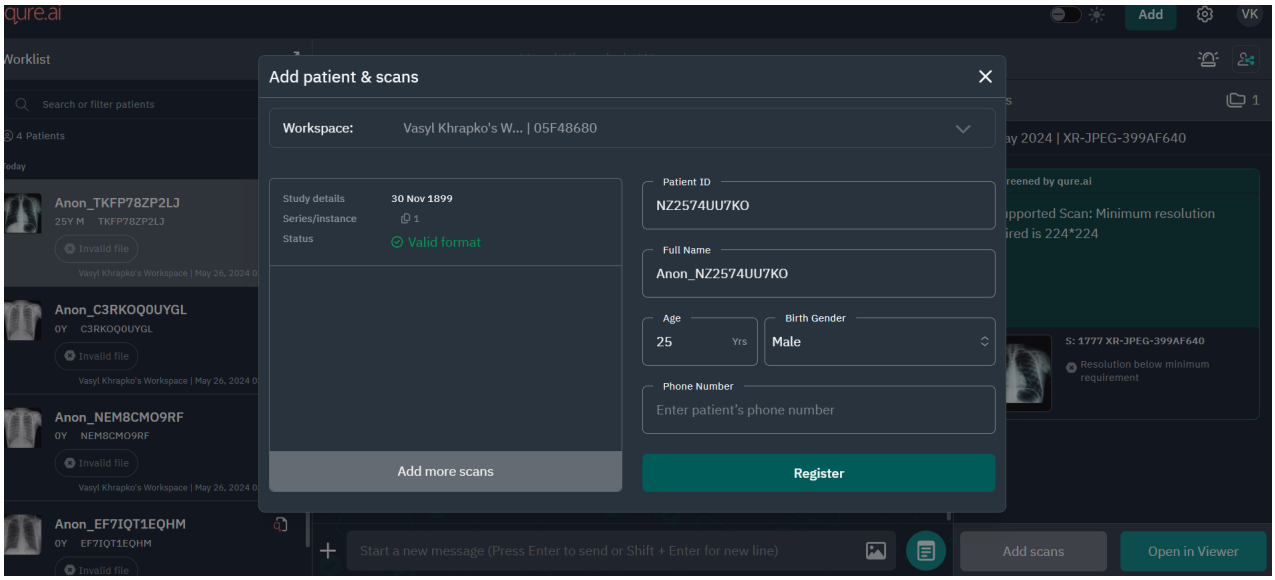


Рисунок 1.1 – Скриншот створення запиту на аналіз зображення в системі Qure.ai

Після виконання аналізу є доступ до перегляду результатів (рис. 1.2) дослідження, а також до історії попередніх досліджень.

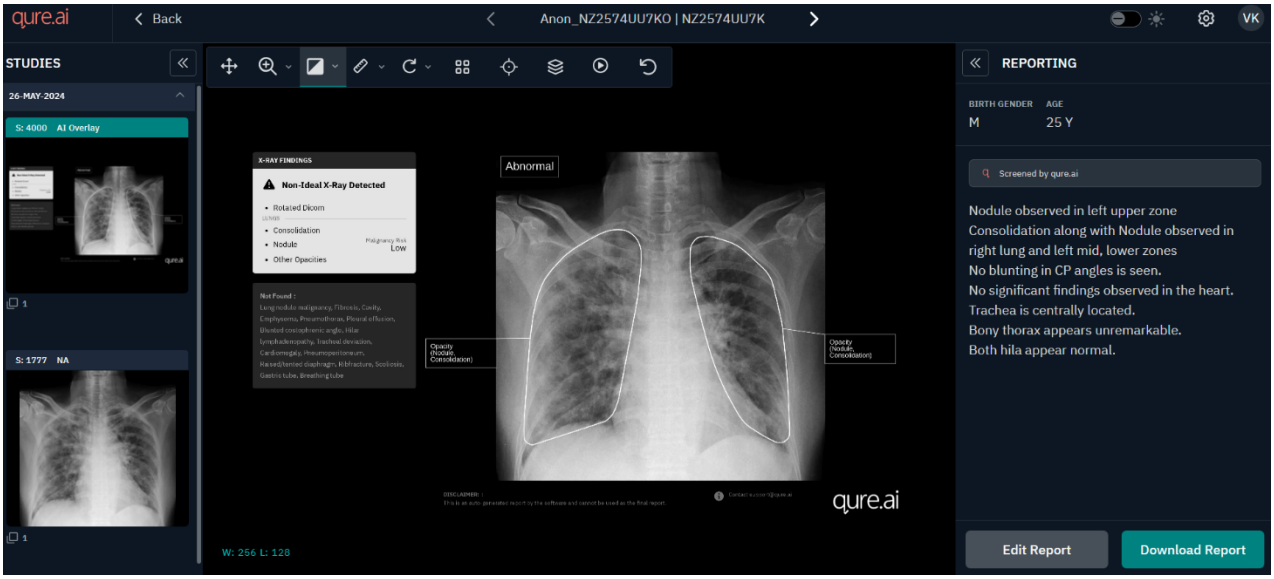


Рисунок 1.2 – Скриншот результатів діагностики в системі Qure.ai

У результаті дослідження пацієнт отримує текстовий висновок щодо виявлених при дослідженні аномалій. У цій системі також відсутня версійність аналізів для виконання повторних діагностик для завантаженого зображення та описаних для нього метаданих.

1.2.3 Triage

Triage [7] – це додаток, що використовує моделі ШІ для виявлення хвороб шкіри за фотографіями. Ця система орієнтована на індивідуальне використання користувачами для раннього виявлення власних захворювань, щоб допомоги в прийнятті рішення звернення до медичних закладів для лікування [8]. Для початку роботи в системі необхідно зареєструвати акаунт користувача, який надає безкоштовний доступ на 30 днів. Після цього необхідно купляти доступ до системи для виконання аналізу.

Система дозволяє завантажити фотографію шкіри та пройти опитування (рис. 1.3) щодо місця ураження на тілі та характеру ураження, після чого виконується аналіз фотографії за допомогою моделей ШІ.

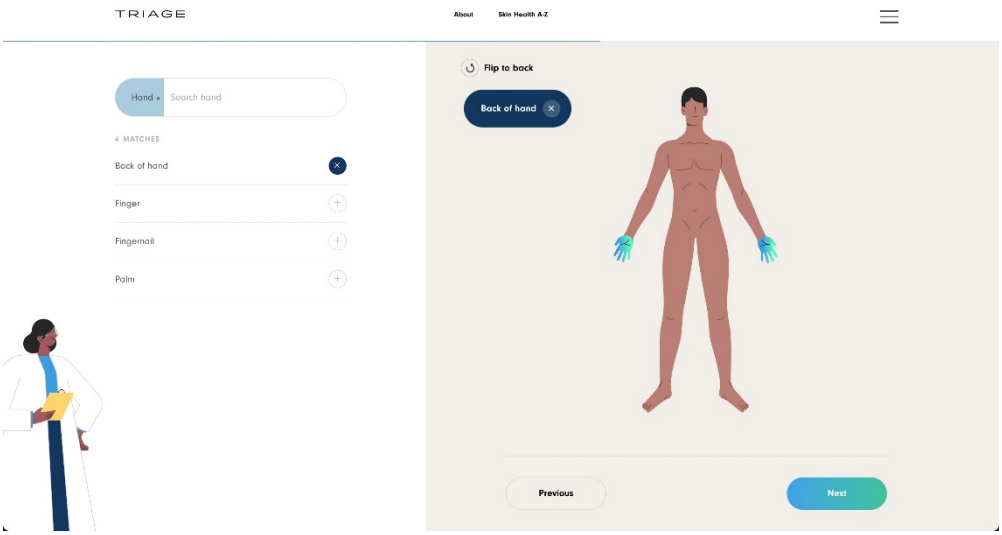


Рисунок 1.3 – Скриншот вибору місця ураження шкіри на тілі діагностики в системі Triage

У результаті користувач отримує інформацію про п’ять найімовірніших хвороб, виявлених моделями ШІ (рис. 1.4).

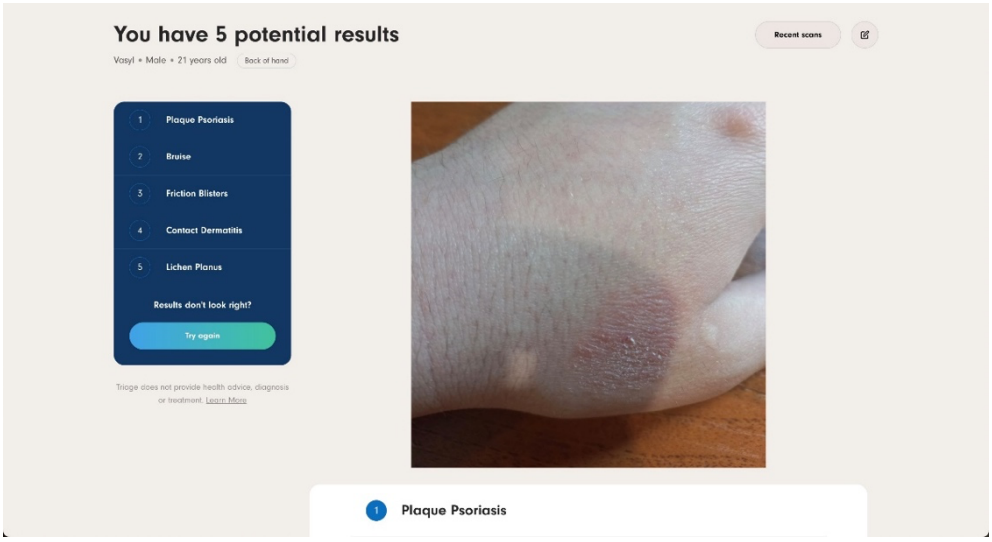


Рисунок 1.4 – Скриншот результату діагностики в системі Triage

Система надає доступ для перегляду результатів дослідження та перегляду історії дослідження. З недоліків системи можна визначити вузьку спеціалізацію на діагностику захворювань лише шкіри, а також неможливість обирати конкретні типи досліджень та моделі ШІ, що будуть використовуватись для аналізу.

1.3 Порівняльний аналіз оглянутих систем-аналогів

У табл. 1.1. надано порівняння загальних властивостей, функціональних та нефункціональних характеристик проаналізованих систем-аналогів, для виявлення переваг та недоліків кожної системи та формування вимог до розроблюваної системи.

Таблиця 1.1. – Порівняння розглянутих систем-аналогів

Ознака системи	***med.comsys.kpi.ua***	***Qure.ai***	***Triage***
Доступність системи	Є безкоштовний доступ	Доступ за платною підпискою	Доступ за платною підпискою

Продовження таблиці 1.1

Ознака системи	***med.comsys.kpi.ua***	***Qure.ai***	***Triage***
Доступна роль в системі	Лікар (з можливістю створення діагностик для багатьох пацієнтів)	Лікар (з можливістю створення діагностик для багатьох пацієнтів)	Звичайний користувач-пацієнт, з можливістю виконання індивідуальних діагностик
Спеціалізація досліджень	Великий перелік досліджень шкіри (14 видів дослідження), дослідження легенів	Дослідження зображень рентгену грудної клітини та зображення при скринінгу КТ	Доступне лише дослідження шкіри
Виконання діагностик зображень за допомогою моделей ШІ	Так	Так	Так
Перегляд результатів досліджень	Так	Так	Так
Перегляд історії досліджень	Так	Так	Так

Продовження таблиці 1.1

Ознака системи	***med.comsys.kpi.ua***	***Qure.ai***	***Triage***
Вибір типу діагностики	Так	Автоматичне виявлення типу діагностики відповідно до завантаженого зображення без явного вибору	Автоматичний вибір типу діагностики відповідно до завантаженого зображення та відповіді на додаткові питання до діагностики
Виконання декількох типів діагностики для однієї фотографії	Доступно лише для попередньо визначених наборів різних типів діагностик	Типи діагностики обираються одночасно, можливість вибору типів не передбачена	Відсутнє
Вибір моделей ШІ для виконання діагностики	Відсутній	Відсутній	Відсутній

Кінець таблиці 1.1

Ознака системи	***med.comsys.kpi.ua***	***Qure.ai***	***Triage***
Вибір одночасного використання декількох моделей ШІ для виконання досліджень	Вибір недоступний	Вибір недоступний	Вибір недоступний
Можливість користування системою під час виконання діагностики (асинхронна взаємодія для діагностики)	Так	Так	Так

Відповідно до порівняння, наведеного в табл. 1.1, системи Qure.ai та Triage не надають можливість використання платформ повною мірою без попередньої покупки підписки на платформу, що ускладнює доступ до виконання діагностик та унеможливлює їх використання для великої кількості потенційних користувачів. Для системи, що буде розроблюватись, можливість безкоштовного доступу користувачів є важливою. Також платформи

med.comsys.kpi.ua та Qure.ai орієнтуються на використання лікарями, що ускладнює їх використання для попереднього діагностування власних захворювань для користувачів-пацієнтів. Разом з тим, платформи Triage та Qure.ai надають доступ до досить обмежених типів досліджень, med.comsys.kpi.ua є більш універсальною платформою, проте все одно виділяється наявність лише діагностик шкіри та легенів. Для даного дипломного проєкту важливим є можливість використання великої кількості моделей ШІ та можливість до швидкого розширення типів діагностик та моделей ШІ, що використовуються.

Щодо наявності основних функцій, розглянуті моделі надають доступ до виконання діагностик зображень за допомогою моделей ШІ, перегляду результатів та перегляду історії діагностики, проте надають обмежений доступ до вибору типів діагностики, не надають можливість виконувати один тип діагностики за допомогою різних моделей ШІ для отримання більш точного середнього значення ймовірності ураження. Також розглянуті платформи не забезпечують можливість вибору моделей ШІ, якими буде виконана діагностика.

Таким чином, можна зробити висновок, що жодна з розглянутих систем не є досконалою та не надає доступ до всіх необхідних функцій. Відповідно, є доцільним створення більш довершеного рішення, тому розробка дипломного проєкту спрямована на створення системи, яка покриє більшу частину необхідної функціональності, що відсутня в системах-аналогах.

1.4 Визначення основних вимог до системи

Для проєктування та розробки серверної частини системи для спільного використання моделей штучного інтелекту для діагностики захворювань було визначено основні функціональні та нефункціональні вимоги для системи.

Система розрахована для використання користувачами з двома ролями: пацієнт та адміністратор. Відповідно, було складено таблиці функціональних (табл. 1.2) нефункціональних (табл 1.3) вимог до системи.

Таблиця 1.2. – Функціональні вимоги до системи

Вимога	***Роль користувача***	***Пріоритет***
Вхід в систему	Адміністратор, Пацієнт	Високий
Створення акаунтів користувачів	Адміністратор	Високий
Перегляд доступних типів діагностики	Адміністратор, Пацієнт	Високий
Керування доступними типами діагностик	Адміністратор	Високий
Перегляд доступних моделей ШІ для діагностики	Адміністратор, Пацієнт	Високий
Керування моделями ШІ	Адміністратор	Високий
Перегляд версій моделей ШІ	Адміністратор	Високий
Керування версіями моделей ШІ	Адміністратор	Високий
Вимкнення/увімкнення доступу до моделі ШІ	Адміністратор	Середній

Продовження таблиці 1.2

Вимога	***Роль користувача***	***Пріоритет***
Вимкнення/увімкнення доступу до певної версії моделі ШІ	Адміністратор	Середній
Створення запитів на діагностику захворювань за зображенням	Пацієнт	Високий
Перегляд результату діагностики	Пацієнт	Високий
Перегляд історії діагностик та результатів	Пацієнт	Середній
Моніторинг стану системи	Адміністратор	Низький
Перегляд логів компонентів системи	Адміністратор	Низький

Важливою деталлю поточної версії реалізації системи є те, що користувач-пацієнт не буде мати можливості самостійно реєструватись. Наразі передбачається, що за бажання отримання доступу до системи, майбутній користувач буде контактувати з адміністраторами поза системою, і адміністратор після отримання необхідних даних буде створювати акаунт користувача в системі.

Табл. 1.3 відображає перелік нефункціональних вимог, що відповідають за характеристики системи, які не стосуються безпосередньо сценаріїв у додатку.

Таблиця 1.3. – Нефункціональні вимоги до системи

Вимога	***Пріоритет***
Швидка обробка запитів на діагностику	Середній
Можливість відправити запит на діагностику і продовжувати користуватись системою	Високий
Масштабованість (можливість збільшувати ресурси для обробки, щоб підтримувати більшу кількість одночасних запитів)	Середній
Забезпечення конфіденційності особистих даних	Високий
Впровадження механізму автентифікації та авторизації користувачів	Високий
Надання документації з інтеграції клієнтів з серверним API	Середній
Забезпечення резервного копіювання даних	Середній

ВИСНОВОК ДО РОЗДІЛУ 1

У першому розділі було виконано аналіз предметної області, розглянуті системи, що забезпечують можливість виконання діагностики захворювань за допомогою аналізу зображень моделями ШІ. Було проаналізовано системи med.comsys.kpi.ua, Qure.ai та Triage. У результаті виконання дослідження системи було зроблено висновок, що хоча всі вони реалізовують можливість діагностування захворювань за допомогою моделей ШІ, було визначено, що вони не є досконалими та не вирішують завдання системи для діагностування повною мірою.

Таким чином, було прийнято рішення доцільності розробки нової системи відповідно до теми бакалаврської роботи, що буде враховувати переваги кожної розглянутої системи, буде вирішувати недоліки, що були виявлені в проаналізованих системах та забезпечувати необхідну функціональність для ефективного і повного вирішення завдання діагностики захворювань за зображеннями за допомогою моделей ШІ. Для цього було визначено перелік основних функціональних та нефункціональних вимог до системи, на основі яких буде виконуватись проєктування та розробка системи.

РОЗДІЛ 2. ОГЛЯД ПІДХОДІВ ТА ТЕХНОЛОГІЙ ДЛЯ РОЗРОБКИ СИСТЕМИ

2.1 Аналіз та вибір архітектури системи

У цьому пункті розглянуто способи реалізації архітектури розроблюваного серверного додатка, а саме монолітну та мікросервісну архітектуру, визначено переваги та недоліки кожного виду архітектури та виконано вибір архітектури відповідно до характеристик системи

2.1.1 Огляд монолітної архітектури

Монолітна архітектура [9] – це спосіб розробки програмного забезпечення, при якому всі компоненти та модулі застосунку об’єднані в одну кодову базу та працюють як один процес. При такій архітектурі застосунку всі модулі системи розгортаються разом.

Переваги монолітної архітектури:

1) Простота розробки, оскільки всі модулі системи містяться в одній кодовій базі, процес налаштування взаємодії між модулями та розуміння логіки роботи додатка є простішим.
2) Простота розгортання, оскільки модулі системи не потрібно розгортати окремо, всі вони містять в одному застосунку та розгортаються як один проєкт. Для підтримки доступності та функціональності сервера достатньо підтримувати лише один сервіс.
3) Швидша взаємодія між модулями додатка, оскільки взаємодія між компонентами системи відбувається в одному додатку, не потрібно організовувати комунікацію між сервісами.

4) Більша надійність при роботі з даними, оскільки всі дані опрацьовуються в межах одного процесу, не потрібно забезпечувати надійність даних при передачі між сервісами.

Проте дана архітектура має недоліки:

1) Складність масштабування, оскільки всі модулі додатка запускаються в одному процесі, при збільшенні навантаження на окремі модулі додатка, неможливо збільшити ресурси тільки для цих модулів, натомість доводиться додавати копію всього сервісу з усіма модулями.
2) Менша відмовостійкість, оскільки при збої в одній частині додатка, сервіс повністю стає недоступним. Додаток повністю не може працювати, якщо якийсь з його компонентів містить помилку.
3) Відсутність гнучкості у виборі технологій, оскільки всі модулі додатка розробляються в межах одного сервісу, відсутня можливість використання різних мов програмування, різних інструментів для розробки конкретних модулів системи.
4) Складність підтримки великого сервісу при збільшені кодової бази додатка, оскільки всі компоненти додатка знаходяться разом, збільшення обсягів додатка може спричиняти ускладнення для розробки.

2.1.2 Огляд мікросервісної архітектури

Мікросервісна архітектура [10] – підхід до організації компонентів додатка, коли система складається з певної кількості незалежних між собою сервісів, кожний з яких виконує певну функцію (відповідає за певний модуль системи).

Переваги мікросервісної архітектури:

1) Можливість масштабувати сервіси окремо, відповідно до їх навантаження, не спричиняючи масштабування всієї системи цілком.

2) Розгортання сервісів як окремих додатків збільшує відмовостійкість системи, оскільки навіть при збої одного модуля, інші сервіси системи можуть продовжувати роботу.
3) Можливість розділення додатка на прості модулі, кожний з яких може розроблятись та підтримуватись незалежно від інших.
4) Можливість використання різних технологій, мов програмування для кожного сервісу окремо.

Недоліки мікросервісної архітектури:

1) Необхідність організації комунікації між сервісами ускладнює розробку сервісів системи.
2) Необхідність налаштування розгортання для кожного сервісу.
3) Складність в підтримці узгодженості даних через необхідність передачі їх між сервісами та забезпечення узгодженості даних між сервісами.

2.1.3 Висновки щодо вибору архітектури

У попередніх пунктах було розглянуто переваги та недоліки монолітної та мікросервісної архітектур. Вибір щодо способу реалізації архітектури системи було виконано відповідно до розглянутих у першому розділі вимог до системи. Відповідно до розглянутих вимог можна виділити основний модуль системи – модуль для виконання діагностики зображень за допомогою моделей ШІ. Оскільки виконання діагностики за допомогою різних моделей ШІ буде вимагати виконання інтенсивних обчислень, що вплине на вимоги до ресурсів сервісів, варто передбачити можливість горизонтального масштабування для цієї частини додатка. Водночас, робота модулів системи, пов’язаних з обслуговуванням запитів від користувачів до сервера, забезпечення контролю доступу, автентифікації, авторизації, збереження даних про моделі системи, не вимагає складних обчислень та інтенсивного використання ресурсів, тому для

цих модулів доцільно використати один компонент системи та не розділяти їх на окремі мікросервіси, щоб не збільшувати складність системи та не вирішувати завдання організації комунікації між цими сервісами.

Отже, було вирішено обрати мікросервісну архітектуру та розділити систему на два сервіси: сервіс для виконання діагностик за зображенням за допомогою моделей ШІ та сервіс для обслуговування API запитів та виконання операцій, пов’язаних з користувачами, типа досліджень, моделями ШІ та їхніми версіями.

2.2 Організація комунікації між сервісами системи

У попередньому пункті було виконано вибір мікросервісної архітектури для реалізації серверної частини системи спільного використання моделей ШІ для діагностування захворювань за зображеннями. Це передбачає необхідність організації комунікації між сервісами системи. У даному пункті виконано огляд способів організації комунікації між сервісами та обрано більш ефективний спосіб.

2.2.1 Синхронна комунікація між сервісами

Синхронна комунікація між сервісами – це спосіб взаємодії між компонентами системи, при якому відправник запиту очікує негайної відповіді від одержувача запиту. Тобто при цьому способі комунікації відбувається блокування сервіса-відправника, доки він не отримає відповідь від сервіса-отримувача, або доки запит не завершитись з помилкою через завершення часу очікування відповіді [11].

Найпоширенішим способом організації синхронної комунікації між вебсервісами є використання HTTP/HTTPS запитів. Перевагами такого способу

комунікації є простота в реалізації, оскільки він не потребує додаткових інструментів для реалізації взаємодії між сервісами, а також наявність чіткої послідовності взаємодії між сервісами.

При цьому такий спосіб взаємодії не підходить для випадків, коли відповідь від сервіса-отримувача не може бути надіслана одразу, або якщо для обробки запиту потрібний значний час. У такому випадку сервіс, який звертається до іншого сервісу буде заблокований протягом всього часу обробки запиту і також існує можливість отримання помилки через закінчення часу очікування на обробку запиту.

2.2.2 Асинхронна комунікація між сервісами

Асинхронна комунікація [12] – це спосіб взаємодії між компонентами системи, при якому сервіс-відправник не очікує відповіді від сервіса-отримувача одразу, і може продовжувати виконання інших задач незалежно від того, чи отримав він дані від сервіса-отримувача одразу, чи отримає їх лише через певний час. Тобто обробка запитів може виконуватись в фоновому режимі, без необхідності блокування сервісу.

Для реалізації цього способу комунікації можна розглянути використання брокерів повідомлень для забезпечення надсилання повідомлень від сервісу-відправника, збереження запитів в черзі, що обслуговується брокером повідомлень, та надання задачі на обробку до сервіса-отримувача, коли він буде готовий виконати обробку запиту.

Перевагами цього способу є відсутність блокування сервісу-відпраника, він може продовжувати свою роботу без очікування негайної відповіді. Також даний спосіб забезпечує більшу надійність при комунікації, оскільки задачі зберігаються в чергах, що обслуговуються брокером повідомлень, та не спричиняють помилки через тайм-аути виконання запитів.

Такий спосіб є складнішим в реалізації, оскільки вимагає реалізацію взаємодії обох сервісів з брокером повідомлень. Також він не підходить, коли сервіс-відправник потребує негайного отримання даних.

2.2.3 Висновки щодо вибору способу комунікації між сервісами системи

Відповідно до розглянутих переваг та недоліків синхронної та асинхронної способів комунікації між сервісами було обрано використання асинхронного способу комунікації з використанням брокера повідомлень. Такий вибір обґрунтовано сценаріями взаємодії між сервісом для діагностування захворювань за зображеннями з використанням моделей ШІ та сервісом обробки API запитів. Комунікація між сервісами буде відбуватись для обробки запитів на діагностику та отримання результатів діагностики. Оскільки виконання діагностики може займати певний час та не відбуватись негайно, що унеможливлює використання синхронного способу комунікації, було обрано асинхронний спосіб комунікації. Ще одним аргументом для вибору асинхронного способу комунікації є необхідність забезпечення спільного доступу до моделей ШІ для багатьох користувачів. Оскільки сервіси з обробки зображень не можуть забезпечити можливість одночасної обробки великої кількості діагностик, збереження запитів на діагностику в черзі та послідовна обробка (відповідно до асинхронного способу комунікації з використанням брокера повідомлень) є способом розв’язання даної проблеми.

2.3 Архітектура компонентів системи

Відповідно до описаної архітектури сервісів системи та вимог до збереження даних про сутності системи в базі даних та файлів для діагностики та файлів моделей ШІ, було побудовано діаграму компонентів системи.

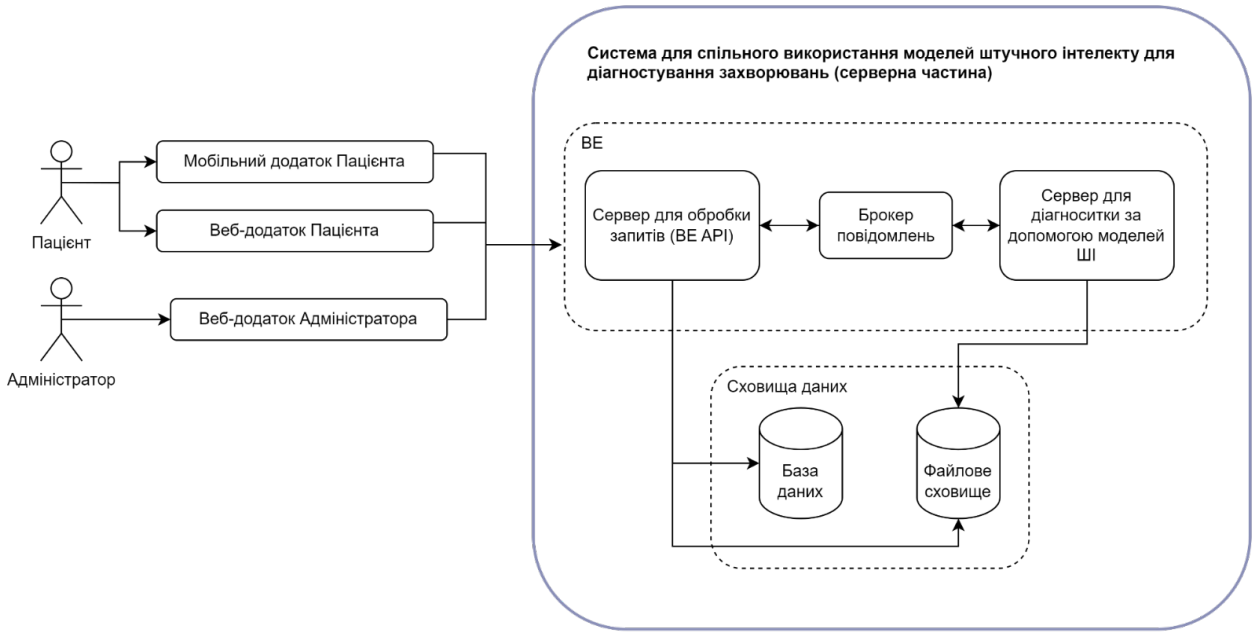


Рисунок 2.1 – Архітектура компонентів системи

Відповідно до рис. 2.1 розроблювальна система буде складатись з сервера для обробки API запитів від клієнтів, сервера для діагностики захворювань за зображеннями за допомогою моделей ШІ. Комунікація між цими серверами буде організована за допомогою брокера повідомлень. Також API сервер буде мати доступ до операцій в базі даних. Для збереження файлів для діагностики та для збереження файлів моделей, API сервер буде звертатись до файлового сховища. Також сервер для діагностики буде мати доступ до файлового сховища, щоб отримувати доступ до файлів моделей ШІ, а також до зображень для діагностики.

2.4 Аналіз та вибір типу бази даних

2.4.1 Реляційні бази даних

Реляційні бази даних [13] реалізовують збереження даних системи у вигляді таблиць, де кожна з таблиць відповідає за сутність бази даних, та зв’язків між таблицями. Взаємозв’язки в таких таблицях встановлюються за допомогою первинних ключів (primary key) та зовнішніх ключів (foreign key).

Перевагами реляційних баз даних є:

1) Чітко визначена структура таблиць системи
2) Наявність потужної мови формування запитів SQL, що забезпечує можливість зручно маніпулювати структурою бази даних, вмістом таблиць та виконувати ефективні запити на отримання складних даних з БД.
3) Наявність транзакційної моделі з використанням ACID транзакцій [14] для забезпечення контролю цілісності даних в БД.
4) Можливість оптимізації виконання запитів на отримання даних з таблиці за допомогою використання технології індексів в реляційній БД.

Водночас чітко визначена структура таблиць обмежує можливість гнучко змінювати моделі даних та накладає обмеження на додавання даних різної структури, що є недоліком до систем, у яких використовується велика кількість даних різної структури та немає чітко визначених структур сутностей БД. Також існують випадки, при яких виконання операцій з великою кількістю пов’язаних між собою таблиць є менш ефективним, ніж використання специфічних структур нереляційних баз даних, які можуть значно оптимізувати такі операції.

2.4.2 Нереляційні бази даних

На відміну від реляційних баз даних, нереляційні не використовують табличну модель збереження даних. Натомість вони надають власний формат збереження даних, наприклад зберігання даних у вигляді документів, словників, графів та інших структур [15].

Така реалізація забезпечує можливість більш гнучкої організації збереження даних та дозволяє динамічно змінювати структуру для збережуваних даних в базі даних. Також кожне з цих рішень має перелік задач, для якої орієнтована дана нереляційна база даних, тому використання нереляційних баз даних може ефективно покрити специфічні кейси.

Недоліками нереляційних баз даних є відсутність чіткої моделі, що зменшує організованість збереження даних. Також більшість нереляційних баз даних мають власні специфічні інструменти для виконання операцій в БД та маніпуляції з даними, тому потрібно реалізовувати різну логіку взаємодії з БД для різних рішень нереляційних баз даних. Також нереляційні бази даних не забезпечують такого рівня узгодженості даних, який забезпечують ACID транзакції в реляційних БД.

2.4.3 Вибір типу бази даних

Відповідно до проаналізованих переваг та недоліків реляційних та нереляційних баз даних, було прийнято рішення використовувати реляційну базу даних. Головним аргументом для такого рішення є можливість визначення наперед зв’язків між пов’язаними сутностями (типи діагностик – моделі ШІ – версії моделей ШІ).

2.5 Опис обраних технологій для компонентів системи

2.5.1 СУБД PostgreSQL

PostgreSQL [16] – це об’єктно-реляційна система управління базами даних з відкритим вихідним кодом. Те, що дана СУБД є рішенням з відкритим кодом надає переваги для вибору її, порівняно з іншими комерційними СУБД (наприклад, Oracle Database [17]), оскільки вона надає безкоштовний доступ та можливість повного контролю за кодовою базою проєкту.

PostgreSQL надає можливості користуватись перевагами реляційних баз даних, забезпечує створення БД з чіткою структурою таблиць, взаємодією між таблицями, забезпечення цілісності за допомогою ACID транзакцій. Також PostgreSQL надає можливість виконання паралельних запитів, що підвищує продуктивність запитів до бази даних. Також PostgreSQL має переваги над схожими рішеннями з відкритим кодом Maria DB (форк MySQL), оскільки PostgreSQL є об’єктно реляційною, що надає можливість підтримки додаткових типів, таких як JSON, JSONB, XML та інші.

Відповідно до характеристик та властивостей СУБД PostgreSQL повністю відповідає вимогам системи до зберігання даних, забезпечення швидкого та безпечного доступу до них та забезпечення цілісності даних в базі даних.

2.5.2 Брокер повідомлень RabbitMQ

RabbitMQ [18] – це потужний брокер повідомлень з відкритим вихідним кодом, що забезпечує протокол AMQP (Advanced Message Queuing Protocol) [19]. Він забезпечує можливість організації безпечної та ефективної асинхронної комунікації в розподілених системах.

Цей інструмент має відкритий програмний код, що робить можливість безкоштовного використання та локального розгортання або розгортання в приватних мережах. Це є перевагою даного брокера повідомлень порівняно з іншими комерційними рішеннями (наприклад, Amazon SQS [20], Azure Service Bus [21]).

RabbitMQ гарантує доставлення повідомлень через реалізацію механізму підтвердження доставлення (Acknowledgments). Також RabbitMQ має можливість забезпечення персистентності даних, шляхом зберігання даних про повідомлення на диску, що надає можливість відновлення повідомлень після збоїв. Також RabbitMQ є популярним рішенням, для роботи з яким розроблена велика кількість бібліотек на різних мовах програмування, що робить налаштування взаємодії з RabbitMQ швидким та простим. RabbitMQ надає можливість створення черг повідомлень, для взаємодії з різними компонентами системи. Також RabbitMQ підтримує можливість кластеризації, що надає можливість до масштабування при великому навантаженні на систему.

Також перевагою RabbitMQ, порівняно з ще одним популярним та ефективним брокером повідомлень Apache Kafka [22], є більш просте налаштування та управління RabbitMQ.

2.5.3 Файлове сховище MinIO

MinIO [23] – це високопродуктивне розподілене об’єктне сховище з відкритим вихідним кодом, що надає можливість використання його в системі як сховище для збереження зображень для діагностики та файлів моделей ШІ.

MinIO реалізовує інтерфейс, який є сумісним з інтерфейсом сервісу Amazon S3 [24], який реалізовує велика кількість рішень, що надає можливість зміни технології за потреби (при збільшені вимог до безпечності, масштабованості системи, виникнення передумов для переходу до використання хмарного провайдера). Перевагою MinIO порівняно з Amazon S3

є наявність відкритого вихідного коду, що робить можливим його безкоштовне використання та розгортання локально або в приватних мережах.

MinIO задовольняє всі вимоги системи щодо надійності збережених файлів, забезпечення швидкого та зручного доступу до файлів, забезпечення безпечності зберігання файлів та можливості до масштабування. Даний сервіс забезпечує можливість виконання реплікації даних для забезпечення відмовостійкості, дозволяє виконувати шифрування даних, забезпечуючи високий рівень захищеності доступу до файлів.

Також перевагою MinIO, порівняно зі схожим популярним рішенням файлового сховища Ceph, є простіше налаштування та керування системою.

ВИСНОВОК ДО РОЗДІЛУ 2

У другому розділі дипломної роботи було проаналізовано підходи до реалізації архітектури системи. У результаті порівняння монолітної та мікросервісної архітектури було обрано використання мікросервісної архітектури, через необхідність забезпечення можливості масштабування компонента діагностики захворювань за зображеннями за допомогою моделей ШІ. У результаті було обрано мікросервісну архітектуру та розподіл системи на два сервіси: сервіс для виконання діагностик за зображеннями за допомогою моделей ШІ та сервіс для обслуговування API запитів та виконання операцій з сутностями в базі даних.

Для обраного способу розробки архітектури було обрано використання асинхронної комунікації між сервісами через те, що виконання діагностики за допомогою моделей ШІ буде вимагати часу на виконання та не зможе забезпечити синхронну комунікацію та можливість обробляти одночасно всі запити на діагностику.

Було визначено основні компоненти системи та зв’язки між ними та побудовано діаграму архітектури компонентів системи. Для реалізації системи було обрано використання раніше описаних сервісів, брокера повідомлень для забезпечення асинхронної комунікації, бази даних як сховища даних та файлового сховища для збереження зображень користувачів для аналізу та файлів моделей ШІ.

Було виконано аналіз типів баз даних та обрано використання реляційної бази даних. Як систему управління базою даних було обрано PostgreSQL.

Також було обрано, проаналізовано та порівняно з аналогами інструменти для забезпечення асинхронної взаємодії між сервісами – брокера повідомлень RabbitMQ, а також було виконано дослідження переваг і недоліків файлового

сховища MinIO, обґрунтовано доцільність вибору даної технології для реалізації файлового сховища.

У результаті роботи над другим розділом диплома було отримано спроєктовану архітектуру додатка та обрано технології для реалізації системи.

РОЗДІЛ 3. РОЗРОБКА СИСТЕМИ

У попередніх розділах були визначені функціональні та нефункціональні вимоги до системи, проаналізовано та обрано архітектуру системи, спосіб комунікації між сервісами, способи взаємодії між компонентами системи, а також технології та сервіси, що будуть використовуватись під час розробки системи. У цьому розділі буде описано розробку системи з використанням обраних підходів і технологій.

3.1 Опис структури системи

Відповідно до описаної архітектури системи та архітектури компонентів, система буде складатись з двох сервісів: сервісу для обробки API запитів від клієнтів, а також сервера для діагностики захворювань за зображеннями за допомогою моделей ШІ. Для розробки кожного з сервісів було вирішено реалізовувати окремі серверні додатки. Для забезпечення простоти розробки та розгортання додатків, реалізація коду була виконана в спільному репозиторії, щоб був розділений на дві директорії з вихідним кодом для описаних сервісів системи (рис. 3.1). Відповідно сервіс для обробки API запитів був розроблений в директорії api, сервіс для діагностики зображень – у директорії diseases-analyzers.

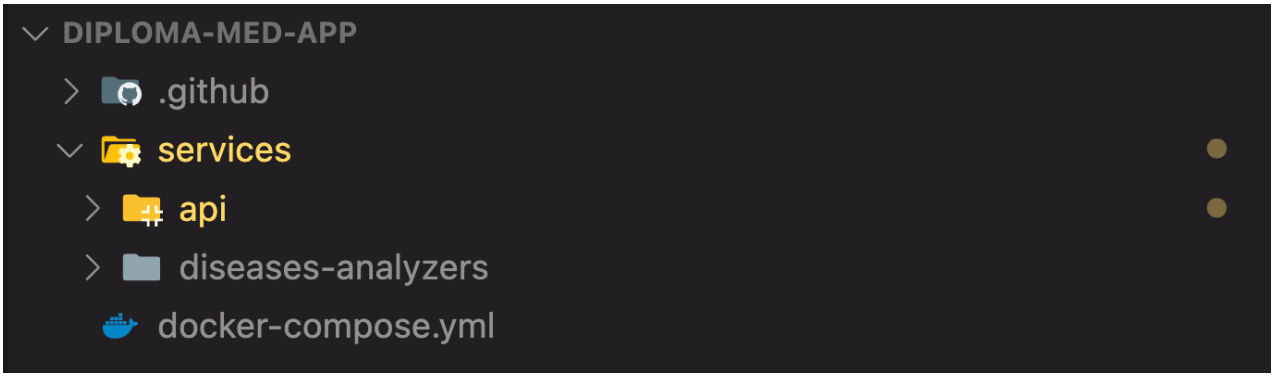


Рисунок 3.1 – Структура серверних додатків системи

3.2 Розробка сервісу для обслуговування API запитів

3.2.1 Структура сервісу

Сервіс для обслуговування API запитів побудований з використанням фреймворку для створення прогресивних серверних додатків на Node.js – Nest.js, який надає можливість побудови модульної архітектури серверного додатка. У такій архітектурі функціональність системи розділяється на окремі незалежні модулі, кожний з яких відповідає за реалізацію певної ізольованої частини додатка. Така структура забезпечує чітке виокремлення компонентів системи та меж їхньої відповідальності, можливість розробляти декілька модулів одночасно, незалежно один від одного, можливість повторно використовувати один модуль в різних компонентах системи та в різних системах, а також є засобом, за допомогою якого можна налаштовувати та тестувати модулі окремо від інших компонентів системи.

Використовуючи даний підхід було визначено структуру серверного додатка (рис. 3.2):

- common – директорія для визначення загальних утиліт, типів, об’єктів, інтерфейсів, допоміжних функцій, констант, сервісів, що будуть спільними та будуть використовуватись у різних модулях програми;
- config – модуль для конфігурації компонентів додатка, у якому будуть визначені параметри для конфігурації модулів, сервісів, та інших сутностей додатка, а також надано інструменти для отримання доступу до конфігураційних даних в інших модулях;
- systems – містить системні модулі, які забезпечують інфраструктурну функціональність додатка, а саме відповідають за інтеграцію із зовнішніми сховищами та сторонніми сервісами;
- database – модуль, що відповідає за інтеграцію з базою даних;

- message-queue – модуль, що відповідає за інтеграцію з чергою повідомлень;
- storage – модуль, що відповідає за взаємодію з файловим сховищем;
- app.module.ts – головний модуль, що вміщає в собі інші модулі, що використовуються в застосунку та забезпечує їхнє завантаження та конфігурацію;
- main.ts – вхідна точка додатка, у якій ініціалізується та налаштовується додаток, а також запускається HTTP-сервер.
- modules – директорія, що вміщає в собі модулі, які реалізовують бізнес-логіку додатка

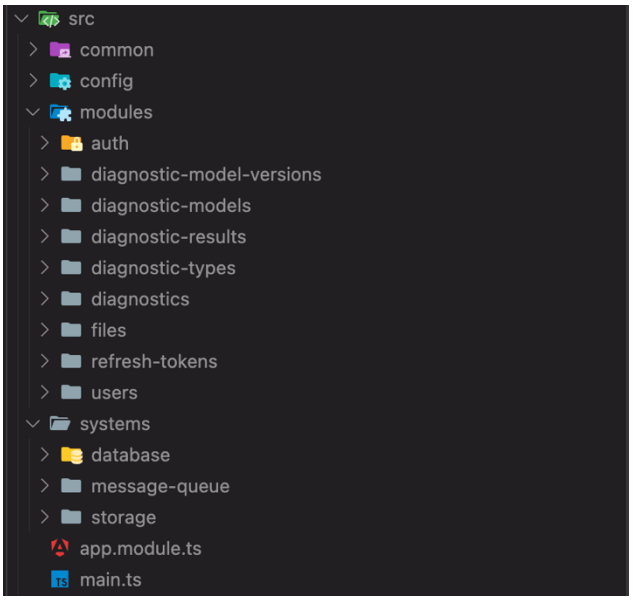


Рисунок 3.2 – Структура сервісу для обслуговування API запитів

Кожний з описаних модулів бізнес-логіки реалізовує окрему частину функціональності додатка, що виконується за допомогою визначення компонентів:

- контролерів, які відповідають за обробку HTTP-запитів від клієнтів та делегування виконання операцій до сервісів бізнес-логіки;
- сервісів, що реалізовують основні сценарії додатка, а саме забезпечують виконання бізнес-правил, роботу з даними та взаємодію з іншими сервісами.

3.2.2 Конфігурація серверного додатка

Для конфігурації серверного додатка було використано файли конфігурації оточення (.env), які застосовуються для визначення змінних середовища, що використовуються для налаштування додатка. Використання таких файлів забезпечує можливість виділити конфігурації в окремий файл, що забезпечує простоту в керуванні конфігураціями та безпеку, через можливість захисту всіх конфігурацій в одному файлі. Також такий підхід дозволяє ізолювати додаток від середовища, в якому він буде запускатись, а також гнучко керувати оточенням, в якому буде запущений додаток, наприклад, використовувати різні файли конфігурацій залежно від переданого значення оточення для розробки, production, або тестування.

Було визначено конфігурації додатка для запуску в режимі розробки (development) (рис. 3.3), режимі роботи з реальними користувачами (production) та тестуванні.

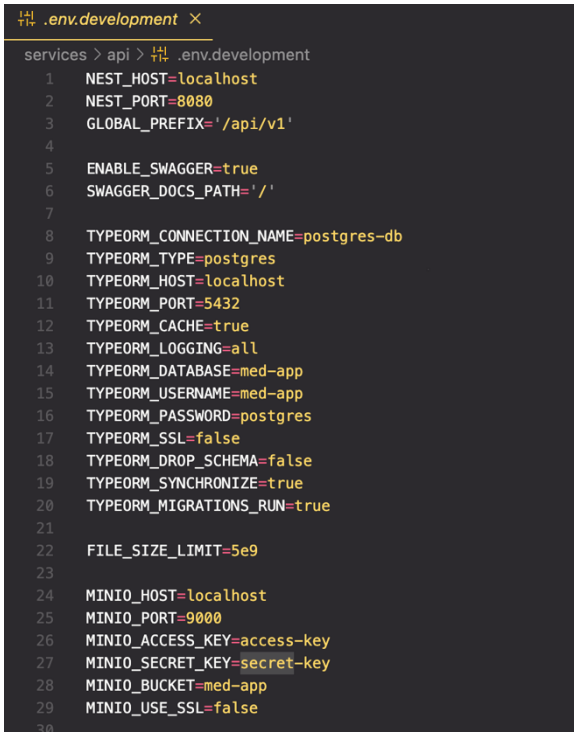
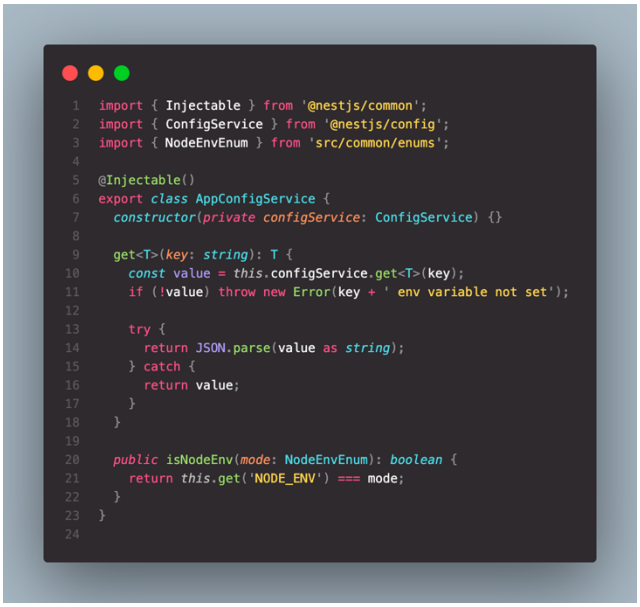


Рисунок 3.3 – Фрагмент конфігураційного .env файлу з параметрами додатка для роботи в режимі розробки

Для взаємодії з конфігураціями було визначено модуль AppConfigModule, у якому реалізовано сервіс AppConfigService (рис. 3.4), який використовує стандартний сервіс Nest.js – ConfigService – для отримання конфігурацій з .env файлу.



```
import { Injectable } from '@nestjs/common';
import { ConfigService } from '@nestjs/config';
import { NodeEnvEnum } from 'src/common/enums';

@Injectable()
export class AppConfigService {
  constructor(private configService: ConfigService) {}

  get<T>(key: string): T {
    const value = this.configService.get<T>(key);
    if (!value) throw new Error(key + ' env variable not set');

    try {
      return JSON.parse(value as string);
    } catch {
      return value;
    }
  }

  public isNodeEnv(mode: NodeEnvEnum): boolean {
    return this.get('NODE_ENV') === mode;
  }
}
```

Рисунок 3.4 – Фрагмент сервісу для взаємодії з конфігураціями .env

При налаштуванні модуля конфігурації відповідно до значення змінної середовища NODE_ENV визначається файл, який буде використовуватись для отримання конфігураційних змінних (рис. 3.5):

```
import { Global, Module } from '@nestjs/common';
import { ConfigModule } from '@nestjs/config';
import { AppConfigService } from './app-config.service';

@Global()
@Module({
  imports: [
    ConfigModule.forRoot({ envFilePath: `.env.${process.env.NODE_ENV}` }),
  ],
  providers: [AppConfigService],
  exports: [AppConfigService],
})
export class AppConfigModule {}
```

Рисунок 3.5 – Фрагмент коду з вибором файлу конфігурацій

3.2.3 Налаштування з’єднання з базою даних у додатку

У даному проєкті в якості сховища даних було обрано використання реляційної бази даних з використанням СУБД PostgreSQL. Для взаємодії з БД у додатку було використано ORM TypeORM, яка надає інструменти для підключення та взаємодії з базою даних. Для з’єднання з базою даних у додатку було визначено конфігурації у .env файлі. Після цього, було визначено метод сервісу конфігурації AppConfigService, для налаштування з’єднання з БД, який надає доступ до визначених конфігурацій (рис. 3.6):

Рисунок 3.6 – Скриншот метод для отримання налаштувань БД

У даному файлі визначено тип БД, з якою буде виконуватись взаємодія, параметри з’єднання, параметри логування, кешування, а також параметри роботи зі структурою та даними БД.

Після визначення методу для отримання налаштувань, було створено модуль DatabaseModule (рис. 3.7), який забезпечує з’єднання з БД та можливість взаємодії. Для цього був використаний TypeOrmModule, який виконує налаштування підключення до БД, використовуючи визначений раніше метод configService.getDbConfig().

```
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { AppConfigModule } from '../../config';
import { AppConfigService } from '../../config/app-config.service';

@Module({
  imports: [
    TypeOrmModule.forRootAsync({
      imports: [AppConfigModule],
      inject: [AppConfigService],
      useFactory: (configService: AppConfigService) => ({
        ...configService.getDbConfig(),
      }),
    }),
  ],
  exports: [TypeOrmModule],
})
export class DatabaseModule {}
```

Рисунок 3.7 – Створення модуля DatabaseModule для взаємодії з БД

3.2.4 Створення структури бази даних

Відповідно до визначених вимог до системи та бізнес-логіки було створено структуру бази даних (таблиць, їхніх колонок, та зв'язків між таблицями):

- Таблиця users відповідає за збереження інформації про пацієнтів (зберігає дані для автентифікації та авторизації email, password, role; а також персональні дані користувача name, age, sexAtBirth).
- Таблиця refresh-tokens відповідає за збереження refresh токенів користувача, для оновлення токенів для доступу до системи, та зберігає value – захешоване значення токена та посилається на таблицю users, на користувача, якому належить цей токен).
- Таблиця files відповідає за збереження даних про завантажені файли (поля fileName, fileExt, fileNameWithExt, pathInStorage, mimetype), src.
- Таблиця типів діагностик diagnostic-types (поля name, description)

- Таблиця завантажених моделей для діагностики diagnostic-models (поля name, description, queueName, status) та посилання на таблицю types – тип діагностики, для якої буде використовуватись дана модель).
- Таблиця версії моделі для діагностики diagnostic-model-versions (поля name, version, description, status, та посилання на модель diagnostic-models, до якої прив’язана дана версія, а також посилання на файл моделі files).
- Таблиця інформації про діагностику diagnostics (поля name, description, status), а також зв’язки з таблицею users (користувач, до якого відноситься дана діагностика), files (зображення для діагностики захворювань).
- Таблиця результатів діагностики diagnostic-results, що містить результати діагностики зображення користувача – числове значення ймовірності захворювання – за допомогою певної версії моделі (поля status, diseaseProbability, resultDescription), а також зв’язки з таблицею diagnostics, та diagnostic-model-versions (посилання на модель, якою було виконано дослідження)

Для створення описаних сутностей БД за допомогою коду додатка було використано інструменти TypeORM. Було визначено класи як сутності БД за допомогою декоратора @Entity. Для визначення структури таблиць (колонок, їхніх типів, обмежень цілісності) було використано декоратор @Column. Для визначення зв’язків між таблицями, були використані декоратори @OneToOne, @OneToMany, відповідно до типів зв’язків між таблицями. Для виокремлення

спільних колонок таблиць було створено клас CommonEntity, що визначає колонки id, createdAt, updatedAt, спільні для багатьох таблиць БД (рис. 3.8).

```typescript
export abstract class CommonEntity extends BaseEntity {
  @ApiProperty()
  @PrimaryGeneratedColumn('uuid')
  id: string;

  @ApiProperty()
  @CreateDateColumn({ type: 'timestamptz' })
  createdAt: Date;

  @ApiProperty({ readOnly: true })
  @UpdateDateColumn({ type: 'timestamptz' })
  updatedAt: Date;
}
```

Рисунок 3.8 – Скриншот класу CommonEntity, який визначає спільні колонки таблиць БД

Таким чином було виконано створення таблиць БД для всіх описаних сутностей. Повний приклад створення сутності за допомогою інструментів TypeORM наведений на рис. 3.9:

```typescript
@Entity({ name: 'diagnostic-results' })
export class DiagnosticResultEntity extends CommonEntity {
  @ApiProperty({
    type: () => DiagnosticEntity,
    required: true,
    nullable: false,
  })
  @ManyToOne(() => DiagnosticEntity, {
    onDelete: 'CASCADE',
    eager: false,
    nullable: false,
  })
  @JoinColumn()
  diagnostic: Partial<DiagnosticEntity>;

  @JoinColumn()
  @ApiProperty({
    type: () => FileEntity,
    required: true,
    nullable: false,
  })
  @ManyToOne(() => DiagnosticModelVersionEntity, {
    onDelete: 'CASCADE',
    nullable: false,
    eager: false,
  })
  modelVersion: Partial<DiagnosticModelVersionEntity>;

  @ApiProperty({
    enum: DiagnosticResultStatus,
    default: DiagnosticResultStatus.PENDING,
  })
  @Column({
    type: 'enum',
    enum: DiagnosticResultStatus,
    default: DiagnosticResultStatus.PENDING,
  })
  status: DiagnosticResultStatus;

  @Transform(({ value }) => parseFloat(value))
  @ApiProperty({ type: Number, required: false, example: 23, nullable: true })
  @Column({ type: 'numeric', scale: 2, precision: 5, nullable: true })
  diseaseProbability: number;

  @ApiProperty({
    type: 'string',
    maxLength: 500,
    required: false,
    nullable: true,
  })
  @Column({ length: 500, nullable: true })
  resultDescription: string;
}
```

Рисунок 3.9 – Скриншот створення таблиці diagnostic-results засобами TypeORM

У результаті було визначено та створено таблиці в БД (рис. 3.10).

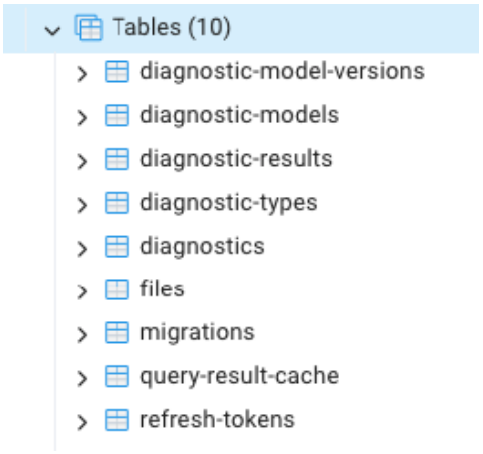
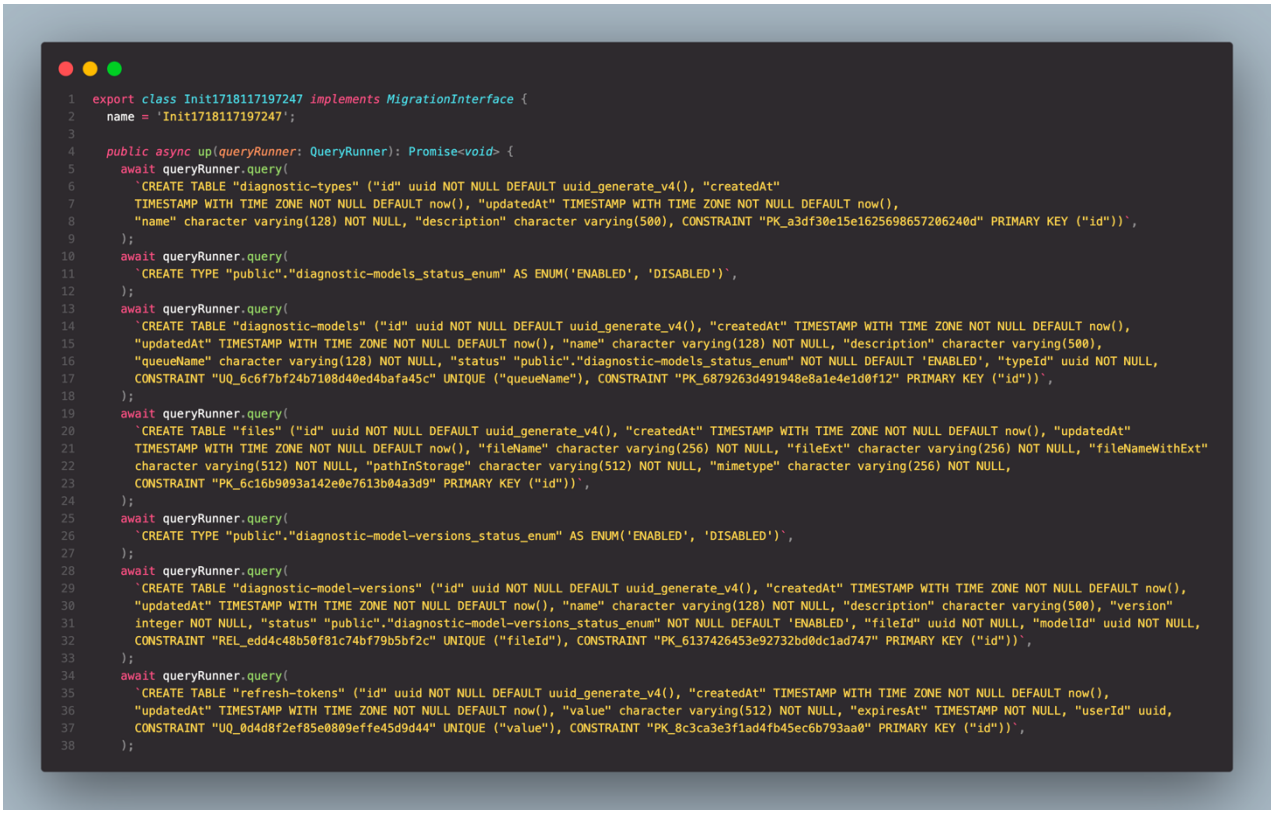


Рисунок 3.10 – Скриншот створеної структури таблиць БД

3.2.5 Міграції бази даних

Міграції [25] – це механізм для керування та зміни структури таблиць, колонок та зв'язків між таблицями в БД. Цей механізм допомагає змінювати структуру бази даних, без необхідності видалення та створення нової БД зі зміненою структурою, та є надзвичайно корисними при використанні в production системах, які працюють з таблицями даних реальних користувачів, які не можна видаляти. Тому для систем, які працюють з реальними користувачами, створення та зміна структури БД за допомогою міграцій є критично важливою. Оскільки поточна структура БД генерується за допомогою інструментів TypeORM, неможливо гарантувати, що не буде необхідності змінити структуру конкретної сутності або зв'язків між сутностями в процесі розвитку та підтримку додатка. Тому для генерації структури БД буде використано міграції БД, і при роботі з реальними користувачами для кожної зміни у визначених сутностях додатка, потрібно буде виконувати відповідні зміни за допомогою створення нових міграцій. Тому було реалізовано init-міграцію, що створює початкову структуру БД та встановлює зв'язки між таблицями. Для генерації міграції відповідно до поточної структури сутностей додатка, позначених через декоратор @Entity, було використано TypeORM CLI,

який забезпечує генерацію файлів міграцій. Фрагмент створеного файлу наведений на рис. 3.11.

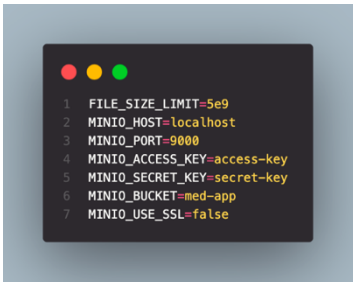


```
export class Init1718117197247 implements MigrationInterface {
  name = 'Init1718117197247';

  public async up(queryRunner: QueryRunner): Promise<void> {
    await queryRunner.query(
      `CREATE TABLE "diagnostic-types" ("id" uuid NOT NULL DEFAULT uuid_generate_v4(), "createdAt"
      TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT now(), "updatedAt" TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT now(),
      "name" character varying(128) NOT NULL, "description" character varying(500), CONSTRAINT "PK_a3df30e15e1625698657206240d" PRIMARY KEY ("id"))`,
    );
    await queryRunner.query(
      `CREATE TYPE "public"."diagnostic-models_status_enum" AS ENUM('ENABLED', 'DISABLED')`,
    );
    await queryRunner.query(
      `CREATE TABLE "diagnostic-models" ("id" uuid NOT NULL DEFAULT uuid_generate_v4(), "createdAt" TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT now(),
      "updatedAt" TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT now(), "name" character varying(128) NOT NULL, "description" character varying(500),
      "queueName" character varying(128) NOT NULL, "status" "public"."diagnostic-models_status_enum" NOT NULL DEFAULT 'ENABLED', "typeId" uuid NOT NULL,
      CONSTRAINT "UQ_6c6f7bf24b7108d40ed4bafa45c" UNIQUE ("queueName"), CONSTRAINT "PK_6879263d491948e8a1e4e1d0f12" PRIMARY KEY ("id"))`,
    );
    await queryRunner.query(
      `CREATE TABLE "files" ("id" uuid NOT NULL DEFAULT uuid_generate_v4(), "createdAt" TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT now(), "updatedAt"
      TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT now(), "fileName" character varying(256) NOT NULL, "fileExt" character varying(256) NOT NULL, "fileNameWithExt"
      character varying(512) NOT NULL, "pathInStorage" character varying(512) NOT NULL, "mimetype" character varying(256) NOT NULL,
      CONSTRAINT "PK_6c16b9093a142e0e7613b04a3d9" PRIMARY KEY ("id"))`,
    );
    await queryRunner.query(
      `CREATE TYPE "public"."diagnostic-model-versions_status_enum" AS ENUM('ENABLED', 'DISABLED')`,
    );
    await queryRunner.query(
      `CREATE TABLE "diagnostic-model-versions" ("id" uuid NOT NULL DEFAULT uuid_generate_v4(), "createdAt" TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT now(),
      "updatedAt" TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT now(), "name" character varying(128) NOT NULL, "description" character varying(500), "version"
      integer NOT NULL, "status" "public"."diagnostic-model-versions_status_enum" NOT NULL DEFAULT 'ENABLED', "fileId" uuid NOT NULL, "modelId" uuid NOT NULL,
      CONSTRAINT "REL_edd4c48b50f81c74bf79b5bf2c" UNIQUE ("fileId"), CONSTRAINT "PK_6137426453e92732bd0dc1ad747" PRIMARY KEY ("id"))`,
    );
    await queryRunner.query(
      `CREATE TABLE "refresh-tokens" ("id" uuid NOT NULL DEFAULT uuid_generate_v4(), "createdAt" TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT now(),
      "updatedAt" TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT now(), "value" character varying(512) NOT NULL, "expiresAt" TIMESTAMP NOT NULL, "userId" uuid,
      CONSTRAINT "UQ_0d4d8f2ef85e0809effe45d9d44" UNIQUE ("value"), CONSTRAINT "PK_8c3ca3e3f1ad4fb45ec6b793aa0" PRIMARY KEY ("id"))`,
    );
  }
```

Рисунок 3.11 – Скриншот фрагмента init-міграції для створення структури БД

3.2.6 Інтеграція з файловим сховищем MinIO

Для інтеграції з файловим сховищем MinIO було додано конфігурації для з'єднання з сервісом, а також конфігурації для лімітів розміру файлів, що завантажуються в системі (рис. 3.12).



```
FILE_SIZE_LIMIT=5e9
MINIO_HOST=localhost
MINIO_PORT=9000
MINIO_ACCESS_KEY=access-key
MINIO_SECRET_KEY=secret-key
MINIO_BUCKET=med-app
MINIO_USE_SSL=false
```

Рисунок 3.12 – Скриншот налаштувань для інтеграції з MinIO

Після цього було створено модуль StorageModule, що містить в собі логіку роботи з файловим сховищем. Для виконання операцій завантаження файлів, читання файлів та видалення файлів було створено сервіс StorageService, який реалізовує методи для виконання перерахованих операцій для файлового сховища MinIO (рис. 3.13).

Рисунок 3.13 – Скриншот сервісу StorageService для взаємодії з MinIO

Після цього даний сервіс було використано в модулі FilesModule, який забезпечує отримання файлу від користувача в контролері FilesController, та виконання операцій сервісом FilesStorage: збереження файлу на файлове сховище MinIO та збереження даних про файл у БД (рис. 3.14). Також було реалізовано методи контролера та сервісу для видалення файла.

```
async createOne(
  file: IMultipartFile,
  data?: Partial<FileEntity>,
  pathToFile = UNCATEGORIZED_ROOT_DIRECTORY,
  transactionManager?: EntityManager,
) {
  const repository = transactionManager
    ? transactionManager.getRepository<FileEntity>(
        this.fileEntityRepository.target,
      )
    : this.fileEntityRepository;

  const uploadedFileEntity = await this.storageService.createOne(
    file,
    pathToFile,
  );
  const entityLike = repository.create(uploadedFileEntity);
  const entityToCreate = repository.merge(entityLike, data);
  const { id } = await repository.save(entityToCreate).catch(() => {
    throw new BadRequestException(filesServiceErrorMessages.invalidData);
  });
  return this.findOne(
    { id },
    { loadEagerRelations: true },
    transactionManager,
  );
}
```

Рисунок 3.14 – Код методу збереження файлу у сервісі FilesService

3.2.7 Налаштування взаємодії з чергою повідомлень RabbitMQ

Для налаштування взаємодії з чергою повідомлень RabbitMQ у серверному додатку було додано конфігурації для з'єднання з сервісом (протокол, хост, username, password). Для інтеграції був налаштований MessageQueueModule, у якому реалізовано код сервісів для встановлення з'єднання з RabbitMQ (RabbitMQConnectionService), сервіс для надсилання повідомлень (RabbitMQPublisherService) та сервіс для отримання повідомлень (RabbitMQConsumerService).

Сервіс RabbitMQConnectionService (рис. 3.15) за допомогою бібліотеки amqplib [26] забезпечує можливість під'єднання до сервісу RabbitMQ, а також надає можливість створювати канали взаємодії та черги повідомлень. Цей

сервіс використовується в інших двох сервісах для роботи з RabbitMQ, забезпечуючи роботу зі з'єднанням, каналами та чергами.

```
@Injectable()
export class RabbitMQConnectionService {
  private readonly logger = new Logger(RabbitMQConnectionService.name);

  private readonly rabbitMQProtocol =
    this.configService.get<string>('RABBIT_MQ_PROTOCOL');
  private readonly rabbitMQHost =
    this.configService.get<string>('RABBIT_MQ_HOST');
  private readonly rabbitMQUser = this.configService.get<string>(
    'RABBIT_MQ_DEFAULT_USER',
  );
  private readonly rabbitMQPassword = this.configService.get<string>(
    'RABBIT_MQ_DEFAULT_PASS',
  );
  private connection: amqp.Connection;
  private channel: amqp.Channel;

  constructor(
    private readonly configService: AppConfigService,
    @Inject(forwardRef(() => DiagnosticsService))
    private readonly diagnosticsService: DiagnosticsService,
  ) {}

  async setUpConnection() {
    this.connection = await amqp.connect(
      `${this.rabbitMQProtocol}://${this.rabbitMQUser}:${this.rabbitMQPassword}@${this.rabbitMQHost}`,
    );
    this.channel = await this.connection.createChannel();
    await this.assertQueueForModels();
  }

  async assertQueue(queue: string) {
    this.logger.debug(`Asserting queue - ${queue}`);
    await this.channel.assertQueue(queue, { durable: true });
  }

  async assertQueueForModels() {
    const modelQueueNames =
      await this.diagnosticsService.selectModelQueueNames();

    for (const modelQueueName of modelQueueNames) {
      await this.assertQueue(modelQueueName);
    }
  }

  getChannel() {
    return this.channel;
  }
}
```

Рисунок 3.15 – Скриншот коду RabbitMQConnectionService

Сервіс RabbitMQPublisherService (рис. 3.16) забезпечує відправлення повідомлень у чергу. За допомогою нього буде виконуватись відправка повідомлень в чергу для обробки сервісом для діагностики зображень за допомогою моделей ШІ.

```
@Injectable()
export class RabbitMQPublisherService {
  private readonly logger = new Logger(RabbitMQPublisherService.name);

  constructor(
    private readonly rabbitMQConnectionService: RabbitMQConnectionService,
  ) {}

  async addMessageToQueue(queue: string, messageData: unknown) {
    const channel = this.rabbitMQConnectionService.getChannel();
    await this.rabbitMQConnectionService.assertQueue(queue);
    channel.sendToQueue(queue, this.convertData(messageData), {
      persistent: true,
    });

    this.logger.debug(
      `Message sent to queue (${queue}) with task data ${messageData}`,
    );
  }

  convertData(data: unknown) {
    return Buffer.from(JSON.stringify(data));
  }
}
```

Рисунок 3.16 – Скриншот коду RabbitMQPublisherService

Сервіс RabbitMQConsumerService (рис. 3.17) реалізовує код для обробки повідомлень від RabbitMQ. Хук OnInit [27] використовується для виконання операцій після певних подій додатка (в цьому випадку після ініціалізації модуля MessageQueueModule буде встановлено з'єднання з RabbitMQ та назначено обробку повідомлень про результат діагностики захворювань за зображеннями користувачів.

```
@Injectable()
export class RabbitMQConsumerService {
  private readonly logger = new Logger(RabbitMQConsumerService.name);

  constructor(
    private readonly rabbitMQConnectionService: RabbitMQConnectionService,
    private readonly diagnosticsService: DiagnosticsService,
  ) {}

  async onModuleInit() {
    await this.rabbitMQConnectionService.setUpConnection();

    const channel = this.rabbitMQConnectionService.getChannel();
    await this.setUpDiagnosticResultHandler(channel);
  }

  async setUpDiagnosticResultHandler(channel: Channel) {
    const queue = MessageQueueName.DIAGNOSTIC_RESULT;
    await this.rabbitMQConnectionService.assertQueue(queue);
    await channel.consume(
      queue,
      this.handleDiagnosticResultMessage.bind(this, channel),
    );
  }

  async handleDiagnosticResultMessage(channel: Channel, msg: ConsumeMessage) {
    let data = null;
    try {
      data = this.getMessageData<DiagnosticResultData>(msg);
      await this.diagnosticsService.handleDiagnosticResult(data);
      channel.ack(msg);
    } catch (err) {
      this.handleError(err, msg, data);
    }
  }

  getMessageData<T extends object>(msg: ConsumeMessage): T {
    return JSON.parse(msg.content.toString());
  }

  handleError(err: Error, msg?: ConsumeMessage, data?: unknown) {
    this.logger.error(
      `Error ${err}, when handling message:`,
      msg,
      'Message data: ',
      data,
    );
  }
}
```

Рисунок 3.17 – Скриншот коду RabbitMQConsumerService

При обробці результатів діагностики дані про результат зберігаються у відповідний запис таблиці diagnostic-results.

3.2.8 Реалізація сервісів

Сервіси у серверному додатку відповідають за реалізацію операцій, пов'язаних з бізнес-логікою, обробкою даних та взаємодію зі сторонніми

сервісами та сховищами (базами даних, чергами повідомлень, файловими сховищами, зовнішніми системами). У Nest.js додатках сервіси визначаються через декоратор @Injectable та інкапсулюють логіку роботи з певною частиною бізнес-логіки, що визначається модулем, у якій сервіс був створений. Таким чином, кожний сервіс інкапсулює в собі логіку для вирішення одного завдання, що робить код структурованим та легким для розуміння. Також однією з основних ідей Nest.js є взаємодія через механізм ін'єкції залежностей, який дозволяє використовувати методи одних сервісів у середині інших, при цьому прив'язуючись лише до інтерфейсу сервісів, що використовуються, а не для конкретної реалізації. Такий підхід зменшує зв'язність між модулями та дозволяє змінювати деталі реалізації одних сервісів, не порушуючи логіку роботи інших.

Для спрощення розробки та виділення основних операцій роботи з даними в додатку (CRUD операцій), був виділений абстрактний базовий сервіс BaseService (рис 3.18), який реалізовує операції на TypeORM сутностями, за допомогою інструментів TypeORM Repository, використовуючи дженерики (generics). Використання типів-дженериків полягає в ідеї абстрагування від конкретних типів даних при розробці, що дозволяє створювати загальний код, який буде працювати з багатьма різними типами даних, що забезпечує можливість гнучко розробляти універсальні функції, класи, інтерфейси та інші програмні сутності, які зможуть бути повторно використані для різних типів даних [28].

```
export abstract class BaseService<T extends CommonEntity> {
  constructor(
    protected entityRepository: Repository<T>,
    protected serviceErrorMessages: TServiceErrorMessages,
  ) {}

  async findAll(
    options: FindManyOptions<T> = { loadEagerRelations: true },
    transactionManager?: EntityManager,
  ): Promise<T[]> {
    const repository = transactionManager
      ? transactionManager.getRepository<T>(this.entityRepository.target)
      : this.entityRepository;

    return repository.find(options).catch(() => {
      throw new NotFoundException(this.serviceErrorMessages.entitiesNotFound);
    });
  }

  async findOne(
    conditions: FindOptionsWhere<T>,
    options: FindOneOptions<T> = { loadEagerRelations: true },
    transactionManager?: EntityManager,
  ): Promise<T> {
    const repository = transactionManager
      ? transactionManager.getRepository<T>(this.entityRepository.target)
      : this.entityRepository;

    return repository
      .findOneOrFail({
        ...options,
        where: conditions,
      })
      .catch(() => {
        throw new NotFoundException(this.serviceErrorMessages.entityNotFound);
      });
  }

  async createOne(
    entity: Partial<T>,
    transactionManager?: EntityManager,
  ): Promise<T> {
    const repository = transactionManager
      ? transactionManager.getRepository<T>(this.entityRepository.target)
      : this.entityRepository;

    const entityToCreate = repository.create(entity as T);
    const { id } = await repository.save(entityToCreate).catch(() => {
      throw new BadRequestException(this.serviceErrorMessages.invalidData);
    });
    return this.findOne(
      { id } as FindOptionsWhere<T>,
      { loadEagerRelations: true },
      transactionManager,
    );
  }
```

Рисунок 3.18 – Скриншот фрагмента коду базового сервісу BaseService

За допомогою наслідування даного сервісу, було розроблено сервіси роботи з бізнес-логікою додатка:

- UsersService – використовує базові CRUD операції для роботи з таблицею користувачів, а також реалізовує методи для отримання даних про користувачів адміністратором, приховуючи їхні персональні дані (вік та стать).
- DiagnosticTypesService – використовує базові CRUD операції для роботи таблицею типів діагностик
- DiagnosticModelVersionsService – використовує базові CRUD операції для роботи таблицею версій моделей для діагностики.
- DiagnosticModelsService – окрім базових CRUD операцій для роботи з таблицею моделей для діагностики, використовує сервіс StorageService для збереження файлів моделей на файловому сховищі та сервіс DiagnosticModelVersionsService для керування версіями моделей, оновленням статусів моделей та їхніх версій, а

також наданням інформації клієнтам про доступність моделей та їхніх версій.

- DiagnosticResultsService – використовує базові CRUD операції для роботи таблицею результатів діагностик.
- DiagnosticsService – окрім базових CRUD операцій для роботи з таблицею діагностик, використовує StorageService для збереження зображень для діагностики, а також створенням запитів до сервісів аналізу зображень за допомогою моделей ШІ (рис. 3.19). Також даний сервіс надає користувачам інформацію про результати діагностики та форматує їх у зручний для клієнтів формат зі структурою, відображеної на рис. 3.20.

```
async uploadDiagnosticByModels(
  diagnosticId: string,
  modelIds: string[],
  imagePathInStorage: string,
  transactionManager?: EntityManager,
): Promise<void> {
  const uniqueModelIds = new Set(modelIds); // remove duplicate values
  let uploadedModels = 0;

  for (const modelId of uniqueModelIds) {
    const model = await this.diagnosticModelsService
      .selectAvailableModelVersion({ id: modelId }, null, transactionManager)
      .catch(() => null);

    const availableVersion = model?.versions?.[0];

    if (!availableVersion) {
      continue;
    }

    const result = await this.diagnosticResultsService.createOne(
      {
        diagnostic: { id: diagnosticId },
        modelVersion: { id: availableVersion.id },
      },
      transactionManager,
    );

    await this.rabbitMQPublisherService.addMessageToQueue(model.queueName, {
      resultId: result.id,
      imagePathInStorage,
    });

    uploadedModels++;
  }

  if (uploadedModels === 0) {
    throw new BadRequestException(
      ErrorMessagesEnum.NO_VALID_MODELS_FOR_DIAGNOSIS_SPECIFIED,
    );
  }
}
```

Рисунок 3.19 – Скриншот збереження сутності діагностики та створення повідомлень для аналізу в сервісах діагностики за допомогою моделей ШІ

```
{
  "id": "034e07bb-00f4-44a6-a3ed-1420a686a3a8",
  "createdAt": "2024-06-12T10:19:38.471Z",
  "updatedAt": "2024-06-12T10:19:38.471Z",
  "name": "My new skin diagnostic",
  "description": "",
  "status": "COMPLETED_WITH_ISSUES",
  "image": {
    "id": "c996d8ce-ce84-4171-b491-3190f66cfac2",
    "createdAt": "2024-06-12T10:19:38.471Z",
    "updatedAt": "2024-06-12T10:19:38.471Z",
    "fileName": "Skin1.png",
    "fileExt": ".png",
    "fileNameWithExt": "1718187578472-Skin1.png",
    "pathInStorage": "uploads/diagnostic-images/e5333601-bfee-4af8-bc01-ad1370087c32/034e07bb-00f4-44a6-a3ed-1420a686a3a8/1718187578472-Skin1.png",
    "mimetype": "image/png",
    "src": "http://localhost:9000/med-app/uploads/diagnostic-images/e5333601-bfee-4af8-bc01-ad1370087c32/034e07bb-00f4-44a6-a3ed-1420a686a3a8/1718187578472-Skin1.png"
  },
  "types": {
    "1ae1b71d-e508-459d-a147-03a9a6ba5071": {
      "id": "1ae1b71d-e508-459d-a147-03a9a6ba5071",
      "createdAt": "2024-06-11T16:28:34.241Z",
      "updatedAt": "2024-06-11T16:28:34.241Z",
      "name": "Діагностування фурункульозу",
      "description": null,
      "models": {
        "66b16e87-cca0-48aa-add8-1b50dcd2bb37": {
          "id": "66b16e87-cca0-48aa-add8-1b50dcd2bb37",
          "createdAt": "2024-06-12T10:12:32.073Z",
          "updatedAt": "2024-06-12T10:12:32.073Z",
          "name": "Замовлена Tensorflow модель для діагностування фурункульозу",
          "description": "Model description example",
          "queueName": "EXT_FURUNKOLOSIUS_CUSTOM_MODEL_QUEUE",
          "status": "ENABLED",
          "versions": {
            "b1c87753-debf-420c-a976-2858d25e42aa": {
              "id": "b1c87753-debf-420c-a976-2858d25e42aa",
              "createdAt": "2024-06-12T10:14:16.625Z",
              "updatedAt": "2024-06-12T10:14:16.625Z",
              "name": "Example name",
              "description": "Model description example",
              "version": 1,
              "status": "ENABLED",
              "results": {
                "34e1f8e4-1641-46d9-b439-661d1b05867a": {
                  "id": "34e1f8e4-1641-46d9-b439-661d1b05867a",
                  "createdAt": "2024-06-12T10:19:38.471Z",
                  "updatedAt": "2024-06-12T10:19:38.471Z",
                  "status": "FAILED",
                  "diseaseProbability": null,
                  "resultDescription": null
                }
              }
            }
          }
        },
        "db64028c-6cb2-4ce7-8627-e5899f7e51dd": {
          "id": "db64028c-6cb2-4ce7-8627-e5899f7e51dd",
          "createdAt": "2024-06-12T10:11:55.440Z",
          "updatedAt": "2024-06-12T10:11:55.440Z",
          "name": "Модель Tensorflow для діагностування фурункульозу",
          "description": "Model description example",
          "queueName": "FURUNKOLOSIUS_CUSTOM_MODEL_QUEUE",
          "status": "ENABLED",
          "versions": {
            "9b285fe3-65ff-4ff0-b54e-9a98e6687068": {
              "id": "9b285fe3-65ff-4ff0-b54e-9a98e6687068",
              "createdAt": "2024-06-12T10:14:38.257Z",
              "updatedAt": "2024-06-12T10:14:38.257Z",
              "name": "Furnuculosius",
              "description": "Model description example",
              "version": 3,
              "status": "ENABLED",
              "results": {
                "d1bdde1c-10bb-4f36-a106-b9bb974761e2": {
                  "id": "d1bdde1c-10bb-4f36-a106-b9bb974761e2",
                  "createdAt": "2024-06-12T10:19:38.471Z",
                  "updatedAt": "2024-06-12T10:19:38.471Z",
                  "status": "COMPLETED",
                  "diseaseProbability": "32.80",
                  "resultDescription": null
                }
              }
            }
          }
        }
      }
    }
  }
}
```


Рисунок 3.20 – Скриншот формату отримання інформації про стан діагностики зображення

3.2.9 Реалізація модуля автентифікації та авторизації

Для перевірки особи користувача, прав доступу користувача до ресурсів системи та забезпечення обмеження доступу до даних у додатку було розроблено модуль автентифікації та авторизації AuthModule.

У ньому використовується механізм токенів для перевірки прав доступу користувача на певний ресурс (авторизації). Для реалізації механізму токенів було використано підхід використання JSON Web Tokens (JWT), підписаного секретним ключем. Для збільшення надійності системи було обрано використання пари токенів Access (короткостроковий токен, який надає доступ до ресурсів системи) та Refresh (довгостроковий токен, який дозволяє

отримувати нову пару токенів без необхідності автентифікації в системи). Для створення токенів було використано бібліотеку nestjs/jwt [29], яка надає сервіс JwtService для генерації та підписання токенів (рис. 3.21).

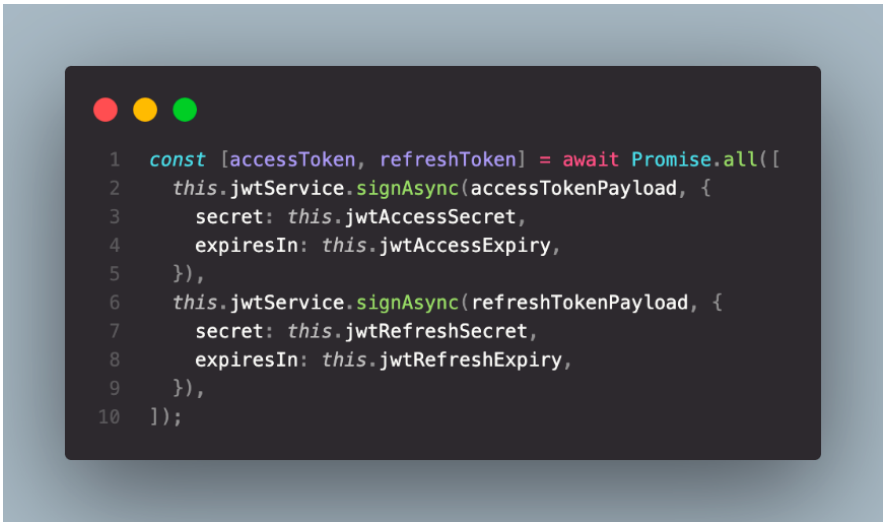


Рисунок 3.21 – Скриншот створення токенів за допомогою JwtService з бібліотеки nestjs/jwt

Процес автентифікації був реалізований через створення ендпоінту для логіну користувача, який перевіряє облікові дані користувача (email, password) та видає токен доступу до ресурсів відповідного акаунту, якщо введені дані є коректними.

Для роботи з Refresh токенами було створено таблицю БД refresh-tokens та модуль RefreshTokensModule, який містить RefreshTokensService, що забезпечує виконання операцій з refresh токенами. Також у даному сервісі виконується контроль за кількістю одночасних сесій користувача (кількістю одночасно валідних пар access та refresh токенів). Якщо при новому логіні кількість сесій виходить за ліміт, то найдавніша сесія стає неактивною (через видалення refresh токену). Сервіс для роботи з токенами наведений на рис. 3.22.

```
@Injectable()
export class RefreshTokensService extends BaseService<RefreshTokenEntity> {
  tokenExpirationTime = this.appConfigService.get<string>(
    'JWT_REFRESH_EXPIRATION_TIME',
  );

  constructor(
    @InjectRepository(RefreshTokenEntity)
    private readonly refreshTokenEntityRepository: Repository<RefreshTokenEntity>,
    private readonly appConfigService: AppConfigService,
    private readonly usersService: UsersService,
  ) {
    super(refreshTokenEntityRepository, refreshTokensServiceErrorMessages);
  }

  async createToken(
    condition: FindOptionsWhere<UserEntity>,
    data: Partial<RefreshTokenEntity>,
  ): Promise<RefreshTokenEntity> {
    const user = await this.usersService.findOne(condition);

    const tokenDuration = ms(this.tokenExpirationTime);
    const currentDateTime = new Date();
    const expiresAt = new Date(currentDateTime.getTime() + tokenDuration);

    const refreshTokenModel: Partial<RefreshTokenEntity> = {
      ...data,
      user,
      expiresAt,
    };

    return this.createOne(refreshTokenModel);
  }

  async deleteExpiredTokens(condition: FindOptionsWhere<RefreshTokenEntity>) {
    const expiredTokens = await this.refreshTokenEntityRepository.find({
      where: condition,
      order: { createdAt: 'DESC' },
      skip: USER_MAX_SESSIONS,
    });

    const tokenToDeleteIds = expiredTokens.map(({ id }) => id);
    if (tokenToDeleteIds.length) {
      await this.refreshTokenEntityRepository.delete(tokenToDeleteIds);
    }
  }
}
```

Рисунок 3.22 – Скриншот коду сервісу RefreshTokensService

Для забезпечення контролю за доступом до ресурсів в додатку було визначено сервіси, що перевіряють валідність токенів – AccessTokenStrategy та RefreshTokenStrategy. Результат виконання операцій в даних сервісів використовується Guard-сервісами AccessTokenGuard та RefreshTokenGuard, які було додано до API ендпоінтів. Вони визначають, що для даного ендпоінта необхідно виконувати перевірку доступу користувача (рис. 3.23). Також в AccessTokenGuard перевіряється не лише валідність токена користувача, а й роль користувача. Відповідно до вимог, у додатку наявні два типи користувачів – Адміністратор та Пацієнт. Для забезпечення обмеження доступу Пацієнтів до ресурсів Адміністратора було створено декоратор Role, який прив’язує до API ендпоінтів ролі, з якими користувачам дозволений доступ до даного ресурсу.

Після цього, AccessTokenGuard має доступ до переліку дозволених ролей та ролі користувача, і забезпечує заборону доступу користувачів до ресурсів, що недоступні для їхньої ролі.

```
@Injectable()
export class AccessTokenGuard extends AuthGuard('jwt') {
  constructor(private readonly reflector: Reflector) {
    super();
  }

  public handleRequest(
    _err: Error,
    user: UserEntity,
    info: Error,
    ctx: ExecutionContext,
  ): UserEntity | any {
    if (_err || info || !user) {
      throw new UnauthorizedException(ErrorMessagesEnum.UNAUTHORIZED);
    }

    const allowedRoles = this.reflector.get<UserRoleEnum[]>(
      'roles',
      ctx.getHandler(),
    );

    if (!allowedRoles || allowedRoles.includes(user.role)) return user;
    throw new ForbiddenException(ErrorMessagesEnum.FORBIDDEN);
  }
}
```

Рисунок 3.23 – Скриншот Guard'у для перевірки доступу користувача до ресурсу з переданим access токеном

Приклад використання AccessTokenGuard наведений на рис. 3.24

```
@UseGuards(AccessTokenGuard)
 @Role(UserRoleEnum.ADMIN)
 @Get(':id')
 findOne(@Param() conditions: IdDto): Promise<DiagnosticModelEntity> {
   return this.diagnosticModelsService.findOneWithVersions(conditions);
 }
```

Рисунок 3.24 – Скриншот використання AccessTokenGuard

Для забезпечення надійності критичних даних користувача (паролю), збереження пароля виконується у захешованому вигляді, щоб навіть при витоку даних, не можна було визначити реальний пароль користувача. Для цього було використано бібліотеку Node.js bcrypt [30], яка надає доступ до функцій хешування. Хешування пароля при створенні та кожній зміні користувача було реалізовано за допомогою тригерів БД та інструментів TypeORM (рис. 3.25).

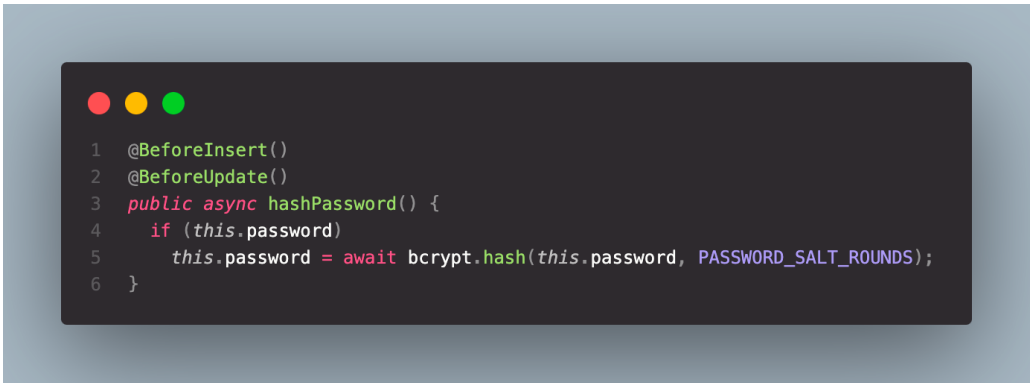


Рисунок 3.25 – Скриншот тригерів для хешування пароля користувача

Також AuthModule забезпечує методи для створення акаунту користувача з роллю пацієнта користувачем з роллю адміністратора, надає доступ користувачам до зміни пароля, оновлення персональних даних та виходу з системи.

3.2.10 Реалізація контролерів для API ендпоінтів

Для забезпечення оброки HTTP-запитів на API ендпоінти було створено контролери серверного додатка. Вони отримують запити від клієнтів, передають ці запити у сервіси для обробки та повертають результати обробки для клієнта. Контролери додатка були визначені через декоратор @Controller, який позначає клас як контролер, а методи через відповідні HTTP-методи @Post, @Get та інші, що позначає методи як обробники запитів на відповідні ендпоінти. Приклад контролеру DiagnosticTypesController наведений на рис. 3.26.

```
@ApiTags('diagnostic-types')
@Controller('diagnostic-types')
@UseGuards(AccessTokenGuard)
@ApiBearerAuth()
@UseInterceptors(ClassSerializerInterceptor)
export class DiagnosticTypesController {
  constructor(
    private readonly diagnosticTypesService: DiagnosticTypesService,
  ) {}

  @Get()
  findAll(): Promise<DiagnosticTypeEntity[]> {
    return this.diagnosticTypesService.findAll();
  }

  @Get(':id')
  findOne(@Param() conditions: IdDto): Promise<DiagnosticTypeEntity> {
    return this.diagnosticTypesService.findOne(conditions);
  }

  @Role(UserRoleEnum.ADMIN)
  @Post()
  createOne(
    @Body() createEntityDto: CreateDiagnosticTypeDto,
  ): Promise<DiagnosticTypeEntity> {
    const model = plainToInstance(DiagnosticTypeEntity, createEntityDto);

    return this.diagnosticTypesService.createOne(model);
  }

  @Role(UserRoleEnum.ADMIN)
  @Patch(':id')
  updateOne(
    @Param() conditions: IdDto,
    @Body() updateEntityDto: UpdateDiagnosticTypeDto,
  ): Promise<DiagnosticTypeEntity> {
    const model = plainToInstance(DiagnosticTypeEntity, updateEntityDto);

    return this.diagnosticTypesService.updateOne(conditions, model);
  }

  @Role(UserRoleEnum.ADMIN)
  @Delete(':id')
  deleteOne(@Param() conditions: IdDto): Promise<DiagnosticTypeEntity> {
    return this.diagnosticTypesService.deleteOne(conditions);
  }
}
```

Рисунок 3.26 – Скриншот коду контролеру DiagnosticTypesController

Вхідні та вихідні дані для ендпоінтів були визначені за допомогою об’єктів передачі даних (Data Transfer Object – DTO), які визначають структуру вхідних та вихідних запитів. Для валідації вхідних даних в DTO були використані декоратори з бібліотеки class-validator [31], що надають можливість виконання валідації полів об’єкта (рис. 3.27).

```
import { ApiProperty } from '@nestjs/swagger';
import { IsNotEmpty, IsOptional, IsString, MaxLength } from 'class-validator';

export class CreateDiagnosticTypeDto {
  @IsNotEmpty()
  @IsString()
  @MaxLength(128)
  @ApiProperty({
    example: 'Melanoma',
    required: true,
    nullable: false,
    maxLength: 128,
  })
  public readonly name: string;

  @IsOptional()
  @IsString()
  @MaxLength(500)
  @ApiProperty({
    example: 'Study of skin photography for the diagnosis of melanoma',
    required: false,
    nullable: true,
    maxLength: 500,
  })
  public readonly description: string;
}
```

Рисунок 3.27 – Скриншот класу об’єкта передачі даних CreateDiagnosticTypeDto

3.2.11 Створення документації для інтеграції з API системи

Для створення документації з інтеграції з API серверу було використано Swagger – набір інструментів для роботи з мовою специфікації HTTP API для визначення структури API ендпоінтів серверу – OpenAPI [32]. Для автогенерації документації було використано бібліотеку @nestjs/swagger [33].

3.3 Розробка сервісу діагностики захворювань за зображеннями за допомогою моделей ШІ

3.3.1 Структура сервісу

Сервіс для виконання діагностики захворювань за зображеннями за допомогою моделей ШІ був розроблений з використанням FastAPI – високопродуктивного фреймворку для розроблення серверних додатків на Python. Головною перевагою даних технологій, окрім широкої підтримки для роботи з моделями Tensorflow, є мінімалістичність, простота, швидкість розгортання та розробки проєктів. Загальна структура проєктів, розроблених за допомогою FastAPI зазвичай складається з наступних компонентів:

- main.py – точка входу в додаток, у якому створюється та налаштовується додаток.
- services – шар додатка, який відповідає за бізнес-логіку, взаємодію зі сторонніми сервісами.
- routers – шар додатка, що визначає обробників маршрутів HTTP-запитів.
- models – визначає моделі бізнес-логіки, з якими працює додаток.
- requirements.txt – файл, у якому міститься список залежностей проєкту.

Оскільки для даного проєкту визначені вимоги включають лише завантаження в пам’ять моделей ШІ, зчитування повідомлень про обробку зображень моделями ШІ з черги повідомлень, отримання доступу до файлу для обробки відповідних зображень за допомогою моделей ШІ, та надсилання в чергу повідомлень інформації про результат обробки, то для проєкту була визначена структура, що наведена на рис. 3.28.

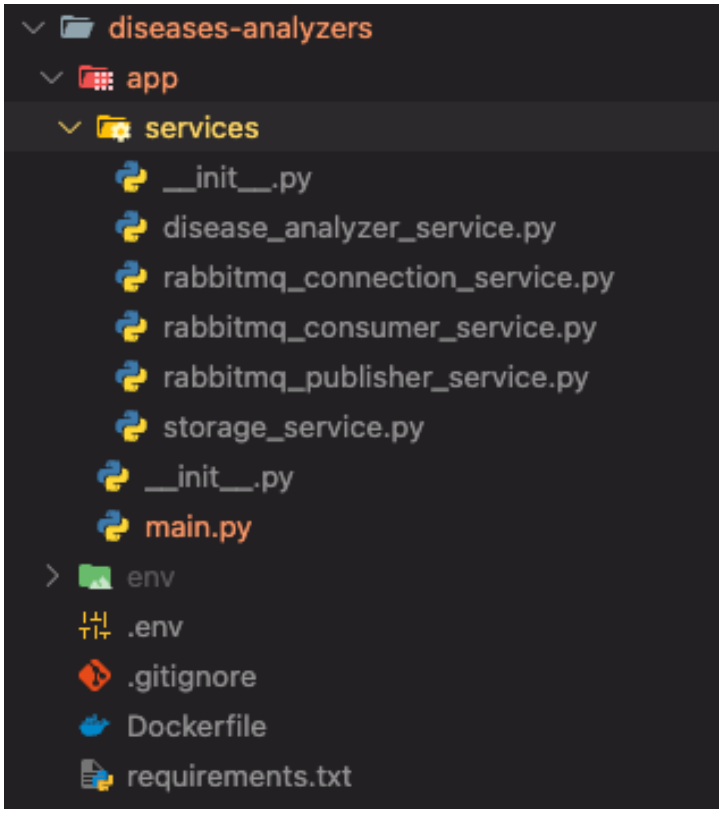


Рисунок 3.28 – Структура сервісу діагностики зображень за допомогою моделей ШІ

Відповідно до рис. 3.28, проєкт буде складатись з файлу точки входу main.py, файлу з переліком моделей requirements.txt, файлом з конфігураціями змінних середовища .env та директорії з сервісами, що містить сервіс для аналізу зображень та взаємодії з моделями ШІ (завантаження в пам’ять), роботу з файловим сховищем MinIO, та сервіси для взаємодії з чергою повідомлень RabbitMQ.

3.3.2 Інтеграція з файловим сховищем MinIO

Для виконання інтеграції з файловим сховищем MinIO в Python-додатку було додано конфігурації для файлового сховища у файл конфігурацій змінних середовища .env (рис. 3.29).

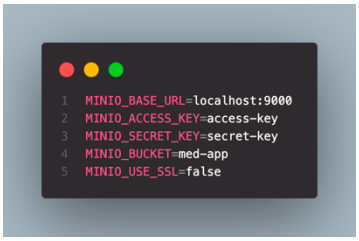


Рисунок 3.29 – Конфігурації для підключення до файлового сховища MinIO в Python-сервісі

Наступним кроком було виконано розробку сервісу для підключення та взаємодії з файловим сховищем MinIO. Для цього було завантажено Python-пакет Minio, що надає інструменти для інтеграції з файловим сховищем. У конструкторі розроблено код підключення до файлового сховища та створення клієнта для виконання операцій з файлами MinIO. Оскільки відповідно до вимог до даного сервісу потрібно лише читання файлу як буферу в пам’ять додатка, було розроблено метод read_file_to_buffer, який за допомогою клієнта MinIO за шляхом файлу на сховищі зчитує його зміст в пам’ять (рис. 3.30).

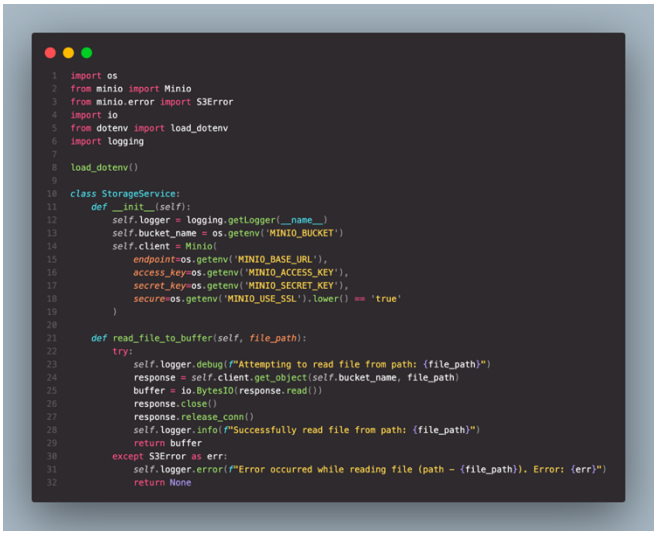


Рисунок 3.30 – Скриншот коду StorageService для виконання операцій з файлами на сховищі MinIO

3.3.3 Налаштування взаємодії з чергою повідомлень RabbitMQ

Для налаштування взаємодії з чергою повідомлень RabbitMQ було додано конфігурації у файл з .env змінними рис. (3.31).

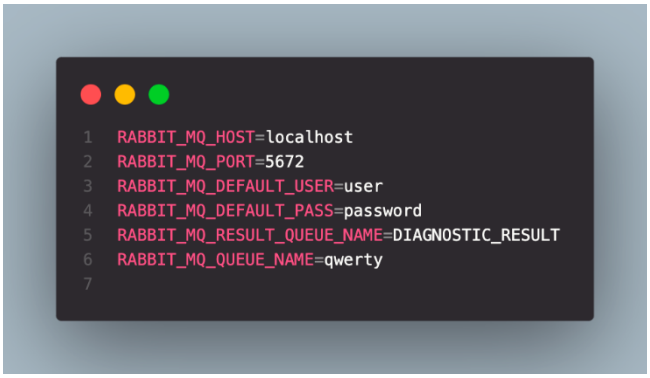


Рисунок 3.31 – Конфігурації для налаштування взаємодії з чергою повідомлень RabbitMQ

Для взаємодії з RabbitMQ було розроблено три сервіси: сервіс для встановлення з'єднання з RabbitMQ (RabbitMQConnectionService), сервіс для надсилання повідомлень (RabbitMQPublisherService) та сервіс для отримання повідомлень (RabbitMQConsumerService). Для роботи з RabbitMQ було використано пакет pika [34], що надає інструменти для інтеграції з чергами повідомлень.

Для більшого спрощення розробки та розуміння обох систем було використано однакові підходи в налаштуванні взаємодії RabbitMQ як в сервісі для обробки API запитів, так і в сервісі для діагностики зображень за допомогою моделей ШІ.

RabbitMQConnectionService забезпечує встановлення зв'язку з RabbitMQ, створення каналу комунікації та створення черг повідомлень (рис. 3.32).

```python
import os
import pika
from dotenv import load_dotenv
import logging

load_dotenv()

class RabbitMQConnectionService:
    def __init__(self):
        self.logger = logging.getLogger(__name__)
        self.rabbitmq_host = os.getenv('RABBIT_MQ_HOST')
        self.rabbitmq_port = os.getenv('RABBIT_MQ_PORT')
        self.rabbitmq_user = os.getenv('RABBIT_MQ_DEFAULT_USER')
        self.rabbitmq_password = os.getenv('RABBIT_MQ_DEFAULT_PASS')
        self.connection = None
        self.channel = None


    def set_up_connection(self):
        self.connection = pika.BlockingConnection(
            pika.ConnectionParameters(
                host=self.rabbitmq_host,
                port=self.rabbitmq_port,
                credentials=pika.PlainCredentials(
                    self.rabbitmq_user,
                    self.rabbitmq_password
                )
            )
        )
        self.channel = self.connection.channel()
        self.channel.basic_qos(prefetch_count=1)

    def assert_queue(self, queue: str):
        self.logger.debug(f"Asserting queue - {queue}")
        self.channel.queue_declare(queue=queue, durable=True)

    def get_channel(self):
        return self.channel

```

Рисунок 3.32 – Скриншот коду сервісу RabbitMQConnectionService

Для виконання операцій надсилання повідомлень в черги повідомлень було розроблено сервіс RabbitMQPublisherService (рис. 3.33)

```python
import json
import logging
import pika
from app.services.rabbitmq_connection_service import RabbitMQConnectionService

class RabbitMQPublisherService:
    def __init__(self, rabbitmq_connection_service: RabbitMQConnectionService):
        self.logger = logging.getLogger(__name__)
        self.rabbitmq_connection_service = rabbitmq_connection_service

    def add_message_to_queue(self, queue: str, message_data: dict):
        channel = self.rabbitmq_connection_service.get_channel()
        self.rabbitmq_connection_service.assert_queue(queue)
        self.logger.debug(f"Sending message to queue: {queue}. Data: {message_data}")
        channel.basic_publish(
            exchange='',
            routing_key=queue,
            body=self.convert_data(message_data),
            properties=pika.BasicProperties(delivery_mode=2)
        )
        print(f"Message sent to queue ({queue}) with task data {message_data}")

    def convert_data(self, data: dict) -> bytes:
        return json.dumps(data).encode('utf-8')

```

Рисунок 3.33 – Скриншот коду сервісу RabbitMQPublisherService

Для читання повідомлень з черги та обробки повідомлень був розроблений сервіс RabbitMQConsumerService (рис. 3.34). У ньому також було визначено метод для обробки запитів для діагностики зображень моделями ШІ (handle_analyze_disease_message, у якому виконується діагностика зображення за допомогою моделі ШІ, отримання ймовірності захворювання та відправлення повідомлення про результат дослідження в чергу повідомлень за допомогою publisher'a).

```python
import json
import os
import logging
from app.services.rabbitmq_connection_service import RabbitMQConnectionService
from app.services.rabbitmq_publisher_service import RabbitMQPublisherService
from app.services.storage_service import StorageService
from app.services.disease_analyzer_service import DiseaseAnalyzerService

class RabbitMQConsumerService:
    def __init__(self, rabbitmq_connection_service: RabbitMQConnectionService,
                 rabbitmq_publisher_service: RabbitMQPublisherService, storage_service: StorageService,
                 disease_analyzer_service: DiseaseAnalyzerService):
        self.logger = logging.getLogger(__name__)
        self.rabbitmq_connection_service = rabbitmq_connection_service
        self.rabbitmq_publisher_service = rabbitmq_publisher_service
        self.storage_service = storage_service
        self.disease_analyzer_service = disease_analyzer_service
        self.queue_name = os.getenv('RABBIT_MQ_QUEUE_NAME')
        self.result_queue_name = os.getenv('RABBIT_MQ_RESULT_QUEUE_NAME')
        self.rabbitmq_connection_service.set_up_connection()
        channel = self.rabbitmq_connection_service.get_channel()
        self.set_up_analyze_disease_handler(channel)


    def set_up_analyze_disease_handler(self, channel):
        self.rabbitmq_connection_service.assert_queue(self.queue_name)
        self.logger.debug(f"Starting to consume from queue: {self.queue_name}")
        channel.basic_consume(
            queue=self.queue_name,
            on_message_callback=self.handle_analyze_disease_message,
            auto_ack=False,
        )
        channel.start_consuming()

    def handle_analyze_disease_message(self, ch, method, properties, body):
        self.logger.debug(f"Received message: {body}")
        data = self.get_message_data(body)
        try:
            self.logger.debug(f"Data: {data}")
            self.handle_analyze_disease_result(data)
            ch.basic_ack(delivery_tag=method.delivery_tag)
        except Exception as err:
            self.handle_error(data, ch, method, body, str(err))

    def get_message_data(self, body):
        return json.loads(body)

    def handle_analyze_disease_result(self, data):
        result_id = data.get('resultId')
        image_path_in_storage = data.get('imagePathInStorage')
        if not result_id or not image_path_in_storage:
            raise ValueError("Missing required data in message")
        buffer = self.storage_service.read_file_to_buffer(image_path_in_storage)
        if not buffer:
            raise ValueError(f"File by specified path (${image_path_in_storage}) not found!")
        diseaseProbability = self.disease_analyzer_service.analyze_disease(buffer)
        result_data = {
            'resultId': result_id,
            'status': 'COMPLETED',
            'diseaseProbability': diseaseProbability,
        }
        self.rabbitmq_publisher_service.add_message_to_queue(self.result_queue_name, result_data)

    def handle_error(self, data, ch, method, body, err_message):
        self.logger.debug(f"Finishing validation with FAIL. Message body: {body}. Reason: {err_message}")
        result_id = data.get('resultId', 'unknown')
        failed_message_data = {
                'resultId': result_id,
                'status': 'FAILED'
            }
        self.rabbitmq_publisher_service.add_message_to_queue(self.result_queue_name, failed_message_data)
        ch.basic_ack(delivery_tag=method.delivery_tag)
```

Рисунок 3.34 – Скриншот коду сервісу RabbitMQConsumerService

3.3.4 Аналіз зображень за допомогою моделей ШІ

Для виконання діагностики зображень за допомогою моделей ШІ був розроблений сервіс DiseaseAnalyzerService. У ньому визначено.2 метода: load_model для завантаження моделі в пам’ять для подальшого використання для аналізу, та analyze_disease, за допомогою якого відбувається виконання діагностики та отримання результату у вигляді числового значення ймовірності захворювання (рис. 3.35). Для можливості налаштування багатьох екземплярів сервісу для діагностики, щоб вони обробляли різні типи запитів, сервіс вичитує модель з MinIO за шляхом до файлу, що відповідає певній версії моделі, що буде використовуватись для аналізу.

```python
class DiseaseAnalyzerService:
    def __init__(self, storage_service: StorageService):
        self.logger = logging.getLogger(__name__)
        self.storage_service = storage_service
        self.model = self.load_model(os.getenv('MODEL_PATH'))

    def load_model(self, model_path):
        try:
            self.logger.debug(f"Loading model from: {model_path}")
            buffer = self.storage_service.read_file_to_buffer(model_path)
            with tempfile.NamedTemporaryFile(delete=False) as temp_model_file:
                temp_model_file.write(buffer.getvalue())
                temp_model_path = temp_model_file.name
            model = tf.keras.models.load_model(temp_model_path)
            self.logger.info("Model loaded successfully")
            os.remove(temp_model_path)
            return model
        except Exception as e:
            self.logger.error(f"Failed to load model: {e}")
            raise e

    def analyze_disease(self, image_buffer):
        try:
            image = tf.io.decode_image(image_buffer.getvalue(), channels=3)
            image = tf.image.resize(image, [224, 224])
            image = tf.expand_dims(image, axis=0)
            prediction = self.model.predict(image)
            self.logger.debug(f"PREDICTION: {prediction}")
            disease_probability = prediction[0][0]
            self.logger.debug(f"Disease probability: {disease_probability}")
            return disease_probability
        except Exception as e:
            self.logger.error(f"Failed to analyze disease: {e}")
            raise e
```

Рисунок 3.35 – Скриншот коду сервісу RabbitMQConsumerService

ВИСНОВОК ДО РОЗДІЛУ 3

У даному розділі було описано та виконано розробку серверної частини системи для спільного використання моделей штучного інтелекту для діагностування захворювань, а саме реалізовано серверний додаток сервісу для обробки API запитів від клієнтів, а також серверний додаток для сервісу діагностики захворювань за зображеннями за допомогою моделей ШІ.

Для сервісу для обробки API запитів було розроблено структуру серверного додатка, виконано налаштування з'єднання з базою даних у додатку за допомогою ORM TypeORM. Після цих кроків було створено структуру бази даних за допомогою інструментів TypeORM. Було виконано створення таблиць для роботи з користувачами, їх даними для авторизації та автентифікації, таблиці для збереження даних про файли, про типи діагностик, моделі для діагностики, їхні версії, а також реалізовано таблиці для збереження інформації про діагностики користувачів та результати діагностик та налаштування зв'язків між таблицями БД. Також було згенеровано початкову міграцію структури бази даних, для можливості подальшої зміни структури бази даних без впливу на наявні дані в БД. Також було виконано інтеграцію з файловим сховищем MinIO, налаштовано сервіс для виконання операцій читання, запису та видалення файлів з файлового сховища. Для комунікації між сервісами, для сервісу обробки API запитів було налаштовано взаємодію з чергою повідомлень RabbitMQ, налаштовані сервіси для встановлення з'єднання з чергою, керування каналами та чергами повідомлень, а також сервіси для відправлення повідомлень в черги та читання повідомлень з черг RabbitMQ. Були реалізовані основні компоненти бізнес-логіки системи – модулі для роботи з моделями діагностики, їхніми типами та версіями, збереження даних про діагностики та результати діагностик в БД. Також було створено модуль автентифікації та авторизації для керування доступом користувачів до ресурсів системи та

забезпеченням обмеження доступу до персональних даних. Були реалізовані контролери для обробки запитів на API ендпоінти та описано схему маршрутів, вхідних та вихідних параметрів для інтеграції клієнтів з API сервером за допомогою інструментів Swagger.

Для сервісу діагностики зображень користувачів за допомогою моделей ШІ було розроблено серверний додаток на Python з використанням фреймворку FastAPI, виконано інтеграцію з файловим сховищем MinIO, для отримання доступу до файлів моделей ШІ та зображень для діагностики, виконано розробку сервісів для комунікації з чергою повідомлень RabbitMQ, а саме сервіси для встановлення з'єднання та створення каналів зв'язку, черг повідомлень, та сервіси для відправлення повідомлень про результати діагностики захворювань в чергу, та сервіс для отримання й оброки повідомлень з черги RabbitMQ. Після цього було розроблено сервіс для роботи з моделями ШІ, а саме для завантаження моделей в пам'ять додатка та аналізу зображень за допомогою завантажених моделей ШІ

У результаті виконання роботи даного розділу було розроблено сервіс для обробки API запитів від клієнтів та сервіс діагностики захворювань за зображеннями за допомогою моделей ШІ, а також реалізовано асинхронну комунікацію між сервісами за допомогою взаємодії з чергою повідомлень RabbitMQ. Розроблена система забезпечує виконання визначених у попередніх розділах функціональних та нефункціональних вимог до системи.

РОЗДІЛ 4. РОЗГОРТАННЯ ТА АНАЛІЗ РОЗРОБЛЕНОЇ СИСТЕМИ

У попередніх розділах було проаналізовано предметну область, сформовано функціональні та нефункціональні вимоги та розроблено систему. У цьому розділі буде розглянуто розгортання системи, документацію з інтеграції клієнтів з API системи, представлено виконання основних сценаріїв системи, виконано аналіз розробленої системи для порівняння розробленої системи з розглянутими системами-аналогами та визначено способи покращення розробленої системи.

4.1 Розгортання компонентів системи

Розгортання розроблених компонентів системи, а також бази даних PostgreSQL, черги повідомлень RabbitMQ та файлового сховища MinIO було виконано в Docker-контейнерах, що забезпечує ізоляцію сервісів від середовища виконання, спрощує розгортання системи та надає можливість однаково запускати контейнери незалежно від host-середовища виконання. Розгортання та налаштування сервісів виконувалось за допомогою docker-compose [35] – інструменту для налаштування, запуску та керування багатьма контейнерами системи. Було розроблено схему розгортання компонентів системи (рис. 4.1)

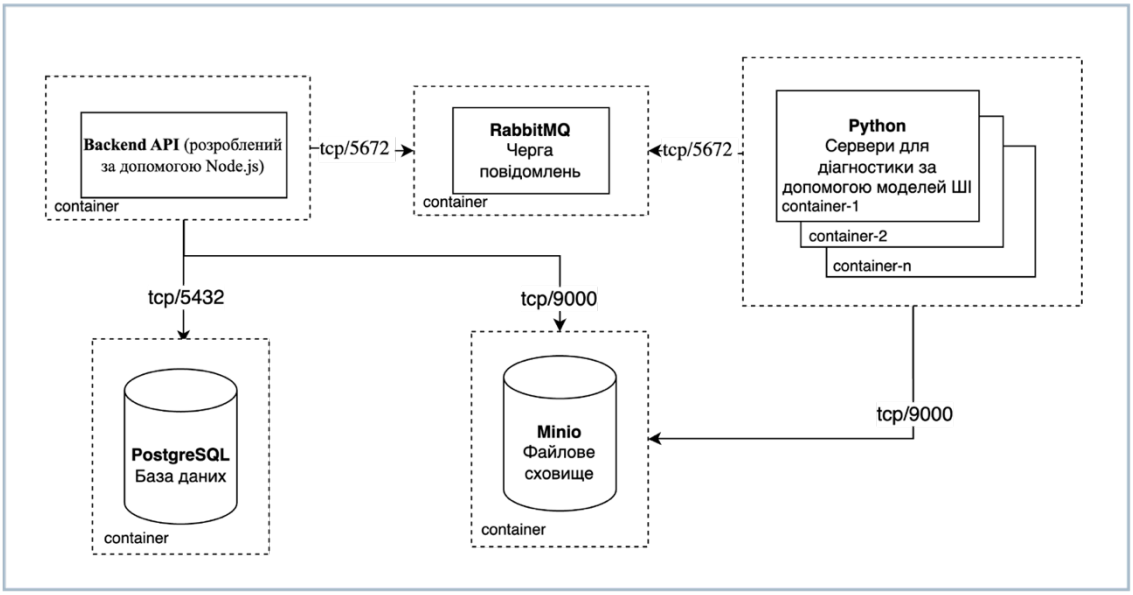


Рисунок 4.1 – Схема розгортання компонентів системи

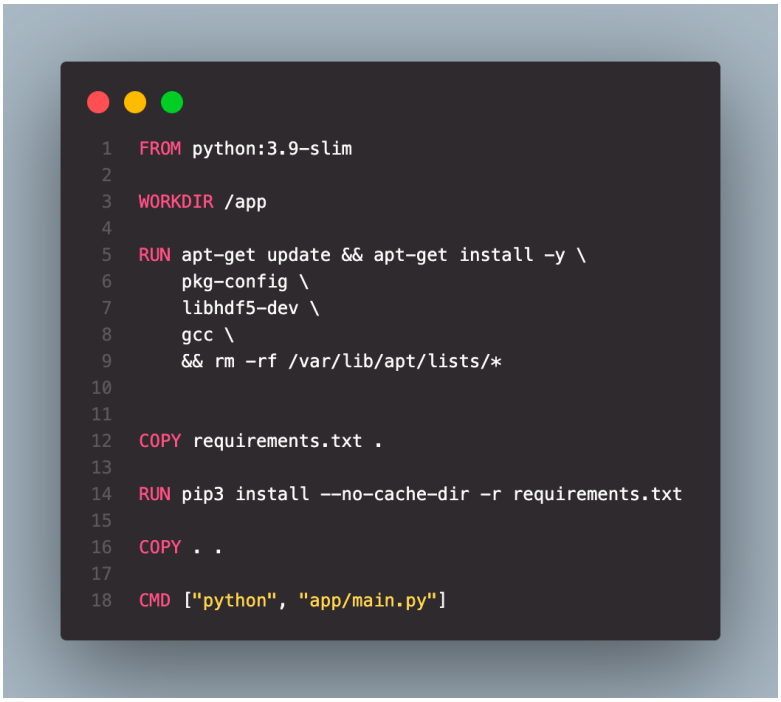
Для розгортання БД PostgreSQL, файлового сховища MinIO та черги повідомлень RabbitMQ було використано офіційні Docker-образи з DockerHub [36].

Для розгортання сервісу для обробки запитів клієнтів до API системи було створено Dockerfile (рис. 4.2) для розгортання Node.js додатку в контейнері.

```
# build

FROM node:20-alpine AS build

WORKDIR /app

COPY package*.json ./

RUN npm ci

COPY . .

RUN npm run build

# run

FROM node:20-alpine

RUN apk add --no-cache tini

ENTRYPOINT ["/sbin/tini", "--"]

WORKDIR /app

COPY --from=build /app/*.env /app/package*.json ./

RUN npm ci --production

COPY --from=build /app/dist ./src/

CMD ["node", "./src/main.js"]
```

Рисунок 4.2 – Скриншот налаштування контейнера серверного додатка для обробки API запитів від клієнтів

Для розгортання сервіса для діагностики хвороб за зображеннями за допомогою моделей ШІ було створено Dockerfile (рис. 4.3) для розгортання Python-додатку в контейнері.



```
FROM python:3.9-slim

WORKDIR /app

RUN apt-get update && apt-get install -y \
    pkg-config \
    libhdf5-dev \
    gcc \
    && rm -rf /var/lib/apt/lists/*


COPY requirements.txt .

RUN pip3 install --no-cache-dir -r requirements.txt

COPY . .

CMD ["python", "app/main.py"]
```

Рисунок 4.3 – Скриншот налаштування контейнера серверного додатка для діагностики хвороб за зображеннями за допомогою моделей ШІ

Наступним кроком, використовуючи оголошені файли для конфігурації Docker-контейнерів для сервісів додатка, було розроблено docker-compose файл, у якому налаштовуються сервіси компонентів системи, зв'язками між ними, порядок розгортання, призначається використання спільної мережі для комунікації компонентів у приватній мережі без доступу ззовні, призначаються volumes для збереження файлів, що не будуть втрачені під час перезапуску контейнерів, для забезпечення постійного збереження файлів сервісів (рис. 4.4).

```yaml
version: '3.9'

services:
  api-service:
    build:
      context: ./services/api
      dockerfile: Dockerfile
    container_name: api-service
    ports:
      - 8080:8080
    depends_on:
      - postgres
      - minio
      - rabbitmq
    networks:
      - network

  diseases-analyzers-service:
    build:
      context: ./services/diseases-analyzers
      dockerfile: Dockerfile
    container_name: diseases-analyzer-1-service
    ports:
      - 8000:8000
    depends_on:
      - postgres
      - minio
      - rabbitmq
    networks:
      - network

  postgres:
    image: postgres:14.1-alpine
    container_name: postgres
    restart: always
    environment:
      - POSTGRES_USER=med-app
      - POSTGRES_PASSWORD=postgres
      - POSTGRES_DB=med-app
    ports:
      - 5432:5432
    networks:
      - network
    volumes:
      - postgres_data:/var/lib/postgresql/data
    healthcheck:
      test: ['CMD-SHELL', 'pg_isready -U med-app']
      interval: 10s
      timeout: 5s
      retries: 5
      start_period: 1m

  minio:
    image: minio/minio
    container_name: minio
    restart: unless-stopped
    ports:
      - '9000:9000'
      - '9001:9001' # port for monitoring (web interface)
    networks:
      - network
    environment:
      MINIO_ROOT_USER: access-key
      MINIO_ROOT_PASSWORD: secret-key
    command: server /data --console-address ":9001"
    volumes:
      - minio_data:/data

  rabbitmq:
    image: rabbitmq:3-management
    container_name: rabbitmq
    ports:
      - '5672:5672'
      - '15672:15672' # port for monitoring (web interface)
    environment:
      RABBITMQ_DEFAULT_USER: user
      RABBITMQ_DEFAULT_PASS: password
    volumes:
      - rabbitmq_data:/var/lib/rabbitmq
    networks:
      - network

networks:
  network:
    driver: bridge

volumes:
  postgres_data:
  minio_data:
  rabbitmq_data:
```

Рисунок 4.4 – Скриншот налаштування розгортання компонентів в контейнерах за допомогою docker-compose

Для збереження файлів у файловому сховищі MinIO за допомогою вебклієнту для адміністрування сервісу було створено та налаштовано bucket, у якому будуть зберігатись файли (рис. 4.5)

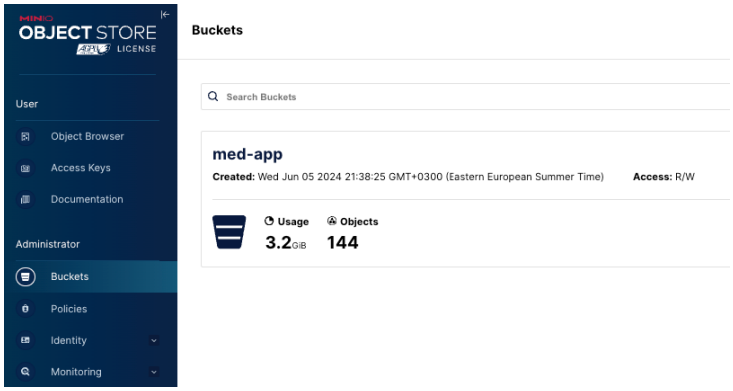


Рисунок 4.5 – Скриншот налаштування bucket’у для файлового сховища MinIO

4.2 Огляд документації з інтеграції клієнтів з API системи

У третьому розділі було описано використання інструментів для створення документації для інтеграції з API системи. У результаті було створено відповідну документацію, в якій відображено список всіх API ендпоінтів, що обробляються серверним додатком, описано типи вхідних даних для запитів на ендпоінти, типи результатів, що будуть повернуті у відповідь, а також позначено необхідність авторизації, ролі, для яких доступні ресурси. Фрагменти документації наведені на рис. 4.6 та рис. 4.7.

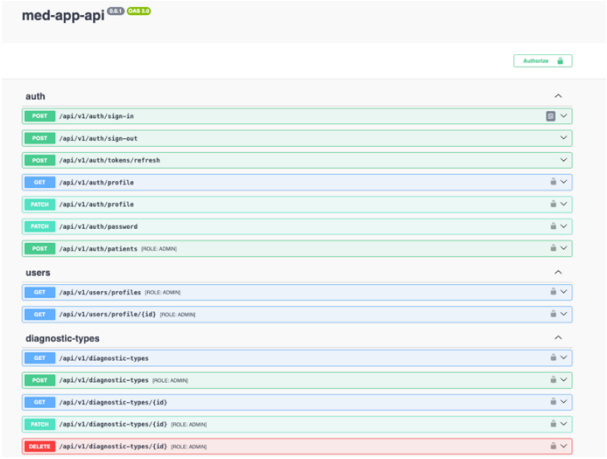


Рисунок 4.6 – Фрагмент документації для ендпоінтів модуля автентифікації та авторизації, користувачів та типів діагностик

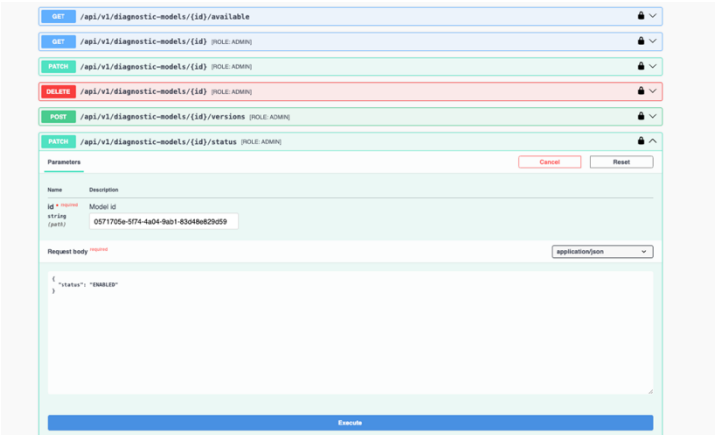


Рисунок 4.7 – Фрагмент документації для ендпоінтів модуля моделей діагностик

Також ще однією значною перевагою розробленої документації є те, що вона є інтерактивною, тобто розробники можуть виконати запити на сторінці документації та протестувати запити до API ендпоінтів. Приклади наведені на рис. 4.8, рис. 4.9 та рис. 4.10.

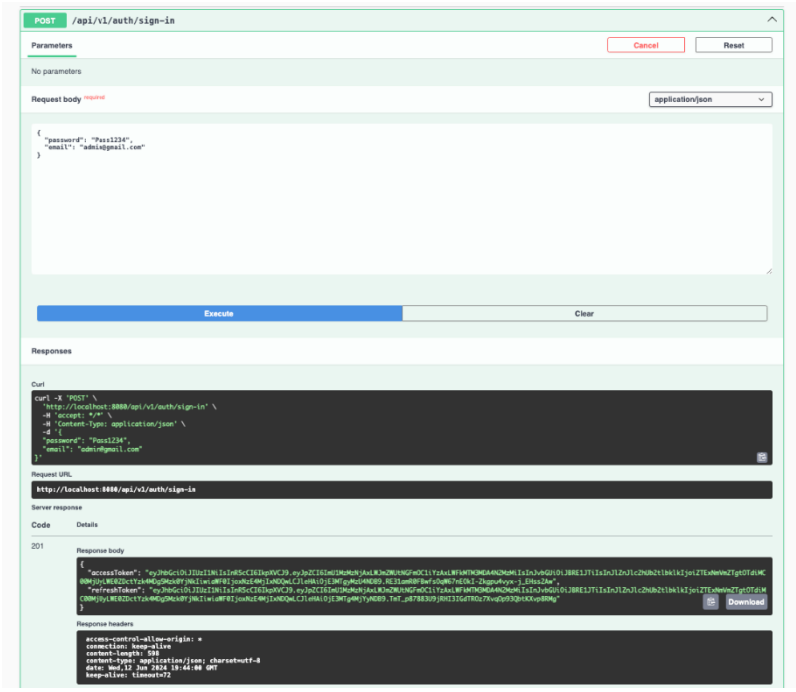


Рисунок 4.8 – Скриншот виконання запиту на сторінці документації для логіну

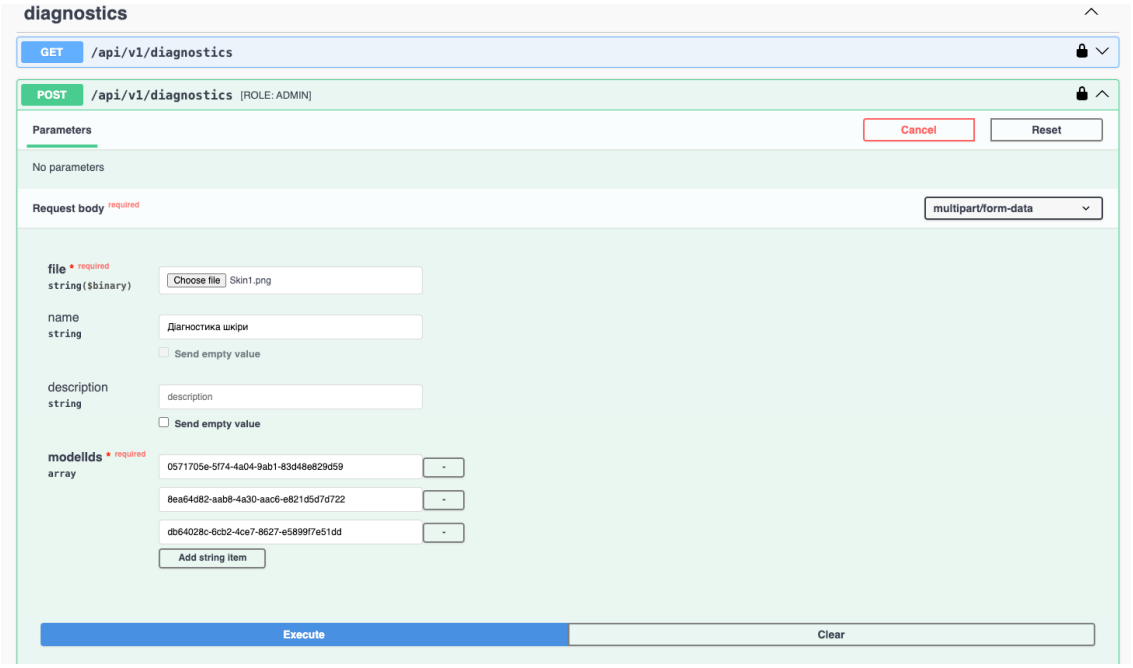


Рисунок 4.9 – Скриншот виконання запиту на сторінці документації для створення діагностики захворювання за зображенням

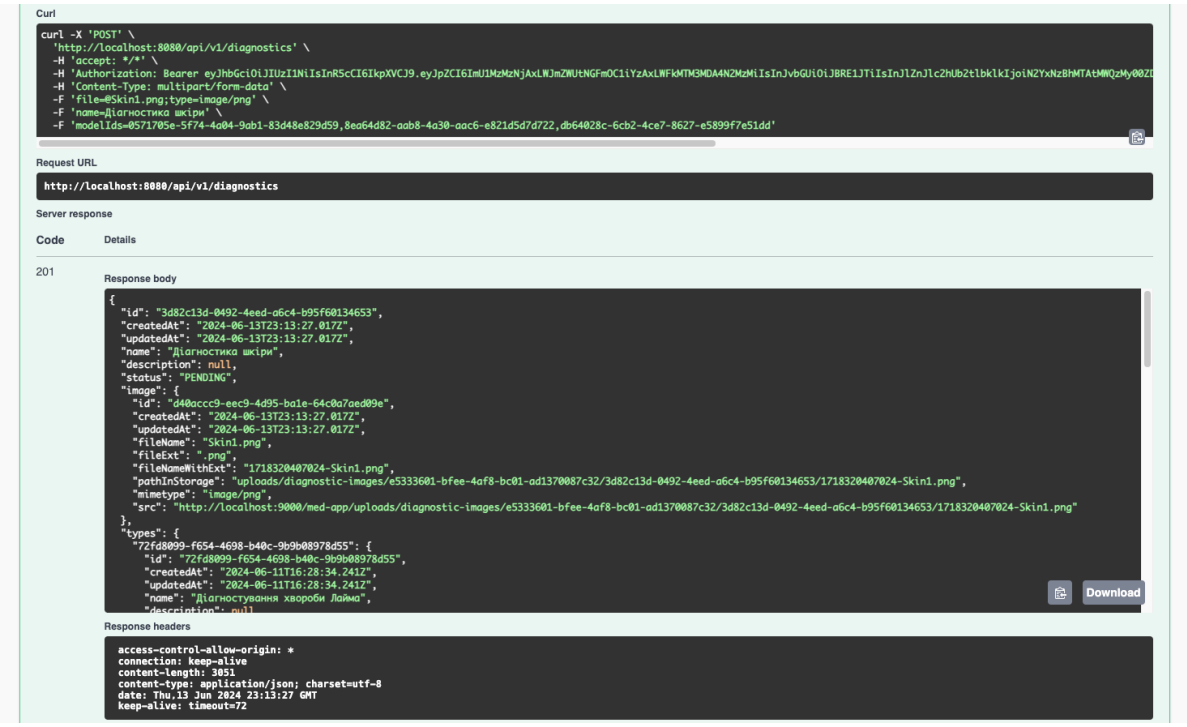


Рисунок 4.10 – Скриншот результату запиту на сторінці документації для створення діагностики захворювання за зображенням

Таким чином, у результаті розробки документації клієнти мають повний перелік ендпоінтів та даних для інтеграції з серверним API, що значно спрощує розробку запитів клієнтів до системи.

4.3 Представлення виконання основних сценаріїв системи

У другому розділі було визначено перелік функціональних вимог до системи, відповідно до яких було реалізовано логіку роботи системи. У результаті було розроблено API ендпоінти та сервіси для обробки сценаріїв системи, які забезпечують виконання цих вимог:

- Вхід в систему – ендпоінт POST /auth/sign-in (рис. 4.11). В результаті виконання запиту користувач отримує пару access/refres токенів, за допомогою яких він може отримувати доступ до захищених ресурсів.

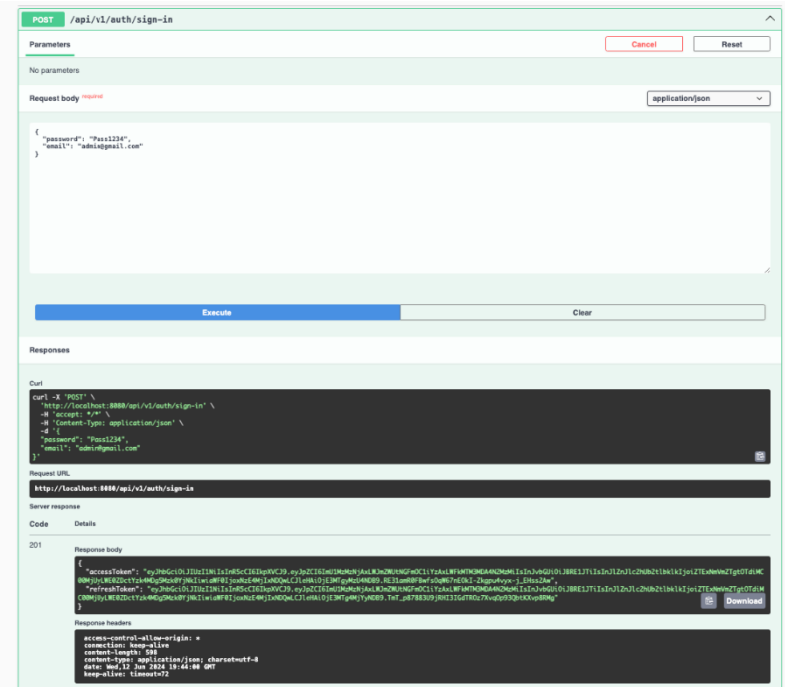


Рисунок 4.11 – Скриншот запиту до ендпоінту POST /auth/sign-in

- Створення акаунтів користувачів – ендпоінт POST /auth/patients (рис. 4.12). У результаті виконання запиту створюється акаунт нового користувача з тимчасовим паролем, користувач має доступ до системи та може змінити пароль після першого входу в систему.

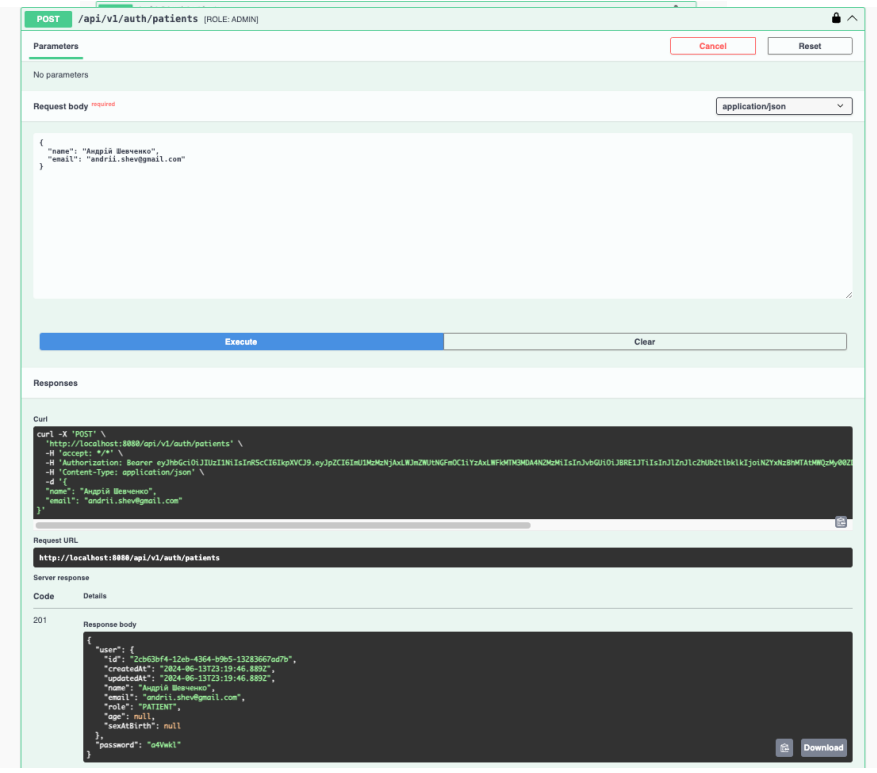


Рисунок 4.12 – Скриншот запиту до ендпоінту POST /auth/patients

- Перегляд доступних типів діагностики – ендпоінт GET /diagnostic-types (рис. 4.13).

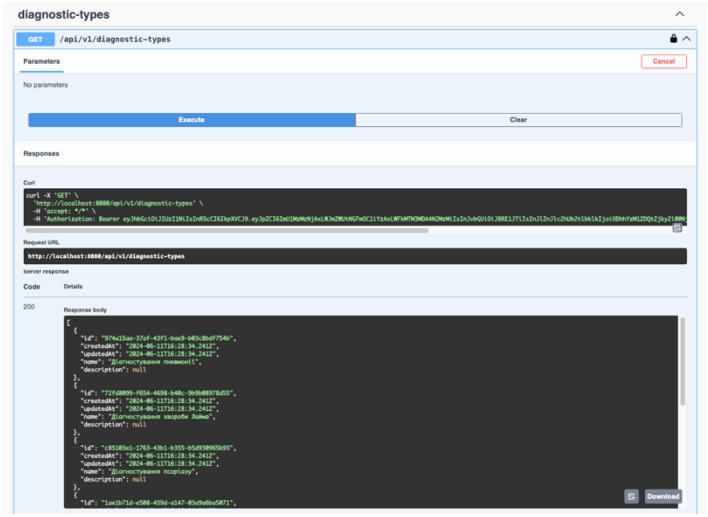


Рисунок 4.13 – Скриншот запиту до ендпоінту GET /diagnostic-types

- Керування доступними типами діагностики – ендпоінти POST /diagnostic-types (рис. 4.14), PATCH /diagnostic-types/{id}, DELETE /diagnostic-types/{id}.

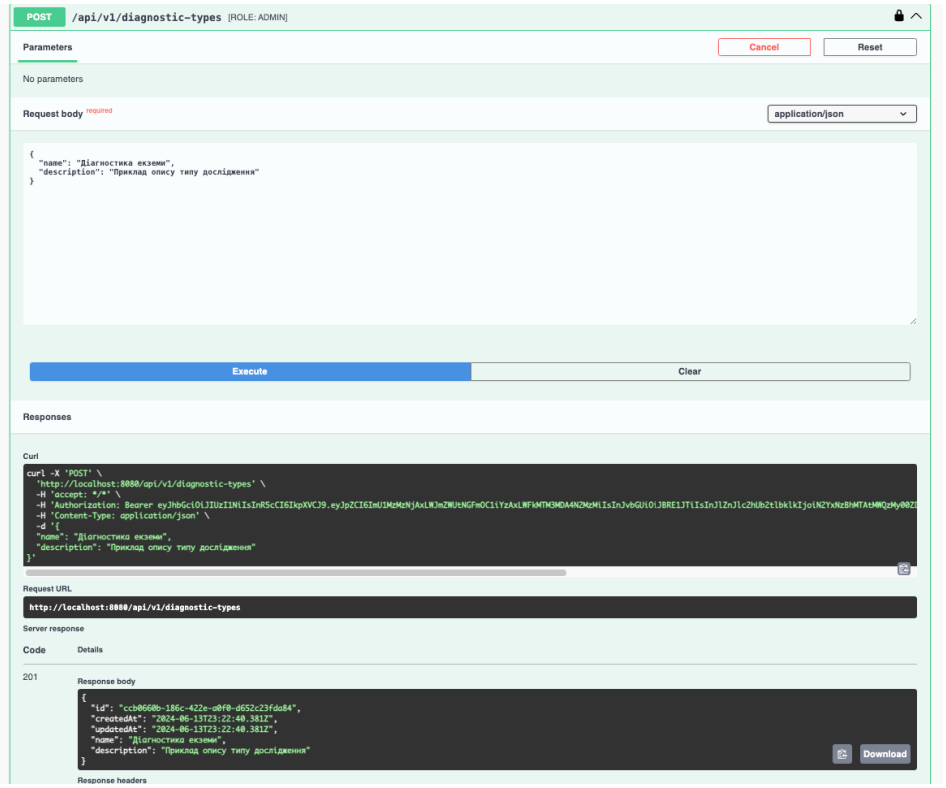
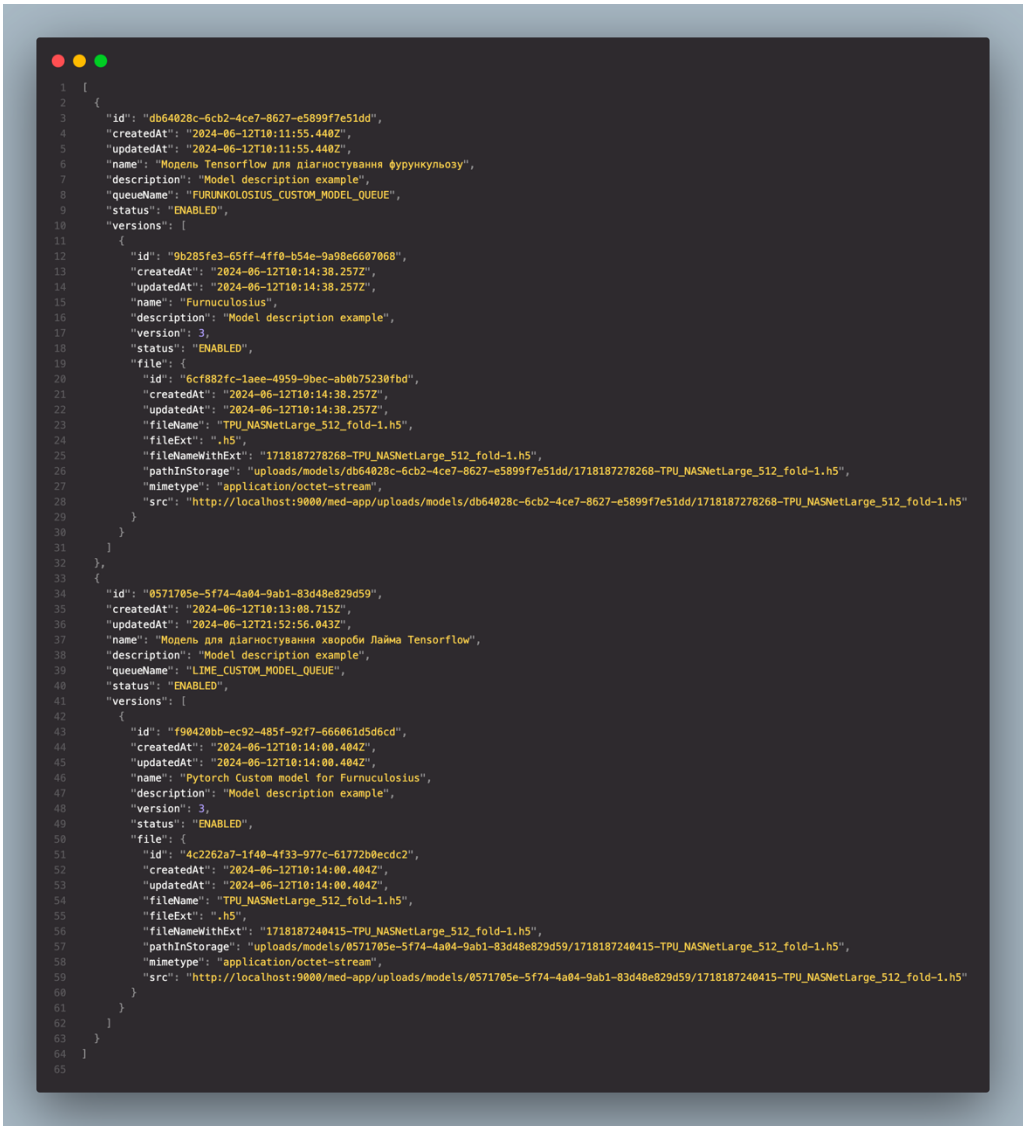


Рисунок 4.14 – Скриншот запиту до ендпоінту POST /diagnostic-types

- Перегляд доступних моделей ШІ для діагностики – ендпоінт GET /diagnostic-models/available (рис. 4.15). У даному ендпоінті обираються лише увімкнені моделі, у яких є хоча б одна увімкнена версія моделі.



```
[
  {
    "id": "db64028c-6cb2-4ce7-8627-e5899f7e51dd",
    "createdAt": "2024-06-12T10:11:55.440Z",
    "updatedAt": "2024-06-12T10:11:55.440Z",
    "name": "Модель Tensorflow для діагностування фурункульозу",
    "description": "Model description example",
    "queueName": "FURUNKOLOSIUS_CUSTOM_MODEL_QUEUE",
    "status": "ENABLED",
    "versions": [
      {
        "id": "9b285fe3-65ff-4ff0-b54e-9a98e6607068",
        "createdAt": "2024-06-12T10:14:38.257Z",
        "updatedAt": "2024-06-12T10:14:38.257Z",
        "name": "Furnuculosius",
        "description": "Model description example",
        "version": 3,
        "status": "ENABLED",
        "file": {
          "id": "6cf882fc-1aee-4959-9bec-ab0b75230fbd",
          "createdAt": "2024-06-12T10:14:38.257Z",
          "updatedAt": "2024-06-12T10:14:38.257Z",
          "fileName": "TPU_NASNetLarge_512_fold-1.h5",
          "fileExt": ".h5",
          "fileNameWithExt": "1718187278268-TPU_NASNetLarge_512_fold-1.h5",
          "pathInStorage": "uploads/models/db64028c-6cb2-4ce7-8627-e5899f7e51dd/1718187278268-TPU_NASNetLarge_512_fold-1.h5",
          "mimetype": "application/octet-stream",
          "src": "http://localhost:9000/med-app/uploads/models/db64028c-6cb2-4ce7-8627-e5899f7e51dd/1718187278268-TPU_NASNetLarge_512_fold-1.h5"
        }
      }
    ]
  },
  {
    "id": "0571705e-5f74-4a04-9ab1-83d48e829d59",
    "createdAt": "2024-06-12T10:13:08.715Z",
    "updatedAt": "2024-06-12T21:52:56.043Z",
    "name": "Модель для діагностування хвороби Лайма Tensorflow",
    "description": "Model description example",
    "queueName": "LIME_CUSTOM_MODEL_QUEUE",
    "status": "ENABLED",
    "versions": [
      {
        "id": "f90420bb-ec92-485f-92f7-666061d5d6cd",
        "createdAt": "2024-06-12T10:14:00.404Z",
        "updatedAt": "2024-06-12T10:14:00.404Z",
        "name": "Pytorch Custom model for Furnuculosius",
        "description": "Model description example",
        "version": 3,
        "status": "ENABLED",
        "file": {
          "id": "4c2262a7-1f40-4f33-977c-61772b0ecdc2",
          "createdAt": "2024-06-12T10:14:00.404Z",
          "updatedAt": "2024-06-12T10:14:00.404Z",
          "fileName": "TPU_NASNetLarge_512_fold-1.h5",
          "fileExt": ".h5",
          "fileNameWithExt": "1718187240415-TPU_NASNetLarge_512_fold-1.h5",
          "pathInStorage": "uploads/models/0571705e-5f74-4a04-9ab1-83d48e829d59/1718187240415-TPU_NASNetLarge_512_fold-1.h5",
          "mimetype": "application/octet-stream",
          "src": "http://localhost:9000/med-app/uploads/models/0571705e-5f74-4a04-9ab1-83d48e829d59/1718187240415-TPU_NASNetLarge_512_fold-1.h5"
        }
      }
    ]
  }
]
```


Рисунок 4.15 – Скриншот тіла відповіді сервера на запит до ендпоінту GET /diagnostic-models/available

- Керування моделями ШІ – ендпоінти POST /diagnostic-models (рис. 4.16), PATCH /diagnostic-models /{id}, DELETE /diagnostic-models /{id}.

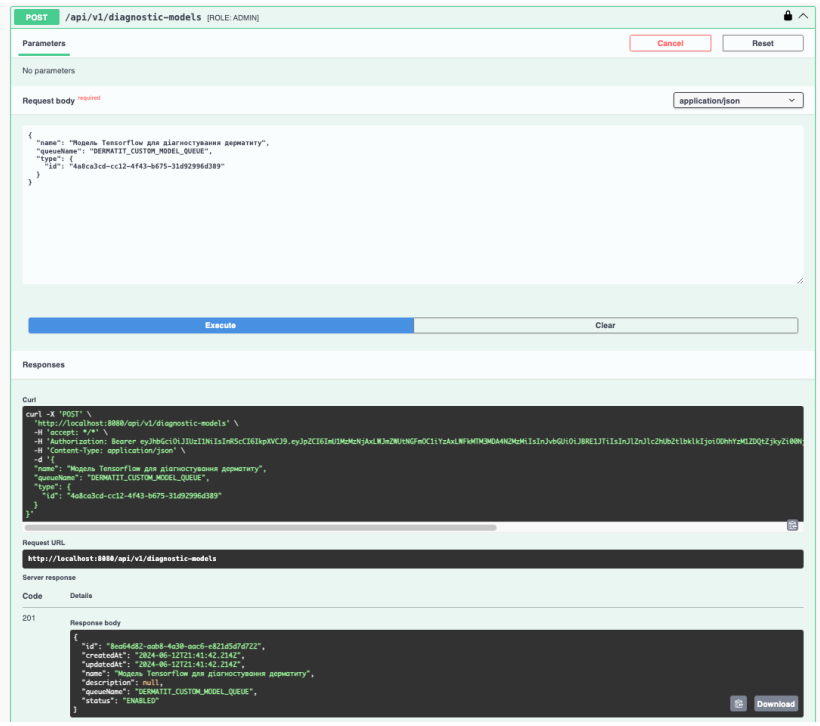


Рисунок 4.16 – Скриншот запита до ендпоінту POST /diagnostic-models

- Перегляд версій моделей ШІ – ендпоінт GET /diagnostic-models/{id} (рис. 4.17). У результаті повертаються дані про модель та перелік всіх версій моделей в усіх статусах.

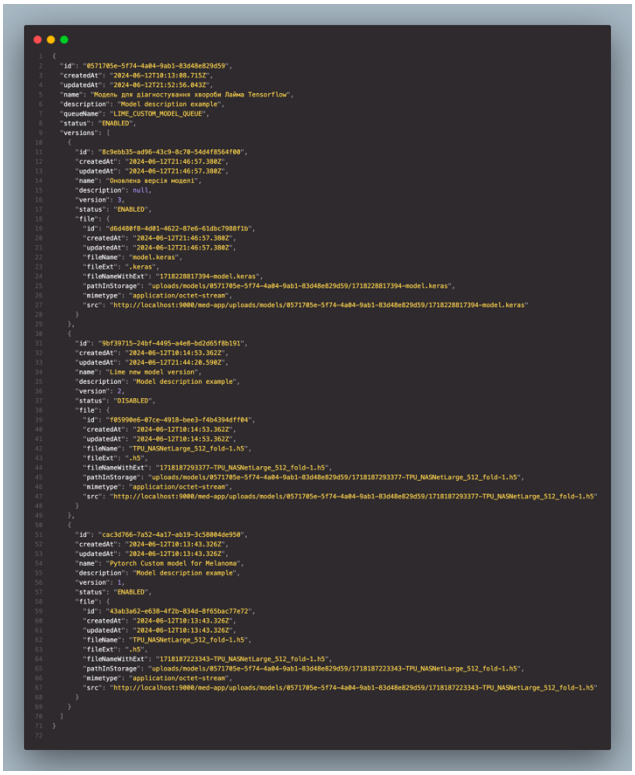


Рисунок 4.17 – Скриншот тіла відповіді запита до ендпоінту GET /diagnostic-models/{id}

- Керування версіями моделей ШІ – ендпоінт POST /diagnostic-models/{id}/versions (рис. 4.18) для додавання нових версій моделі (завантажується файл моделі, та метадані моделі – назва, опис). У результаті додається нова активна версія моделі.

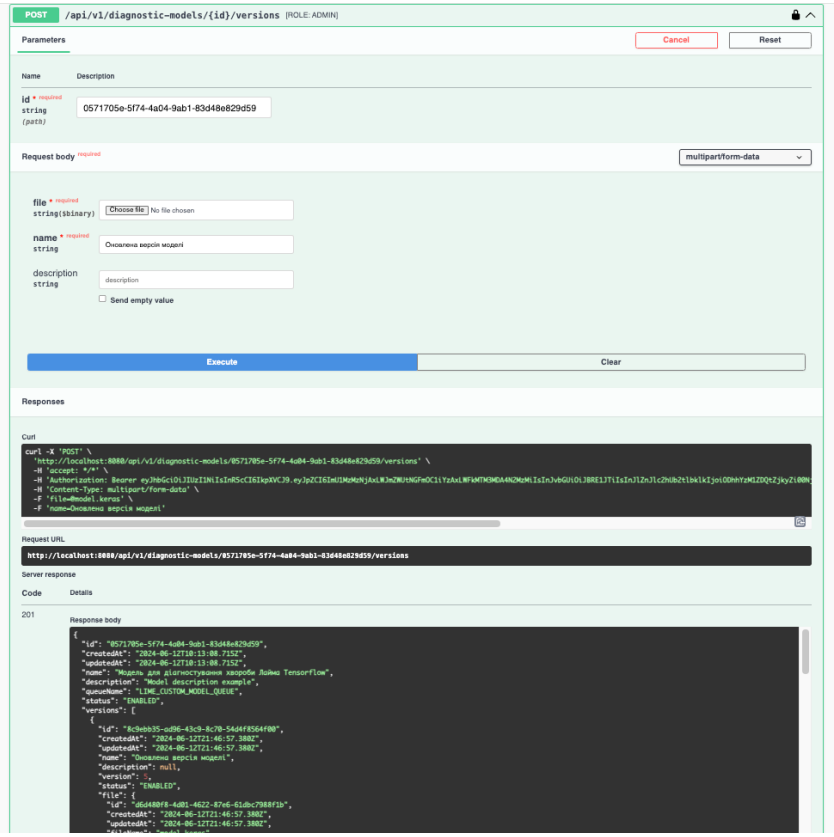


Рисунок 4.18 – Скриншот запита до ендпоінту POST /diagnostic-models/{id}/versions

- Вимкнення/увімкнення доступу до моделі ШІ – ендпоінт PATCH /diagnostic-models/{id}/status (рис. 4.19). Моделі в статусі DISABLED є вимкненими та недоступні для діагностики, моделі в статусі ENABLED є увімкненими та доступні для виконання діагностики.

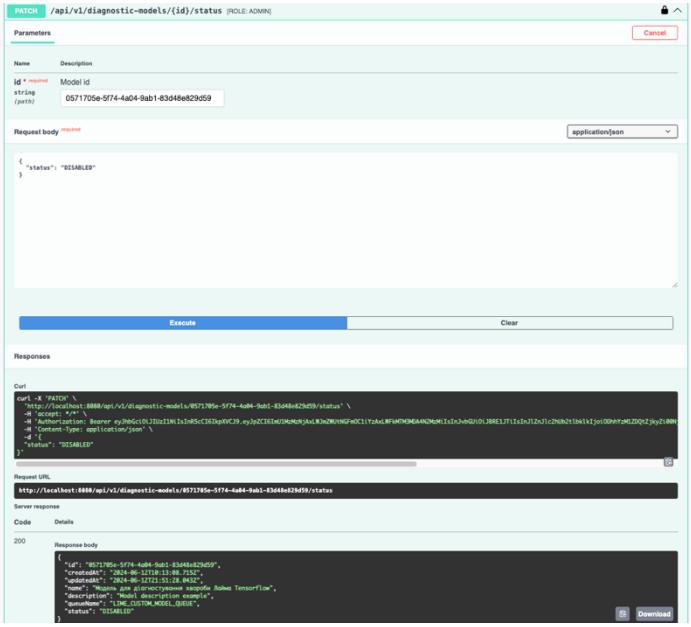


Рисунок 4.19 – Скриншот запита до ендпоінту PATCH /diagnostic-models/{id}/status

- Вимкнення/увімкнення доступу до певної версії моделі ШІ – ендпоінт PATCH /diagnostic-models/versions/{id}/status (рис. 4.20). Версії моделей в статусі DISABLED є вимкненими та недоступні для діагностики, а версії моделей в статусі ENABLED є увімкненими та доступні для виконання діагностики.

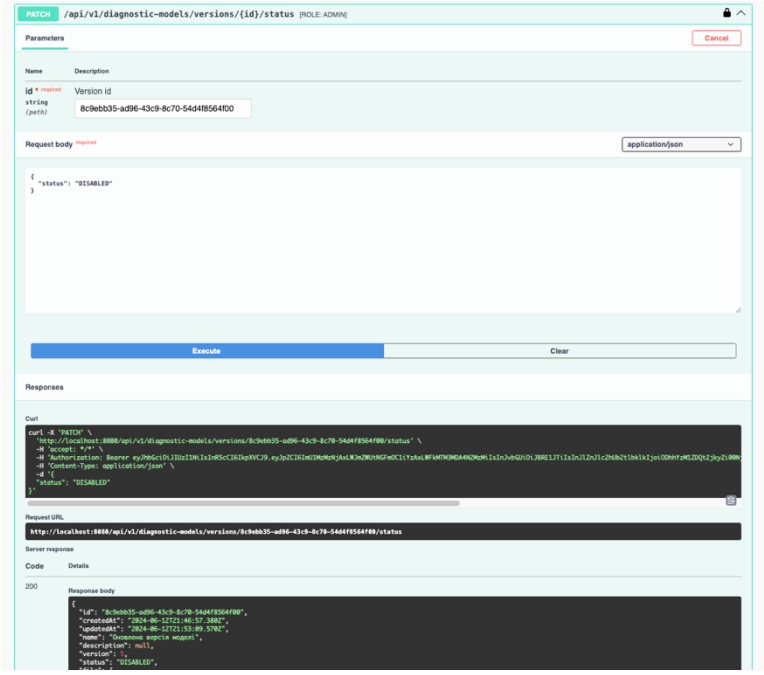


Рисунок 4.20 – Скриншот запита до ендпоінту PATCH /diagnostic-models/versions/{id}/status

- Створення запитів на діагностику захворювань за зображенням – ендпоінт POST /diagnostics (рис. 4.21). В даному запиті передається зображення для діагностики та масив моделей, з якими потрібно виконати дослідження. У коді системи обираються останні доступні версії для переданих моделей та надсилається запит для виконання діагностики в чергу для обробки відповідним сервісом. У результаті створюється об’єкт, що являє собою вкладену структуру всіх типів досліджень, які містять всі версії моделей, які містять всі версії моделей, якими виконується діагностика захворювань (рис 4.22).

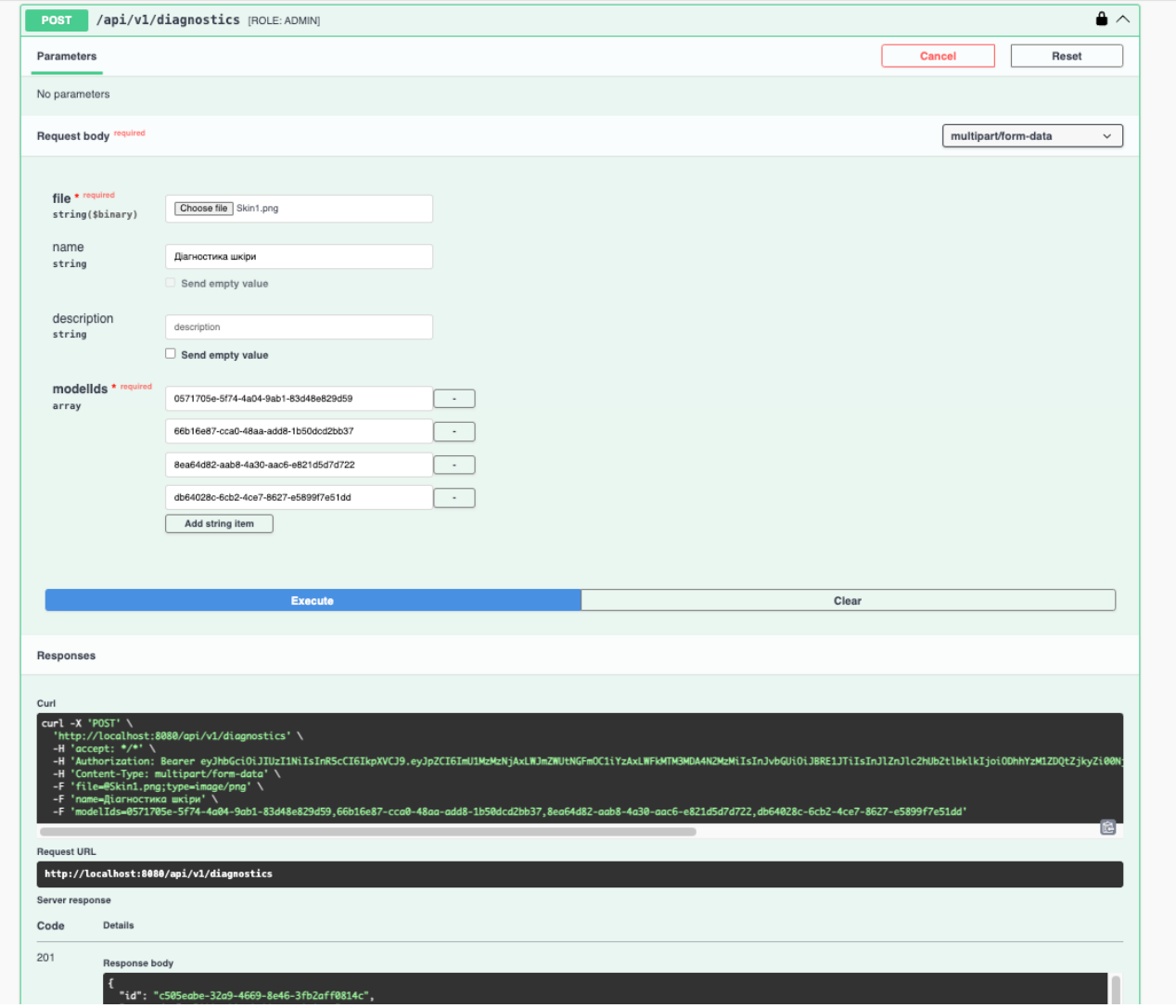


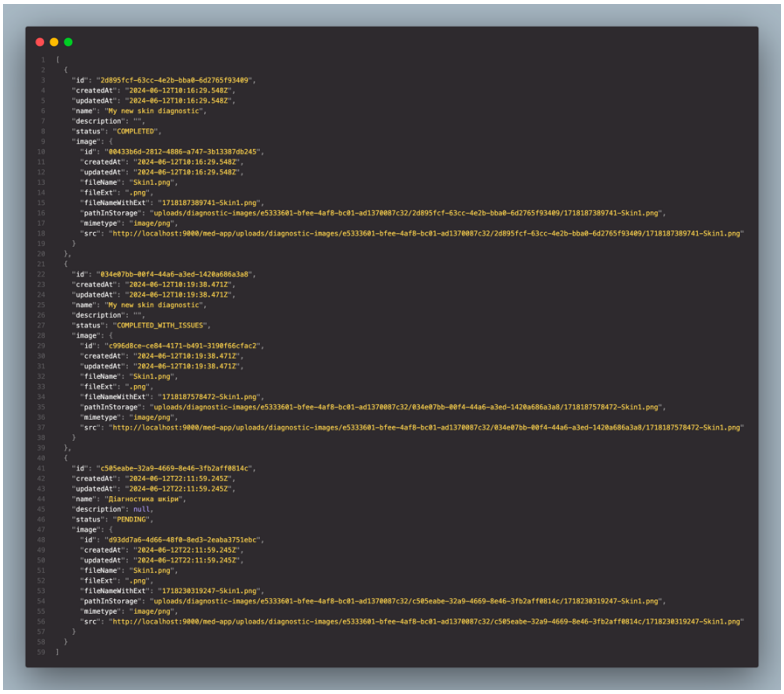
Рисунок 4.21 – Скриншот запита до ендпоінту POST /diagnostics

```
{
  "id": "3d82c13d-0492-4eed-a6c4-b95f60134653",
  "createdAt": "2024-06-13T23:13:27.017Z",
  "updatedAt": "2024-06-13T23:13:27.017Z",
  "name": "Діагностика шкіри",
  "description": null,
  "status": "PENDING",
  "image": {
    "id": "d40accc9-eec9-4d95-ba1e-64c0a7aed09e",
    "createdAt": "2024-06-13T23:13:27.017Z",
    "updatedAt": "2024-06-13T23:13:27.017Z",
    "fileName": "Skin1.png",
    "fileExt": ".png",
    "fileNameWithExt": "1718320407024-Skin1.png",
    "pathInStorage": "uploads/diagnostic-images/e5333601-bfee-4af8-bc01-ad1370087c32/3d82c13d-0492-4eed-a6c4-b95f60134653/1718320407024-Skin1.png",
    "mimetype": "image/png",
    "src": "http://localhost:9000/med-app/uploads/diagnostic-images/e5333601-bfee-4af8-bc01-ad1370087c32/3d82c13d-0492-4eed-a6c4-b95f60134653/1718320407024-Skin1.png"
  },
  "types": {
    "72fd8099-f654-4698-b40c-9b9b08978d55": {
      "id": "72fd8099-f654-4698-b40c-9b9b08978d55",
      "createdAt": "2024-06-11T16:28:34.241Z",
      "updatedAt": "2024-06-11T16:28:34.241Z",
      "name": "Діагностування хвороби Лайма",
      "description": null,
      "models": {
        "0571705e-5f74-4a04-9ab1-83d48e829d59": {
          "id": "0571705e-5f74-4a04-9ab1-83d48e829d59",
          "createdAt": "2024-06-12T10:13:08.715Z",
          "updatedAt": "2024-06-12T21:52:56.043Z",
          "name": "Модель для діагностування хвороби Лайма Tensorflow",
          "description": "Model description example",
          "queueName": "LIME_CUSTOM_MODEL_QUEUE",
          "status": "ENABLED",
          "versions": {
            "f90420bb-ec92-485f-92f7-666061d5d6cd": {
              "id": "f90420bb-ec92-485f-92f7-666061d5d6cd",
              "createdAt": "2024-06-12T10:14:00.404Z",
              "updatedAt": "2024-06-12T10:14:00.404Z",
              "name": "Pytorch Custom model for Furnuculosius",
              "description": "Model description example",
              "version": 3,
              "status": "ENABLED",
              "results": {
                "693bd269-36c9-4b5a-8c2d-1585365c2e26": {
                  "id": "693bd269-36c9-4b5a-8c2d-1585365c2e26",
                  "createdAt": "2024-06-13T23:13:27.017Z",
                  "updatedAt": "2024-06-13T23:13:27.017Z",
                  "status": "PENDING",
                  "diseaseProbability": null,
                  "resultDescription": null
                }
              }
            }
          }
        }
      }
    },
    "1ae1b71d-e508-459d-a147-03a9a6ba5071": {
      "id": "1ae1b71d-e508-459d-a147-03a9a6ba5071",
      "createdAt": "2024-06-11T16:28:34.241Z",
      "updatedAt": "2024-06-11T16:28:34.241Z",
      "name": "Діагностування фурункульозу",
      "description": null,
      "models": {
        "db64028c-6cb2-4ce7-8627-e5899f7e51dd": {
          "id": "db64028c-6cb2-4ce7-8627-e5899f7e51dd",
          "createdAt": "2024-06-12T10:11:55.440Z",
          "updatedAt": "2024-06-12T10:11:55.440Z",
          "name": "Модель Tensorflow для діагностування фурункульозу",
          "description": "Model description example",
          "queueName": "FURUNKOLOSIUS_CUSTOM_MODEL_QUEUE",
          "status": "ENABLED",
          "versions": {
            "9b285fe3-65ff-4ff0-b54e-9a98e6607068": {
              "id": "9b285fe3-65ff-4ff0-b54e-9a98e6607068",
              "createdAt": "2024-06-12T10:14:38.257Z",
              "updatedAt": "2024-06-12T10:14:38.257Z",
              "name": "Furnuculosius",
              "description": "Model description example",
              "version": 3,
              "status": "ENABLED",
              "results": {
                "05613c54-cbea-4ef4-8953-dafe4244ee2d": {
                  "id": "05613c54-cbea-4ef4-8953-dafe4244ee2d",
                  "createdAt": "2024-06-13T23:13:27.017Z",
                  "updatedAt": "2024-06-13T23:13:27.017Z",
                  "status": "PENDING",
                  "diseaseProbability": null,
                  "resultDescription": null
                }
              }
            }
          }
        }
      }
    }
  }
}
```


Рисунок 4.22 – Скриншот сруктури створеної діагностики

- Перегляд результату діагностики – ендпоінт GET /diagnostics/{id}. У результаті повертається об’єкт такої ж структури, як і для ендпоінту на створення діагностики.

- Перегляд історії діагностик – ендпоінт GET /diagnostics (рис 4.23).



```
[
  {
    "id": "2d895fcf-63cc-4e2b-bba0-6d2765f93409",
    "createdAt": "2024-06-12T10:16:29.548Z",
    "updatedAt": "2024-06-12T10:16:29.548Z",
    "name": "My new skin diagnostic",
    "description": "",
    "status": "COMPLETED",
    "image": {
      "id": "00433b6d-2812-4886-a747-3b13387db245",
      "createdAt": "2024-06-12T10:16:29.548Z",
      "updatedAt": "2024-06-12T10:16:29.548Z",
      "fileName": "Skin1.png",
      "fileExt": ".png",
      "fileNameWithExt": "1718187389741-Skin1.png",
      "pathInStorage": "uploads/diagnostic-images/e5333601-bfee-4af8-bc01-ad1370087c32/2d895fcf-63cc-4e2b-bba0-6d2765f93409/1718187389741-Skin1.png",
      "mimetype": "image/png",
      "src": "http://localhost:9000/med-app/uploads/diagnostic-images/e5333601-bfee-4af8-bc01-ad1370087c32/2d895fcf-63cc-4e2b-bba0-6d2765f93409/1718187389741-Skin1.png"
    }
  },
  {
    "id": "034e07bb-00f4-44a6-a3ed-1420a686a3a8",
    "createdAt": "2024-06-12T10:19:38.471Z",
    "updatedAt": "2024-06-12T10:19:38.471Z",
    "name": "My new skin diagnostic",
    "description": "",
    "status": "COMPLETED_WITH_ISSUES",
    "image": {
      "id": "c996d8ce-ce84-4171-b491-3190f66cfac2",
      "createdAt": "2024-06-12T10:19:38.471Z",
      "updatedAt": "2024-06-12T10:19:38.471Z",
      "fileName": "Skin1.png",
      "fileExt": ".png",
      "fileNameWithExt": "1718187578472-Skin1.png",
      "pathInStorage": "uploads/diagnostic-images/e5333601-bfee-4af8-bc01-ad1370087c32/034e07bb-00f4-44a6-a3ed-1420a686a3a8/1718187578472-Skin1.png",
      "mimetype": "image/png",
      "src": "http://localhost:9000/med-app/uploads/diagnostic-images/e5333601-bfee-4af8-bc01-ad1370087c32/034e07bb-00f4-44a6-a3ed-1420a686a3a8/1718187578472-Skin1.png"
    }
  },
  {
    "id": "c505eabe-32a9-4669-8e46-3fb2aff0814c",
    "createdAt": "2024-06-12T22:11:59.245Z",
    "updatedAt": "2024-06-12T22:11:59.245Z",
    "name": "Діагностика акіри",
    "description": null,
    "status": "PENDING",
    "image": {
      "id": "d93dd7a6-4d66-48f0-8ed3-2eaba3751ebc",
      "createdAt": "2024-06-12T22:11:59.245Z",
      "updatedAt": "2024-06-12T22:11:59.245Z",
      "fileName": "Skin1.png",
      "fileExt": ".png",
      "fileNameWithExt": "1718230319247-Skin1.png",
      "pathInStorage": "uploads/diagnostic-images/e5333601-bfee-4af8-bc01-ad1370087c32/c505eabe-32a9-4669-8e46-3fb2aff0814c/1718230319247-Skin1.png",
      "mimetype": "image/png",
      "src": "http://localhost:9000/med-app/uploads/diagnostic-images/e5333601-bfee-4af8-bc01-ad1370087c32/c505eabe-32a9-4669-8e46-3fb2aff0814c/1718230319247-Skin1.png"
    }
  }
]
```


Рисунок 4.23 – Скриншот тіла відповіді на запит до ендпоінту GET /diagnostics

У результаті розроблені ендпоінти та логіка додатка реалізовують всі визначені функціональні вимоги до системи з високою та середньою пріоритетністю.

4.4 Аналіз розробленої системи

У результаті виконання дипломного проєкту була реалізована система, що надає можливість до зручного адміністрування доступних типів дослідження, моделей ШІ та їхніх версій, а також надає пацієнтам доступ до вибору та виконання декількох досліджень одночасно, а також вибору моделей ШІ, якими буде виконано дослідження. При цьому система забезпечує спільний доступ багатьох користувачів до одночасного виконання запитів на діагностики зображення за допомогою моделей ШІ. Відповідно до розглянутих у першому розділі аналогів, система забезпечує виконання функцій, які відсутні в кожній з розглянутих систем. Також система забезпечує всі функціональні вимоги до

системи, пріоритетність яких була визначена як висока та середня, а також забезпечує виконання нефункціональних вимог, а саме впровадження механізму автентифікації та авторизації користувачів, надання можливості відправляти запити на діагностики та продовжувати користуватись системою, можливість до масштабування через можливість додавати контейнери сервісів для аналізу зображень, обмежує доступ до персональних даних користувачів та надає документацію з інтеграції клієнтів з серверним API.

4.5 Способи покращення системи

Підтримка, покращення та розвиток проєкту є важливим фактором у сучасних інформаційних системах. Розробка системи виконувалась таким чином, щоб забезпечити можливість для гнучкого розширення та модифікації системи для її розвитку. Під час аналізу розробки та результатів виконання проєкту було визначено способи покращення системи:

- Реалізація реєстрації та верифікації пацієнтів за допомогою сторонніх сервісів. На цей час у системі доступно лише можливість створення пацієнтів користувачем з роллю адміністратора. Завдання верифікації користувача та надання йому акаунта для користування системою може бути автоматизовано за допомогою сторонніх сервісів (наприклад, з використанням сервісів Дія.Підпис або Bank ID).
- Вирішення завдання автоматичного масштабування ресурсів. Наразі система підтримує можливість одночасного виконання досліджень багатьма контейнерами для різних моделей, а також наявна можливість ручного додавання контейнерів моделей для контейнерів моделей, які вже використовуються, через взаємодію з Docker CLI або Docker API. Це забезпечує необхідний рівень

продуктивності та гнучкості системи, проте потребує залучення сторонньої людини для керування ресурсами. Автоматизація процесу масштабування ресурсів залежно від навантаження на контейнери з різними моделями дозволить автоматично зменшувати або збільшувати кількість запущених реплік контейнерів залежно від зміни навантаження на систему. Для вирішення цієї задачі можна застосувати відповідні інструменти оркестрації та масштабування системи (наприклад, Kubernetes)

- Додавання користувачів з роллю лікарів для взаємодії з пацієнтами. Додавання користувачів з роллю лікарів дозволить виконувати не лише попередню діагностику за допомогою моделей ШІ, а й аналіз результатів діагностики захворювань за зображеннями моделями ШІ кваліфікованими фахівцями, комунікувати з пацієнтами для уточнення симптомів та деталей захворювання, встановлення діагнозів та вибору способів лікування.
- Забезпечення консиліуму моделей. Консиліум моделей забезпечить можливість виконання аналізу зображення для одного типу діагностики, використовуючи результати декількох різних моделей, які виконують цей тип діагностики. Цей процес забезпечується через визначення результату шляхом аналізу та об’єднання результатів декількох подібних моделей ШІ, що підвищить точність та надійність результатів діагностики.

ВИСНОВОК ДО РОЗДІЛУ 4

У четвертому розділі було розглянуто розгортання системи, виконано огляд документації з інтеграції клієнтів з API системи, представлено основні сценарії системи, виконано аналіз розробленої системи та визначено способи покращення системи.

Для виконання розгортання компонентів системи було обрано розгортання кожного компоненту у Docker-контейнері та конфігурація розгортання контейнерів за допомогою docker-compose. Для цього було використано офіційні образи для створення контейнерів бази даних PostgreSQL, файлового сховища MinIO та черги повідомлень RabbitMQ. Також було описано образи для створення контейнерів для сервісу обробки клієнтських API запитів та сервісу для діагностики захворювань за зображеннями за допомогою моделей ШІ. У результаті всі компоненти були описані в docker-compose як окремі сервіси, було призначено використання спільної мережі, та створено конфігурації volumes для контейнерів, що мають зберігати дані, які не будуть втрачатись при перезавантаженні контейнера.

Оглянуто створену за допомогою Swagger документації та виконано запити, які відповідали сценаріям використання системи.

Також було проаналізовано результати розробки системи, виконано порівняння з системами-аналогами та визначено дотримання функціональних та нефункціональних вимог визначених у першому розділі. Наступним кроком було описано способи для покращення системи, а саме реалізацію реєстрації та верифікації пацієнтів, вирішення задач автоматичного масштабування ресурсів, додавання користувачів з роллю лікарів та забезпечення консиліуму моделей.

ВИСНОВКИ

Під час виконання дипломної роботи було реалізовано серверну частину системи для спільного використання моделей ШІ для діагностування захворювань, що надає можливість адміністрування доступних типів діагностик, моделей ШІ та їхніх версій, а також надає можливість кирстувачам-пацієнтам доступ до одночасного виконання діагностики захворювань за зображеннями з використанням багатьох типів досліджень та моделей ШІ.

У першому розділі було проаналізовано предметну область, розглянуто існуючі системи аналоги (сервіси med.comsys.kpi.ua, Qure.ai та Triage), виконано порівняння загальних властивостей проаналізованих систем, розглянуто переваги та недоліки кожної системи та визначено способи покращення існуючих рішень. Наступним кроком було визначено основні функціональні та нефункціональні вимоги до розроблюваної системи.

У другому розділі було проаналізовано та обрано архітектуру системи. Відповідно до завдань системи, а саме забезпечення одночасного доступу роботи користувачів до виконання багатьох діагностик та специфіки процесу діагностики зображення за допомогою моделей ШІ, було обрано розробку двох сервісів: сервісу для обслуговування API запитів від клієнтів та сервісу для виконання діагностики захворювань за зображеннями за допомогою моделей ШІ. Було обрано спосіб комунікації між сервісами системи, визначено компоненти системи та способи взаємодії між ними, обрано технології для реалізації компонентів системи.

У третьому розділі роботи було описано та виконано розробку кожного з сервісів системи. Для сервісу для обслуговування API запитів було визначено структуру сервісу, виконано конфігурування сервісу, налаштовано з’єднання з базою даних PostgreSQL та створено моделі, що відповідають сутностями бази даних. Після цього було реалізовано початкову міграцію для створення

структури бази даних та зв’язків між таблицями, виконано інтеграцію з файловим сховищем MinIO та налагоджено взаємодію з чергою повідомлень RabbitMQ. Наступним кроком було реалізовано модулі та сервіси для роботи з бізнес-логікою серверного додатка та реалізовано модуль автентифікації та авторизації для забезпечення контролю доступу користувачів до ресурсів сервера. Також було розроблено контролери для обробки запитів на ендпоінти сервісу, визначено структуру маршрутів сервера, вхідних параметрів та результатів. Для інтеграції клієнтів з серверним API було виконано розробку документації за допомогою інструментів Swagger.

Для сервісу діагностики захворювань за зображеннями за допомогою моделей ШІ було визначено структуру сервісу, виконано інтеграцію з файловим сховищем MinIO, налаштовано взаємодію з чергою повідомлень RabbitMQ та розроблено сервіс для завантаження моделей ШІ в пам’ять та виконання аналізу за допомогою відповідних моделей.

У четвертому розділі роботи було розглянуто розгортання системи, виконано огляд розробленої документації для інтеграції клієнтів з API системи, продемонстровано основні сценарії системи, виконано аналіз розробленої системи, порівняння з системами-аналогами та порівняння на дотримання функціональних та нефункціональних вимог до системи, визначених в першому розділі. Також було визначено способи до розширення та розвитку системи.

Результатом виконання бакалаврського дипломного проєкту є розроблена серверна частина системи для спільного використання моделей штучного інтелекту для діагностування захворювань, що відповідає визначеним функціональним та нефункціональним вимогам та повною мірою реалізовує функціональність для вирішення визначених задач дипломного проєкту.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Principles and Perspectives in Medical Diagnostic Systems Employing Artificial Intelligence (AI) Algorithms. *IRJEMS | International Research Journal of Economics and Management Studies*. URL: https://irjems.org/irjems-v3i1p144.html (дата звернення: 21.05.2024).
2. Medical Diagnostic Systems Using Artificial Intelligence (AI) Algorithms: Principles and Perspectives. *IEEE Xplore*. URL: https://ieeexplore.ieee.org/abstract/document/9279211 (дата звернення: 21.05.2024).
3. Contributors to Wikimedia projects. Computer-aided diagnosis - Wikipedia. *Wikipedia, the free encyclopedia*. URL: https://en.wikipedia.org/wiki/Computer-aided_diagnosis (дата звернення: 21.05.2024).
4. Платформа штучного інтелекту для дистанційного автоматизованого виявлення і дистанційної автоматизованої діагностики хвороб людини. *Платформа штучного інтелекту для дистанційного автоматизованого виявлення і дистанційної автоматизованої діагностики хвороб людини*. URL: https://med.comsys.kpi.ua/ (дата звернення: 21.05.2024).
5. Qure.ai. *Qure.ai*. URL: https://app.qure.ai/ (дата звернення: 21.05.2024).
6. Qure.ai App User's Manual | Qure.ai Documentation. *Qure.ai App User's Manual | Qure.ai Documentation*. URL: https://documentation.qure.ai/#what-is-qure.ai-app (дата звернення: 21.05.2024).
7. triage. Triage - Detect Disease In A Snap. *Triage - Detect Disease In A Snap*. URL: https://www.triage.com/ (дата звернення: 21.05.2024).

8. *Triage - Detect Disease In A Snap Documentation*. URL: https://www.triage.com/pdf/en/Instructions%20for%20Use.pdf (дата звернення: 21.05.2024).
9. Microservices Pattern: Pattern: Monolithic Architecture. *microservices.io*. URL: https://microservices.io/patterns/monolithic.html (дата звернення: 21.05.2024).
10. Microservices Pattern: Microservice Architecture pattern. *microservices.io*. URL: https://microservices.io/patterns/microservices.html (дата звернення: 21.05.2024).
11. Ozkaya M. Microservices Communications. *Medium*. URL: https://medium.com/design-microservices-architecture-with-patterns/microservices-communications-f319f8d76b71 (дата звернення: 21.05.2024).
12. Ozkaya M. Microservices Asynchronous Message-Based Communication. *Medium*. URL: https://medium.com/design-microservices-architecture-with-patterns/microservices-asynchronous-message-based-communication-6643bee06123 (дата звернення: 21.05.2024).
13. What is a relational database?. *Oracle | Cloud Applications and Cloud Platform*. URL: https://www.oracle.com/database/what-is-a-relational-database/ (дата звернення: 21.05.2024).
14. Учасники проектів Вікімедіа. ACID – Вікіпедія. *Вікіпедія*. URL: https://uk.wikipedia.org/wiki/ACID (дата звернення: 21.05.2024).
15. What Is A Non-Relational Database?. *MongoDB*. URL: https://www.mongodb.com/resources/basics/databases/non-relational (дата звернення: 21.05.2024).
16. PostgreSQL: About. *PostgreSQL: The world's most advanced open source database*. URL: https://www.postgresql.org/about/ (дата звернення: 21.05.2024).

17. AI-Enhanced Data Solutions with Database 23ai. *Oracle | Cloud Applications and Cloud Platform*. URL: https://www.oracle.com/database/ (дата звернення: 21.05.2024).
18. RabbitMQ Essentials. *Google Books*. URL: https://books.google.com.ua/books?hl=en&lr=&id=FoBvAwAAQBAJ&oi=fnd&pg=PT8&dq=RabbitMQ+&ots=87MaGDSUj2&sig=gAUOeIzANOsF13Ufa8gVbXJ%20fItU&redir_esc=y#v=onepage&q=RabbitMQ&f=false (дата звернення: 21.05.2024).
19. Advanced Message Queuing Protocol (AMQP). *ACM Digital Library*. URL: https://dl.acm.org/doi/fullHtml/10.5555/1653247.1653250 (дата звернення: 21.05.2024).
20. Message Queuing Service - Amazon Simple Queue Service - AWS. *Amazon Web Services, Inc.* URL: https://aws.amazon.com/sqs/ (дата звернення: 21.05.2024).
21. Introduction to Azure Service Bus, an enterprise message broker - Azure Service Bus. *Microsoft Learn: Build skills that open doors in your career*. URL: https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-messaging-overview (дата звернення: 21.05.2024).
22. Apache Kafka. *Apache Kafka*. URL: https://kafka.apache.org/intro (дата звернення: 21.05.2024).
23. MinIO | S3 & Kubernetes Native Object Storage for AI. *MinIO*. URL: https://min.io/ (дата звернення: 21.05.2024).
24. Amazon S3 - Cloud Object Storage - AWS. *Amazon Web Services, Inc.* URL: https://aws.amazon.com/s3/ (дата звернення: 21.05.2024).
25. Database Migrations: What are the Types of DB Migrations?. *Prisma's Data Guide*. URL: https://www.prisma.io/dataguide/types/relational/what-are-database-migrations#:~:text=Migrations%20help%20transition%20database%20sche

mas,stru%20ctures%20in%20a%20programmatic%20way (дата звернення: 21.05.2024).

26. amqplib. *npm*. URL: https://www.npmjs.com/package/amqplib (дата звернення: 21.05.2024).
27. Documentation | NestJS - A progressive Node.js framework. *Documentation | NestJS - A progressive Node.js framework*. URL: https://docs.nestjs.com/fundamentals/lifecycle-events#asynchronous-initialization (дата звернення: 21.05.2024).
28. Musser D. R., Stepanov A. A. Generic programming. *SpringerLink*. URL: https://link.springer.com/chapter/10.1007/3-540-51084-2_2 (дата звернення: 21.05.2024).
29. Documentation | NestJS - A progressive Node.js framework. *Documentation | NestJS - A progressive Node.js framework*. URL: https://docs.nestjs.com/security/authentication (дата звернення: 21.05.2024).
30. bcrypt. *npm*. URL: https://www.npmjs.com/package/bcrypt (дата звернення: 21.05.2024).
31. class-validator. *npm*. URL: https://www.npmjs.com/package/class-validator (дата звернення: 21.05.2024).
32. What is OpenAPI? - OpenAPI Initiative. *OpenAPI Initiative*. URL: https://www.openapis.org/what-is-openapi (дата звернення: 21.05.2024).
33. @nestjs/swagger. *npm*. URL: https://www.npmjs.com/package/@nestjs/swagger (дата звернення: 21.05.2024).
34. Introduction to Pika – pika 1.3.2 documentation. *Introduction to Pika – pika 1.3.2 documentation*. URL: https://pika.readthedocs.io/en/stable/ (дата звернення: 21.05.2024).

35. Docker Compose overview. *Docker Documentation*. URL: https://docs.docker.com/compose/ (дата звернення: 21.05.2024).
36. The World's Largest Container Registry | Docker. *Docker*. URL: https://www.docker.com/products/docker-hub/#:~:text=Docker%20Hub%20is%20a%20container,repos%20for%20teams%20%20and%20enterprises (дата звернення: 21.05.2024).

ДОДАТОК 1

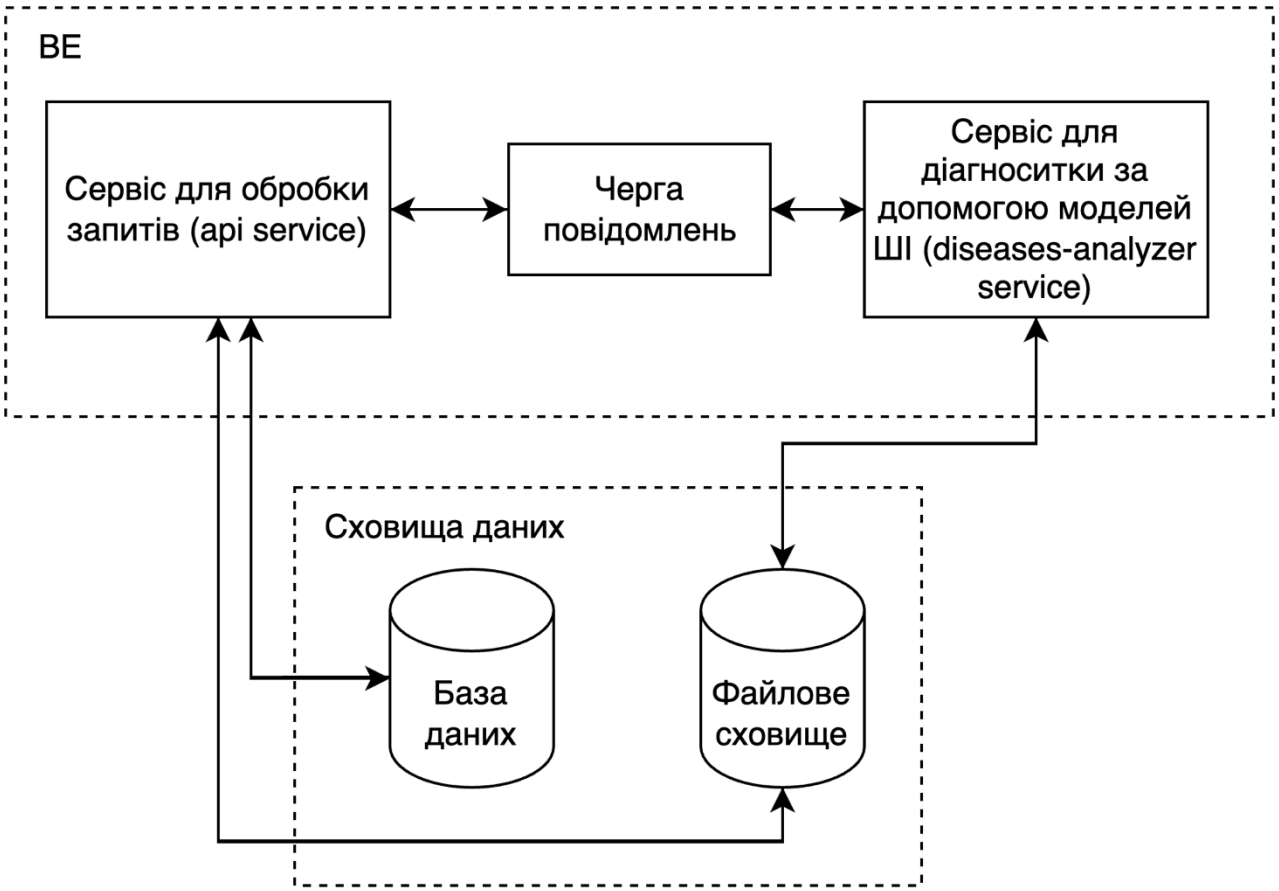
Система для спільного використання моделей штучного інтелекту для діагностування захворювань (серверна частина)

Схема компонентів системи (структурна схема)

ІАЛЦ.467200.004 Д1

Аркушів 1

Київ 2024 р



					ІАЛЦ.467200.004 Д1		
		№ докум.	Підпис	Дата			
Розробив		Храпко В. А.			**Система для спільного використання моделей штучного інтелекту для діагностування захворювань (серверна частина)** Схема компонентів системи (структурна схема)	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Ковальчук О. М.					
						КПІ ім. Ігоря Сікорського, ФІОТ, ІП-04	
Н. Контр.		Волокита А. М.					
Затвердив		Стіренко С. Г.					

ДОДАТОК 2

Система для спільного використання моделей штучного інтелекту для діагностування захворювань (серверна частина)

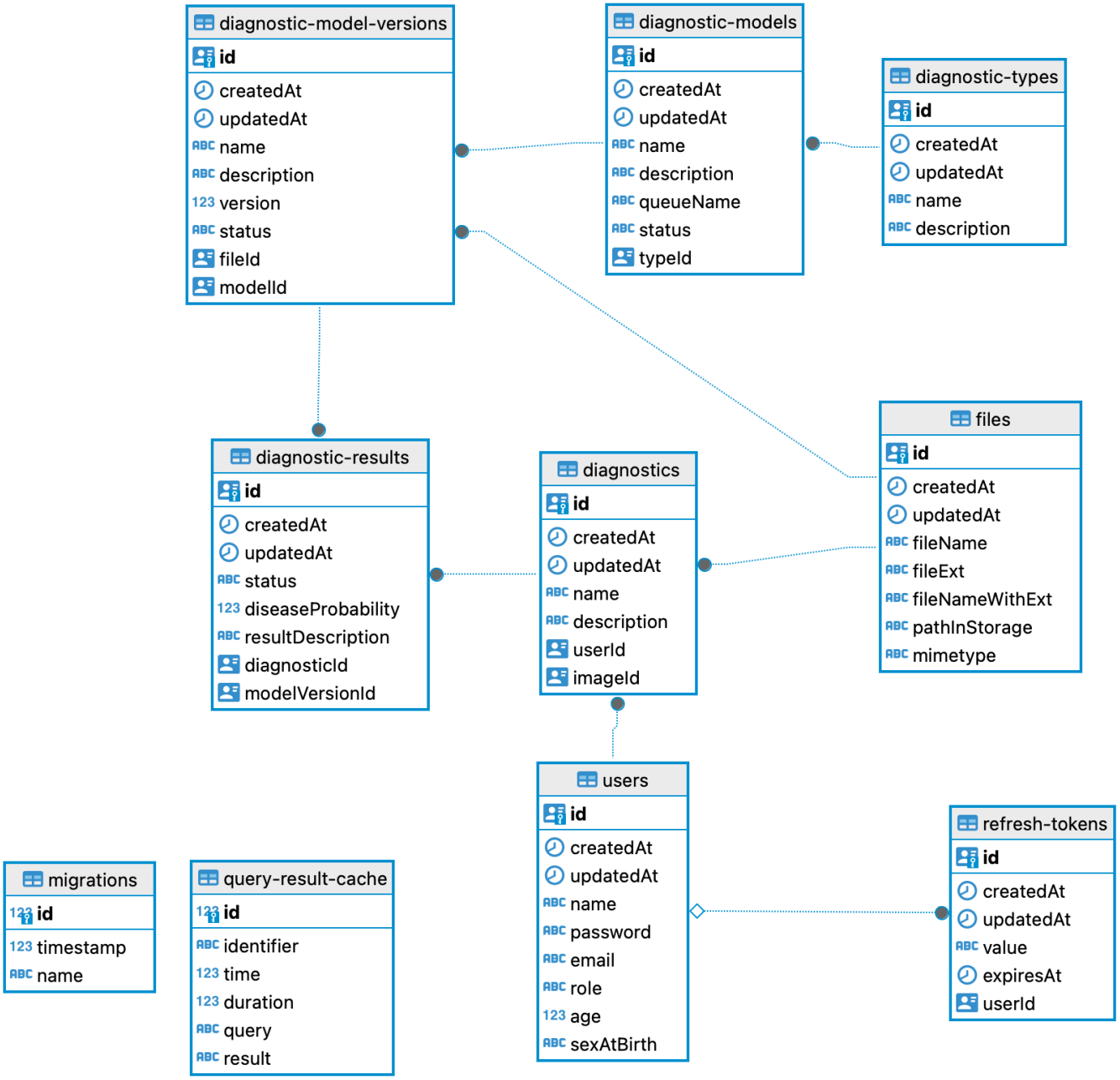
Діаграма бази даних (функціональна схема)

ІАЛЦ.467200.005 Д2

Аркушів 1

Київ 2024 р

Київ 2024 р



					ІАЛЦ.467200.005 Д2	
		№ докум.	Підпис	Дата		
Розробив		Храпко В. А.			Система для спільного використання моделей штучного інтелекту для діагностування захворювань (серверна частина)	Літ. / Аркуш / Аркушів: 1 / 1
Перевірив		Ковальчук О. М.				
Н. Контр.		Волокита А. М.			Діаграма бази даних (функціональна схема)	КПІ ім. Ігоря Сікорського, ФІОТ, ІП-04
Затвердив		Стіренко С. Г.				

ДОДАТОК 3

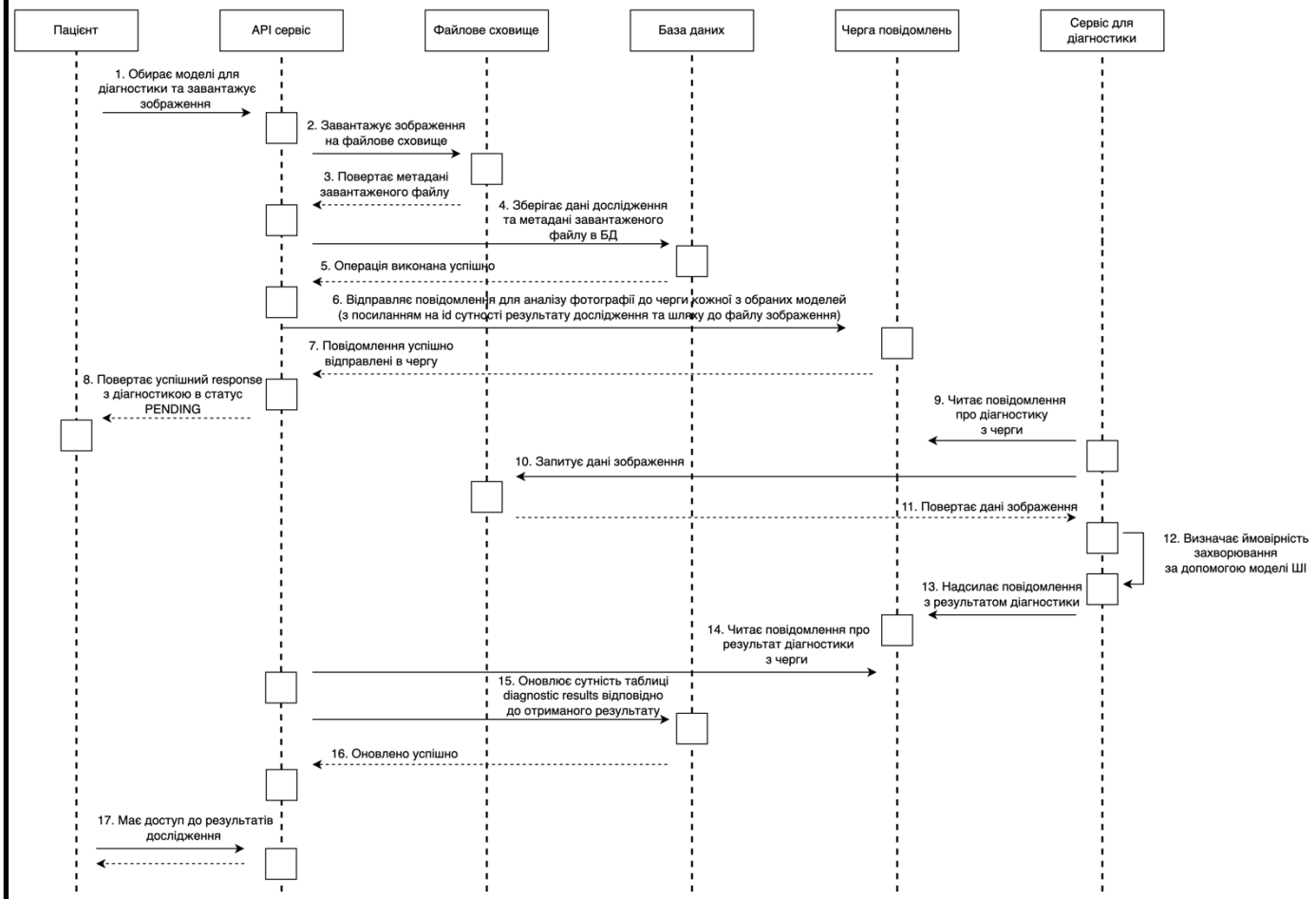
Система для спільного використання моделей штучного інтелекту для діагностування захворювань (серверна частина)

Алгоритм дій програмного забезпечення (принципова схема)

ІАЛЦ.467200.006 Д3

Аркушів 1

Київ 2024 р



					ІАЛЦ.467200.006 Д3		
		№ докум.	Підпис	Дата			
Розробив		Храпко В. А.			**Система для спільного використання моделей штучного інтелекту для діагностування захворювань (серверна частина)** Алгоритм дій програмного забезпечення (принципова схема)	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Ковальчук О. М.					
Н. Контр.		Волокита А. М.				КПІ ім. Ігоря Сікорського, ФІОТ, ІП-04	
Затвердив		Стіренко С. Г.					

ДОДАТОК 4

Система для спільного використання моделей штучного інтелекту для діагностування захворювань (серверна частина)

Текст програмного коду

ІАЛЦ.467200.007 Д4

Аркушів 43

Київ 2024 р

```
// main.ts

import { NestFactory } from '@nestjs/core';
import {
  FastifyAdapter,
  NestFastifyApplication,
} from '@nestjs/platform-fastify';
import { AppModule } from './app.module';
import { DocumentBuilder, SwaggerModule } from '@nestjs/swagger';
import multipart from '@fastify/multipart';
import { validationPipe } from './common/pipes';
import { AppConfigService } from './config/app-config.service';

async function bootstrap() {
  const app = await NestFactory.create<NestFastifyApplication>(
    AppModule,
    new FastifyAdapter(),
  );
  const configService = app.get<AppConfigService>(AppConfigService);

  await app.register(multipart, {
    addToBody: true,
    limits: {
      fileSize: configService.get<number>('FILE_SIZE_LIMIT'),
    },
  });

  const HOST = configService.get<string>('NEST_HOST');
  const PORT = configService.get<number | string>('NEST_PORT');
  const GLOBAL_PREFIX = configService.get<string>('GLOBAL_PREFIX');

  app.setGlobalPrefix(GLOBAL_PREFIX);

  app.enableCors();

  const ENABLE_SWAGGER = configService.get<boolean>('ENABLE_SWAGGER');
  if (ENABLE_SWAGGER) {
    const swaggerConfig = new DocumentBuilder()
      .setTitle(configService.get('npm_package_name'))
      .setVersion(configService.get('npm_package_version'))
      .addBearerAuth()
      .build();

    const document = SwaggerModule.createDocument(app, swaggerConfig);
    const SWAGGER_DOCS_PATH = configService.get<string>('SWAGGER_DOCS_PATH');
    SwaggerModule.setup(SWAGGER_DOCS_PATH, app, document);
  }
  app.useGlobalPipes(validationPipe);
  await app.listen(PORT, HOST, () => {
    console.log(`Server listens on http://${HOST}:${PORT}`);
  });
}
bootstrap();


// app.module.ts

import { Module } from '@nestjs/common';
```

```
import { AppConfigModule } from './config';
import { DatabaseModule } from './systems/database';
import { FilesModule } from './modules/files';
import { UsersModule } from './modules/users';
import { AuthModule } from './modules/auth/auth.module';
import { DiagnosticTypesModule } from './modules/diagnostic-types';
import { DiagnosticModelsModule } from './modules/diagnostic-models';
import { DiagnosticModelVersionsModule } from './modules/diagnostic-model-versions';
import { DiagnosticsModule } from './modules/diagnostics';
import { DiagnosticResultsModule } from './modules/diagnostic-results';
import { MessageQueueModule } from './systems/message-queue';

@Module({
  imports: [
    AppConfigModule,
    DatabaseModule,
    AuthModule,
    UsersModule,
    DiagnosticTypesModule,
    DiagnosticModelsModule,
    DiagnosticModelVersionsModule,
    DiagnosticsModule,
    DiagnosticResultsModule,
    FilesModule,
    MessageQueueModule,
  ],
})
export class AppModule {}
```

// app-config.module.ts

```
import { Global, Module } from '@nestjs/common';
import { ConfigModule } from '@nestjs/config';
import { AppConfigService } from './app-config.service';

@Global()
@Module({
  imports: [
    ConfigModule.forRoot({ envFilePath: `.env.${process.env.NODE_ENV}` }),
  ],
  providers: [AppConfigService],
  exports: [AppConfigService],
})
export class AppConfigModule {}
```

// app-config.service.ts

```
import { Injectable } from '@nestjs/common';
import { ConfigService } from '@nestjs/config';
import { TypeOrmModuleOptions } from '@nestjs/typeorm';
import { join } from 'node:path';
import { NodeEnvEnum } from 'src/common/enums';

@Injectable()
export class AppConfigService {
  constructor(private configService: ConfigService) {}

  get<T>(key: string): T {
    const value = this.configService.get<T>(key);
    if (!value) throw new Error(key + ' env variable not set');

    try {
```

```
      return JSON.parse(value as string);
    } catch {
      return value;
    }
  }

  public isNodeEnv(mode: NodeEnvEnum): boolean {
    return this.get('NODE_ENV') === mode;
  }

  getDbConfig(): TypeOrmModuleOptions {
    return {
      type: this.get<'postgres'>('TYPEORM_TYPE'),
      name: this.get('TYPEORM_CONNECTION_NAME'),
      host: this.get('TYPEORM_HOST'),
      port: this.get('TYPEORM_PORT'),
      cache: this.get('TYPEORM_CACHE'),
      logging: this.get('TYPEORM_LOGGING'),
      database: this.get<string>('TYPEORM_DATABASE'),
      username: this.get('TYPEORM_USERNAME'),
      password: this.get<string>('TYPEORM_PASSWORD'),
      extra: {
        ssl: this.get('TYPEORM_SSL'),
      },
      dropSchema: this.get('TYPEORM_DROP_SCHEMA'),
      synchronize: this.get('TYPEORM_SYNCHRONIZE'),
      migrationsRun: this.get('TYPEORM_MIGRATIONS_RUN'),
      entities: [join(__dirname, '../modules/**/*.entity.{ts,js}')],
      migrations: [join(__dirname, '../systems/database/migrations/*.{ts,js}')],
    };
  }
}
```

// common.entity.ts

```
import { ApiProperty } from '@nestjs/swagger';
import {
  BaseEntity,
  PrimaryGeneratedColumn,
  CreateDateColumn,
  UpdateDateColumn,
} from 'typeorm';

export abstract class CommonEntity extends BaseEntity {
  @ApiProperty()
  @PrimaryGeneratedColumn('uuid')
  id: string;

  @ApiProperty()
  @CreateDateColumn({ type: 'timestamptz' })
  createdAt: Date;

  @ApiProperty({ readOnly: true })
  @UpdateDateColumn({ type: 'timestamptz' })
  updatedAt: Date;
}
```

// base.service.ts

```
import { BadRequestException, NotFoundException } from '@nestjs/common';
import {
  FindManyOptions,
  FindOptionsWhere,
```

```
  FindOneOptions,
  Repository,
  EntityManager,
} from 'typeorm';
import { CommonEntity } from '../entities';
import { TServiceErrorMessages } from '../types';

export abstract class BaseService<T extends CommonEntity> {
  constructor(
    protected entityRepository: Repository<T>,
    protected serviceErrorMessages: TServiceErrorMessages,
  ) {}

  async findAll(
    options: FindManyOptions<T> = { loadEagerRelations: true },
    transactionManager?: EntityManager,
  ): Promise<T[]> {
    const repository = transactionManager
      ? transactionManager.getRepository<T>(this.entityRepository.target)
      : this.entityRepository;

    return repository.find(options).catch(() => {
      throw new NotFoundException(this.serviceErrorMessages.entitiesNotFound);
    });
  }

  async findOne(
    conditions: FindOptionsWhere<T>,
    options: FindOneOptions<T> = { loadEagerRelations: true },
    transactionManager?: EntityManager,
  ): Promise<T> {
    const repository = transactionManager
      ? transactionManager.getRepository<T>(this.entityRepository.target)
      : this.entityRepository;

    return repository
      .findOneOrFail({
        ...options,
        where: conditions,
      })
      .catch(() => {
        throw new NotFoundException(this.serviceErrorMessages.entityNotFound);
      });
  }

  async createOne(
    entity: Partial<T>,
    transactionManager?: EntityManager,
  ): Promise<T> {
    const repository = transactionManager
      ? transactionManager.getRepository<T>(this.entityRepository.target)
      : this.entityRepository;

    const entityToCreate = repository.create(entity as T);
    const { id } = await repository.save(entityToCreate).catch(() => {
      throw new BadRequestException(this.serviceErrorMessages.invalidData);
    });
    return this.findOne(
      { id } as FindOptionsWhere<T>,
      { loadEagerRelations: true },
      transactionManager,
    );
  }

  async updateOne(
```

```
    conditions: FindOptionsWhere<T>,
    entity: Partial<T>,
    transactionManager?: EntityManager,
  ): Promise<T> {
    const repository = transactionManager
      ? transactionManager.getRepository<T>(this.entityRepository.target)
      : this.entityRepository;

    const entityToUpdate = await this.findOne(
      conditions,
      { loadEagerRelations: false },
      transactionManager,
    );
    const updatedEntity = repository.merge(entityToUpdate, entity as T);
    const { id } = await repository.save(updatedEntity).catch(() => {
      throw new BadRequestException(this.serviceErrorMessages.invalidData);
    });

    return this.findOne(
      { id } as FindOptionsWhere<T>,
      { loadEagerRelations: true },
      transactionManager,
    );
  }

  async deleteOne(
    conditions: FindOptionsWhere<T>,
    transactionManager?: EntityManager,
  ): Promise<T> {
    const repository = transactionManager
      ? transactionManager.getRepository<T>(this.entityRepository.target)
      : this.entityRepository;

    const entity = await this.findOne(
      conditions,
      {
        loadEagerRelations: false,
      },
      transactionManager,
    );

    return repository.remove(entity).catch(() => {
      throw new NotFoundException(this.serviceErrorMessages.entityNotFound);
    });
  }
}
```

// database.module.ts

```
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { AppConfigModule } from '../../config';
import { AppConfigService } from '../../config/app-config.service';

@Module({
  imports: [
    TypeOrmModule.forRootAsync({
      imports: [AppConfigModule],
      inject: [AppConfigService],
      useFactory: (configService: AppConfigService) => ({
        ...configService.getDbConfig(),
      }),
    }),
  ],
  exports: [TypeOrmModule],
```

```
})
export class DatabaseModule {}

// message-queue.module.ts

import { Module, forwardRef } from '@nestjs/common';
import { RabbitMQPublisherService } from './rabbit-mq-publisher.service';
import { AppConfigModule } from 'src/config';
import { RabbitMQConsumerService } from './rabbit-mq-consumer.service';
import { RabbitMQConnectionService } from './rabbit-mq-connection.service';
import { DiagnosticsModule } from 'src/modules/diagnostics';

@Module({
  imports: [AppConfigModule, forwardRef(() => DiagnosticsModule)],
  providers: [
    RabbitMQConnectionService,
    RabbitMQPublisherService,
    RabbitMQConsumerService,
  ],
  exports: [RabbitMQPublisherService],
})
export class MessageQueueModule {}

// rabbit-mq-connection.service.ts

import { Inject, Injectable, Logger, forwardRef } from '@nestjs/common';
import * as amqp from 'amqplib';
import { AppConfigService } from 'src/config/app-config.service';
import { DiagnosticsService } from 'src/modules/diagnostics/diagnostics.service';

@Injectable()
export class RabbitMQConnectionService {
  private readonly logger = new Logger(RabbitMQConnectionService.name);

  private readonly rabbitMQProtocol =
    this.configService.get<string>('RABBIT_MQ_PROTOCOL');
  private readonly rabbitMQHost =
    this.configService.get<string>('RABBIT_MQ_HOST');
  private readonly rabbitMQUser = this.configService.get<string>(
    'RABBIT_MQ_DEFAULT_USER',
  );
  private readonly rabbitMQPassword = this.configService.get<string>(
    'RABBIT_MQ_DEFAULT_PASS',
  );
  private connection: amqp.Connection;
  private channel: amqp.Channel;

  constructor(
    private readonly configService: AppConfigService,
    @Inject(forwardRef(() => DiagnosticsService))
    private readonly diagnosticsService: DiagnosticsService,
  ) {}

  async setUpConnection() {
    this.connection = await amqp.connect(

`${this.rabbitMQProtocol}://${this.rabbitMQUser}:${this.rabbitMQPassword}@${this.rabbitMQHos
t}`,
    );
    this.channel = await this.connection.createChannel();
    await this.assertQueueForModels();
  }

  async assertQueue(queue: string) {
```

```
    this.logger.debug(`Asserting queue - ${queue}`);
    await this.channel.assertQueue(queue, { durable: true });
  }

  async assertQueueForModels() {
    const modelQueueNames =
      await this.diagnosticsService.selectModelQueueNames();

    for (const modelQueueName of modelQueueNames) {
      await this.assertQueue(modelQueueName);
    }
  }

  getChannel() {
    return this.channel;
  }
}
```

// rabbit-mq-publisher.service.ts

```
import { Injectable, Logger } from '@nestjs/common';
import { RabbitMQConnectionService } from './rabbit-mq-connection.service';

@Injectable()
export class RabbitMQPublisherService {
  private readonly logger = new Logger(RabbitMQPublisherService.name);

  constructor(
    private readonly rabbitMQConnectionService: RabbitMQConnectionService,
  ) {}

  async addMessageToQueue(queue: string, messageData: unknown) {
    const channel = this.rabbitMQConnectionService.getChannel();
    await this.rabbitMQConnectionService.assertQueue(queue);
    channel.sendToQueue(queue, this.convertData(messageData), {
      persistent: true,
    });

    this.logger.debug(
      `Message sent to queue (${queue}) with task data ${messageData}`,
    );
  }

  convertData(data: unknown) {
    return Buffer.from(JSON.stringify(data));
  }
}
```

// rabbit-mq-consumer.service.ts

```
import { Injectable, Logger } from '@nestjs/common';
import { DiagnosticsService } from 'src/modules/diagnostics/diagnostics.service';
import { RabbitMQConnectionService } from './rabbit-mq-connection.service';
import { Channel, ConsumeMessage } from 'amqplib';
import { MessageQueueName } from './enums';
import { DiagnosticResultData } from 'src/modules/diagnostics/types';

@Injectable()
export class RabbitMQConsumerService {
  private readonly logger = new Logger(RabbitMQConsumerService.name);

  constructor(
    private readonly rabbitMQConnectionService: RabbitMQConnectionService,
    private readonly diagnosticsService: DiagnosticsService,
  ) {}
```

```
  async onModuleInit() {
    await this.rabbitMQConnectionService.setUpConnection();

    const channel = this.rabbitMQConnectionService.getChannel();
    await this.setUpDiagnosticResultHandler(channel);
  }

  async setUpDiagnosticResultHandler(channel: Channel) {
    const queue = MessageQueueName.DIAGNOSTIC_RESULT;
    await this.rabbitMQConnectionService.assertQueue(queue);
    await channel.consume(
      queue,
      this.handleDiagnosticResultMessage.bind(this, channel),
    );
  }

  async handleDiagnosticResultMessage(channel: Channel, msg: ConsumeMessage) {
    let data = null;
    try {
      data = this.getMessageData<DiagnosticResultData>(msg);
      await this.diagnosticsService.handleDiagnosticResult(data);
      channel.ack(msg);
    } catch (err) {
      this.handleError(err, msg, data);
    }
  }

  getMessageData<T extends object>(msg: ConsumeMessage): T {
    return JSON.parse(msg.content.toString());
  }

  handleError(err: Error, msg?: ConsumeMessage, data?: unknown) {
    this.logger.error(
      `Error ${err}, when handling message:`,
      msg,
      'Message data: ',
      data,
    );
  }
}
```

// storage.module.ts

```
import { Module } from '@nestjs/common';
import { StorageService } from './storage.service';
import { AppConfigModule } from 'src/config';

@Module({
  imports: [AppConfigModule],
  providers: [StorageService],
  exports: [StorageService],
})
export class StorageModule {}
```

// storage.service.ts

```
import { BadRequestException, Injectable, Logger } from '@nestjs/common';
import { Client } from 'minio';
import {
  FILE_DEFAULT_CONTENT_TYPE,
  STORAGE_ROOT_DIRECTORY,
} from './storage.constants';

import { AppConfigService } from 'src/config/app-config.service';
```

```
import { ErrorMessagesEnum } from 'src/common/enums';
import { IMultipartFile } from './interfaces';
import { join, parse } from 'node:path';
import { error } from 'node:console';
import { FileEntity } from 'src/modules/files/file.entity';

@Injectable()
export class StorageService {
  private readonly logger = new Logger(StorageService.name);

  private readonly minioClient: Client;
  private readonly bucket: string;

  constructor(private readonly configService: AppConfigService) {
    this.minioClient = new Client({
      endPoint: this.configService.get('MINIO_HOST'),
      port: parseInt(this.configService.get('MINIO_PORT'), 10),
      useSSL: this.configService.get('MINIO_USE_SSL'),
      accessKey: this.configService.get('MINIO_ACCESS_KEY'),
      secretKey: this.configService.get('MINIO_SECRET_KEY'),
    });
    this.bucket = this.configService.get('MINIO_BUCKET');
  }

  public async selectOneData(path: string): Promise<Partial<IMultipartFile>> {
    const fileObj = await this.minioClient.getObject(this.bucket, path);
    const fileStat = await this.minioClient.statObject(this.bucket, path);
    const mimetype = fileStat.metaData['content-type'];

    const chunks = [];
    for await (const chunk of fileObj) {
      chunks.push(chunk);
    }
    const data = Buffer.concat(chunks);
    return { data, mimetype };
  }

  public async createOne(
    file: IMultipartFile,
    pathToFile: string,
  ): Promise<Promise<Partial<FileEntity>>> {
    const { data, filename, mimetype } = file;
    const { ext } = parse(filename);
    const actualFileName = Date.now() + '-' + filename;
    const pathInStorage = join(
      STORAGE_ROOT_DIRECTORY,
      pathToFile,
      actualFileName,
    ).replace(/\\/g, '/');
    const uploadMetadata = {
      'Content-Type': mimetype ?? FILE_DEFAULT_CONTENT_TYPE,
    };

    try {
      await this.minioClient.putObject(
        this.bucket,
        pathInStorage,
        data,
        undefined,
        uploadMetadata,
      );
      this.logger.verbose(
        `MINIO UPLOADED FILE | BUCKET: ${this.bucket}; PATH TO FILE: ${pathInStorage}.
Result: SUCCESS`,
        error,
```

```
      );
      return {
        fileName: filename,
        fileExt: ext,
        fileNameWithExt: actualFileName,
        pathInStorage,
        mimetype,
      };
    } catch (error) {
      this.logger.error(
        `MINIO UPLOAD ERROR | BUCKET: ${this.bucket}; PATH TO FILE: ${pathInStorage}.
Result: FAILED`,
        error,
      );
      await this.deleteOne(pathInStorage);
      throw new BadRequestException(ErrorMessagesEnum.FILE_UPLOAD_ERROR);
    }
  }

  public async deleteOne(pathInStorage: string): Promise<void> {
    await this.minioClient
      .removeObject(this.bucket, pathInStorage)
      .catch(() => {
        throw new BadRequestException(ErrorMessagesEnum.FILE_DELETION_ERROR);
      });
  }
}
```

// storage.constants.ts

```
export const STORAGE_ROOT_DIRECTORY = 'uploads';
export const UNCATEGORIZED_ROOT_DIRECTORY = 'uncategorized';
export const MODELS_DIRECTORY = 'models';
export const DIAGNOSTIC_IMAGE_DIRECTORY = 'diagnostic-images';

export const FILE_DEFAULT_CONTENT_TYPE = 'application/octet-stream';
```

// files.module.ts

```
import { Module } from '@nestjs/common';
import { FileEntity } from './file.entity';
import { TypeOrmModule } from '@nestjs/typeorm';
import { FilesController } from './files.controller';
import { FilesService } from './files.service';
import { StorageModule } from 'src/systems/storage';

@Module({
  imports: [StorageModule, TypeOrmModule.forFeature([FileEntity])],
  controllers: [FilesController],
  providers: [FilesService],
  exports: [FilesService],
})
export class FilesModule {}
```

// files.service.ts

```
import {
  BadRequestException,
  Injectable,
  NotFoundException,
} from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import {
  FindManyOptions,
  FindOptionsWhere,
```

```
  FindOneOptions,
  Repository,
  EntityManager,
} from 'typeorm';
import { FileEntity } from './file.entity';
import { StorageService } from 'src/systems/storage/storage.service';
import { IMultipartFile } from 'src/systems/storage/interfaces';
import { UNCATEGORIZED_ROOT_DIRECTORY } from 'src/systems/storage/storage.constants';
import { filesServiceErrorMessages } from './files.constants';

@Injectable()
export class FilesService {
  constructor(
    @InjectRepository(FileEntity)
    private readonly fileEntityRepository: Repository<FileEntity>,
    private readonly storageService: StorageService,
  ) {}

  async createOne(
    file: IMultipartFile,
    data?: Partial<FileEntity>,
    pathToFile = UNCATEGORIZED_ROOT_DIRECTORY,
    transactionManager?: EntityManager,
  ) {
    const repository = transactionManager
      ? transactionManager.getRepository<FileEntity>(
          this.fileEntityRepository.target,
        )
      : this.fileEntityRepository;

    const uploadedFileEntity = await this.storageService.createOne(
      file,
      pathToFile,
    );
    const entityLike = repository.create(uploadedFileEntity);
    const entityToCreate = repository.merge(entityLike, data);
    const { id } = await repository.save(entityToCreate).catch(() => {
      throw new BadRequestException(filesServiceErrorMessages.invalidData);
    });
    return this.findOne(
      { id },
      { loadEagerRelations: true },
      transactionManager,
    );
  }

  async findOne(
    conditions: FindOptionsWhere<FileEntity>,
    options: FindOneOptions<FileEntity> = { loadEagerRelations: true },
    transactionManager?: EntityManager,
  ): Promise<FileEntity> {
    const repository = transactionManager
      ? transactionManager.getRepository<FileEntity>(
          this.fileEntityRepository.target,
        )
      : this.fileEntityRepository;

    return repository
      .findOneOrFail({
        ...options,
        where: conditions,
      })
      .catch(() => {
        throw new NotFoundException(filesServiceErrorMessages.entityNotFound);
      });
```

```
  }

  async findAll(
    options: FindManyOptions<FileEntity> = { loadEagerRelations: true },
  ): Promise<FileEntity[]> {
    return this.fileEntityRepository.find(options).catch(() => {
      throw new NotFoundException(filesServiceErrorMessages.entitiesNotFound);
    });
  }

  async deleteOne(
    conditions: FindOptionsWhere<FileEntity>,
  ): Promise<FileEntity> {
    const entity = await this.findOne(conditions);
    await this.storageService.deleteOne(entity.pathInStorage);
    return this.fileEntityRepository.remove(entity).catch(() => {
      throw new NotFoundException(filesServiceErrorMessages.entityNotFound);
    });
  }
}
```

// file.entity.ts

```
import { ConfigService } from '@nestjs/config';
import { ApiProperty } from '@nestjs/swagger';
import { Expose } from 'class-transformer';
import * as path from 'path';
import { CommonEntity } from 'src/common/entities';
import { AppConfigService } from 'src/config/app-config.service';
import { Column, Entity } from 'typeorm';

const appConfigService = new AppConfigService(new ConfigService());

@Entity('files')
export class FileEntity extends CommonEntity {
  @ApiProperty({ type: 'string', maxLength: 256, required: true })
  @Column({ type: 'varchar', length: 256 })
  public readonly fileName: string;

  @ApiProperty({ type: 'string', maxLength: 256, required: true })
  @Column({ type: 'varchar', length: 256 })
  public readonly fileExt: string;

  @ApiProperty({ type: 'string', maxLength: 512, required: true })
  @Column({ type: 'varchar', length: 512 })
  public readonly fileNameWithExt: string;

  @ApiProperty({ type: 'string', maxLength: 512, required: true })
  @Column({ type: 'varchar', length: 512 })
  public readonly pathInStorage: string;

  @ApiProperty({ type: 'string', maxLength: 256, required: true })
  @Column({ type: 'varchar', length: 256 })
  public readonly mimetype: string;

  @Expose()
  @ApiProperty({ readOnly: true })
  get src(): string {
    if (!this.pathInStorage) return null;
    const filePath = path
      .join(appConfigService.get('CDN'), this.pathInStorage)
      .replace(/\\/g, '/');

    return new URL(filePath).toString();
  }
```

```
}

// create-file.dto.ts

import { PayloadTooLargeException } from '@nestjs/common';
import { ApiProperty } from '@nestjs/swagger';
import { Transform } from 'class-transformer';
import { IsNotEmpty, IsObject } from 'class-validator';
import { ErrorMessagesEnum } from 'src/common/enums';
import { IMultipartFile } from 'src/systems/storage/interfaces';

export class CreateFileDto {
  @IsNotEmpty()
  @IsObject()
  @Transform(({ value: [file] }) => {
    if (file.limit) {
      throw new PayloadTooLargeException(ErrorMessagesEnum.FILE_TOO_LARGE);
    }
    return file;
  })
  @ApiProperty({
    required: true,
    format: 'binary',
    type: String,
  })
  public readonly file: IMultipartFile;
}

// users.module.ts

import { Module } from '@nestjs/common';
import { UserEntity } from './user.entity';
import { TypeOrmModule } from '@nestjs/typeorm';
import { UsersService } from './users.service';
import { UsersController } from './users.controller';

@Module({
  imports: [TypeOrmModule.forFeature([UserEntity])],
  controllers: [UsersController],
  providers: [UsersService],
  exports: [UsersService],
})
export class UsersModule {}

// users.service.ts


import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { BaseService } from 'src/common/services';
import { FindManyOptions, FindOptionsWhere, Repository } from 'typeorm';
import { usersServiceErrorMessages } from './users.constants';
import { UserEntity } from './user.entity';

@Injectable()
export class UsersService extends BaseService<UserEntity> {
  constructor(
    @InjectRepository(UserEntity)
    private readonly userEntityRepository: Repository<UserEntity>,
  ) {
    super(userEntityRepository, usersServiceErrorMessages);
  }

  async selectUsersProfiles(
    options?: FindManyOptions<UserEntity>,
```

```
  ): Promise<UserEntity[]> {
    const users = await this.findAll(options);
    return users.map((user) => ({
      ...user,
      age: null,
      sexAtBirth: null,
    })) as UserEntity[];
  }

  async selectUserProfile(
    conditions: FindOptionsWhere<UserEntity>,
  ): Promise<UserEntity> {
    const user = await this.findOne(conditions);

    return { ...user, age: null, sexAtBirth: null } as UserEntity;
  }
}
```

// user.entity.ts

```
import { ApiProperty, ApiHideProperty } from '@nestjs/swagger';
import { Entity, Column, BeforeInsert, BeforeUpdate, OneToMany } from 'typeorm';
import { Exclude } from 'class-transformer';
import { CommonEntity } from 'src/common/entities';
import * as bcrypt from 'bcrypt';
import { PASSWORD_SALT_ROUNDS } from 'src/common/constants';
import { SexAtBirthEnum, UserRoleEnum } from './enums';
import { RefreshTokenEntity } from '../refresh-tokens/refresh-token.entity';

@Entity({ name: 'users' })
export class UserEntity extends CommonEntity {
  @ApiProperty({ type: 'string', maxLength: 128 })
  @Column({ length: 128 })
  name: string;

  @ApiHideProperty()
  @Exclude()
  @Column({ length: 256 })
  password: string;

  @ApiProperty({ type: 'string', maxLength: 256, uniqueItems: true })
  @Column({ length: 256, unique: true })
  email: string;

  @ApiProperty({ enum: UserRoleEnum, default: UserRoleEnum.PATIENT })
  @Column({
    enum: UserRoleEnum,
    default: UserRoleEnum.PATIENT,
    nullable: false,
  })
  role: UserRoleEnum;

  @ApiProperty({ type: 'integer', nullable: true })
  @Column({ type: 'integer', nullable: true })
  public age: number;

  @ApiProperty({
    enum: SexAtBirthEnum,
    nullable: true,
  })
  @Column({
    type: 'enum',
    enum: SexAtBirthEnum,
    nullable: true,
  })
```

```
  public readonly sexAtBirth: SexAtBirthEnum;

  @ApiHideProperty()
  @OneToMany(() => RefreshTokenEntity, ({ user }) => user)
  public readonly refreshTokens: RefreshTokenEntity[];

  @BeforeInsert()
  @BeforeUpdate()
  public async hashPassword() {
    if (this.password)
      this.password = await bcrypt.hash(this.password, PASSWORD_SALT_ROUNDS);
  }
}
```

// users.controller.ts

```
import {
  ClassSerializerInterceptor,
  Controller,
  Get,
  Param,
  UseGuards,
  UseInterceptors,
} from '@nestjs/common';
import { UsersService } from './users.service';
import { UserEntity } from './user.entity';
import { IdDto } from 'src/common/dto';
import { ApiBearerAuth, ApiTags } from '@nestjs/swagger';
import { AccessTokenGuard } from '../auth/guards';
import { Role } from '../auth/decorators';
import { UserRoleEnum } from './enums';

@ApiTags('users')
@Controller('users')
@UseGuards(AccessTokenGuard)
@ApiBearerAuth()
@UseInterceptors(ClassSerializerInterceptor)
export class UsersController {
  constructor(private readonly usersService: UsersService) {}

  @Role(UserRoleEnum.ADMIN)
  @Get('profiles')
  findAll(): Promise<UserEntity[]> {
    return this.usersService.selectUsersProfiles({
      where: { role: UserRoleEnum.PATIENT },
    });
  }

  @Role(UserRoleEnum.ADMIN)
  @Get('profile/:id')
  findOne(@Param() conditions: IdDto): Promise<UserEntity> {
    return this.usersService.selectUserProfile(conditions);
  }
}
```

// refresh-tokens.module

```
import { Module } from '@nestjs/common';
import { RefreshTokenEntity } from './refresh-token.entity';
import { TypeOrmModule } from '@nestjs/typeorm';
import { RefreshTokensService } from './refresh-tokens.service';
import { AppConfigModule } from 'src/config';
import { UsersModule } from '../users';

@Module({
```

```
  imports: [
    UsersModule,
    TypeOrmModule.forFeature([RefreshTokenEntity]),
    AppConfigModule,
  ],
  providers: [RefreshTokensService],
  exports: [RefreshTokensService],
})
export class RefreshTokensModule {}

// refresh-tokens.service.ts

import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { FindOptionsWhere } from 'typeorm';
import { Repository } from 'typeorm';
import { RefreshTokenEntity } from './refresh-token.entity';
import { refreshTokensServiceErrorMessages } from './refresh-tokens.constants';
import { BaseService } from 'src/common/services';
import * as ms from 'ms';
import { AppConfigService } from 'src/config/app-config.service';
import { UsersService } from '../users/users.service';
import { UserEntity } from '../users/user.entity';
import { USER_MAX_SESSIONS } from '../auth/auth.constants';

@Injectable()
export class RefreshTokensService extends BaseService<RefreshTokenEntity> {
  tokenExpirationTime = this.appConfigService.get<string>(
    'JWT_REFRESH_EXPIRATION_TIME',
  );

  constructor(
    @InjectRepository(RefreshTokenEntity)
    private readonly refreshTokenEntityRepository: Repository<RefreshTokenEntity>,
    private readonly appConfigService: AppConfigService,
    private readonly usersService: UsersService,
  ) {
    super(refreshTokenEntityRepository, refreshTokensServiceErrorMessages);
  }

  async createToken(
    condition: FindOptionsWhere<UserEntity>,
    data: Partial<RefreshTokenEntity>,
  ): Promise<RefreshTokenEntity> {
    const user = await this.usersService.findOne(condition);

    const tokenDuration = ms(this.tokenExpirationTime);
    const currentDateTime = new Date();
    const expiresAt = new Date(currentDateTime.getTime() + tokenDuration);

    const refreshTokenModel: Partial<RefreshTokenEntity> = {
      ...data,
      user,
      expiresAt,
    };

    return this.createOne(refreshTokenModel);
  }

  async deleteExpiredTokens(condition: FindOptionsWhere<RefreshTokenEntity>) {
    const expiredTokens = await this.refreshTokenEntityRepository.find({
      where: condition,
      order: { createdAt: 'DESC' },
      skip: USER_MAX_SESSIONS,
    });
```

```
    const tokenToDeleteIds = expiredTokens.map(({ id }) => id);
    if (tokenToDeleteIds.length) {
      await this.refreshTokenEntityRepository.delete(tokenToDeleteIds);
    }
  }
}
```

// refresh-token.entity.ts

```
import { ApiProperty } from '@nestjs/swagger';
import { Entity, Column, ManyToOne, JoinColumn, BeforeInsert } from 'typeorm';
import { CommonEntity } from 'src/common/entities';
import { UserEntity } from '../users/user.entity';
import * as bcrypt from 'bcrypt';
import { REFRESH_TOKEN_SALT_ROUNDS } from 'src/common/constants';

@Entity({ name: 'refresh-tokens' })
export class RefreshTokenEntity extends CommonEntity {
  @ApiProperty({ type: 'string', maxLength: 512, uniqueItems: true })
  @Column({ length: 512, unique: true })
  value: string;

  @ApiProperty({ type: () => UserEntity })
  @ManyToOne(() => UserEntity, { onDelete: 'CASCADE' })
  @JoinColumn()
  user: Partial<UserEntity>;

  @ApiProperty({ type: 'string', readOnly: true, format: 'date-time' })
  @Column({ readonly: true })
  expiresAt: Date;

  @BeforeInsert()
  public async hashRefreshToken() {
    if (this.value) {
      this.value = await bcrypt.hash(this.value, REFRESH_TOKEN_SALT_ROUNDS);
    }
  }
}
```

// auth.module.ts

```
import { Module } from '@nestjs/common';
import { AuthController } from './auth.controller';
import { AuthService } from './auth.service';
import { UsersModule } from '../users';
import { RefreshTokensModule } from '../refresh-tokens';
import { JwtModule } from '@nestjs/jwt';
import { AppConfigModule } from 'src/config';
import { AccessTokenStrategy, RefreshTokenStrategy } from './strategies';

@Module({
  imports: [
    UsersModule,
    RefreshTokensModule,
    JwtModule.register({}),
    AppConfigModule,
  ],
  controllers: [AuthController],
  providers: [AuthService, AccessTokenStrategy, RefreshTokenStrategy],
})
export class AuthModule {}
```

// auth.service.ts

```
import {
  ConflictException,
  Injectable,
  UnauthorizedException,
} from '@nestjs/common';
import { CreateUserDto } from '../users/dto';
import { UsersService } from '../users/users.service';
import { RefreshTokensService } from '../refresh-tokens/refresh-tokens.service';
import * as bcrypt from 'bcrypt';
import { AuthTokens, AccessJwtPayload, RefreshJwtPayload } from './types';
import { JwtService } from '@nestjs/jwt';
import { AppConfigService } from 'src/config/app-config.service';
import { UserEntity } from '../users/user.entity';
import { FindOptionsWhere } from 'typeorm';
import { v4 as uuidv4 } from 'uuid';
import { SingInDto } from './dto';
import { RefreshTokenEntity } from '../refresh-tokens/refresh-token.entity';
import { ErrorMessagesEnum } from 'src/common/enums';
import { generateRandomString } from 'src/common/helpers';
import { UserRoleEnum } from '../users/enums';

@Injectable()
export class AuthService {
  private readonly jwtAccessSecret =
    this.appConfigService.get<string>('JWT_ACCESS_SECRET');
  private readonly jwtAccessExpiry = this.appConfigService.get<string>(
    'JWT_ACCESS_EXPIRATION_TIME',
  );
  private readonly jwtRefreshSecret =
    this.appConfigService.get<string>('JWT_REFRESH_SECRET');
  private readonly jwtRefreshExpiry = this.appConfigService.get<string>(
    'JWT_REFRESH_EXPIRATION_TIME',
  );

  constructor(
    private readonly appConfigService: AppConfigService,
    private readonly usersService: UsersService,
    private readonly jwtService: JwtService,
    private readonly refreshTokensService: RefreshTokensService,
  ) {}

  async signUp(createUserDto: CreateUserDto): Promise<AuthTokens> {
    const userExists = await this.usersService
      .findOne({ email: createUserDto.email })
      .catch(() => null);

    if (userExists) {
      throw new ConflictException(ErrorMessagesEnum.USER_ALREADY_EXIST);
    }

    const user = await this.usersService.createOne(createUserDto);
    return this.createUserTokens(user);
  }

  async signIn(data: SingInDto): Promise<AuthTokens> {
    const user = await this.usersService
      .findOne({ email: data.email })
      .catch(() => {
        throw new UnauthorizedException(ErrorMessagesEnum.INVALID_CREDENTIALS);
      });

    const correctPassword = await bcrypt.compare(data.password, user.password);

    if (!correctPassword) {
      throw new UnauthorizedException(ErrorMessagesEnum.INVALID_CREDENTIALS);
```

```
  }

  const tokens = await this.createUserTokens(user);

  await this.refreshTokensService.deleteExpiredTokens({
    user: { id: user.id },
  });

  return tokens;
}

async signOut(refreshTokenId: FindOptionsWhere<RefreshTokenEntity>) {
  return this.refreshTokensService.deleteOne(refreshTokenId);
}

async refreshTokens(
  condition: FindOptionsWhere<UserEntity>,
  refreshTokenId: FindOptionsWhere<RefreshTokenEntity>,
): Promise<AuthTokens> {
  const user = await this.usersService.findOne(condition);

  await this.refreshTokensService.deleteOne(refreshTokenId);

  const tokens = await this.createUserTokens(user);

  return tokens;
}

async generateTokens(
  user: Partial<UserEntity>,
  refreshTokenId: string,
): Promise<AuthTokens> {
  const accessTokenPayload: AccessJwtPayload = {
    id: user.id,
    role: user.role,
    refreshTokenId,
  };

  const refreshTokenPayload: RefreshJwtPayload = {
    ...accessTokenPayload,
    refreshTokenId,
  };

  const [accessToken, refreshToken] = await Promise.all([
    this.jwtService.signAsync(accessTokenPayload, {
      secret: this.jwtAccessSecret,
      expiresIn: this.jwtAccessExpiry,
    }),
    this.jwtService.signAsync(refreshTokenPayload, {
      secret: this.jwtRefreshSecret,
      expiresIn: this.jwtRefreshExpiry,
    }),
  ]);

  const tokens: AuthTokens = {
    accessToken,
    refreshToken,
  };

  return tokens;
}

async createUserTokens(user: Partial<UserEntity>): Promise<AuthTokens> {
  const refreshTokenId = uuidv4();
  const tokens = await this.generateTokens(user, refreshTokenId);
```

```
    await this.refreshTokensService.createToken(
      { id: user.id },
      { value: tokens.refreshToken, id: refreshTokenId },
    );

    return tokens;
  }

  async findUser(
    conditions: FindOptionsWhere<UserEntity>,
  ): Promise<UserEntity> {
    return this.usersService.findOne(conditions);
  }

  async updateUser(
    conditions: FindOptionsWhere<UserEntity>,
    data: Partial<UserEntity>,
  ): Promise<UserEntity> {
    return this.usersService.updateOne(conditions, data);
  }

  async updatePassword(
    conditions: FindOptionsWhere<UserEntity>,
    password: string,
  ): Promise<AuthTokens> {
    const updatedUser = await this.usersService.updateOne(conditions, {
      password,
    });
    return this.createUserTokens(updatedUser);
  }

  generateTemporaryPassword(length = 6): string {
    return generateRandomString(length);
  }

  async createPatient(
    data: Partial<UserEntity>,
  ): Promise<{ user: UserEntity; password: string }> {
    const password = this.generateTemporaryPassword();
    const user = await this.usersService.createOne({
      ...data,
      role: UserRoleEnum.PATIENT,
      password,
    });

    return { user, password };
  }
}
```

// role.decorator.ts

```
import { SetMetadata } from '@nestjs/common';
import { OperationObject } from '@nestjs/swagger/dist/interfaces/open-api-spec.interface';
import { DECORATORS } from '@nestjs/swagger/dist/constants';
import { UserRoleEnum } from 'src/modules/users/enums';
import { ApiBearerAuth, ApiOperation } from '@nestjs/swagger';

export const Role =
  (...roles: UserRoleEnum[]) =>
  (
    target: Record<string, any>,
    key?: string | symbol,
    descriptor?: TypedPropertyDescriptor<any>,
  ): void => {
    const operation: OperationObject = Reflect.getMetadata(
```

```
      DECORATORS.API_OPERATION,
      descriptor.value,
    );

    if (!operation)
      ApiOperation({ summary: `[ROLE: ${roles.join(', ')}]` })(
        target,
        key,
        descriptor,
      );
    else
      operation.summary = `[ROLE: ${roles.join(', ')}]${operation.summary && '; ' +
operation.summary}`;

    SetMetadata('roles', roles)(target, key, descriptor);
    ApiBearerAuth()(target, key, descriptor);
  };

// user.decorator.ts

import { createParamDecorator, ExecutionContext } from '@nestjs/common';
import { UserEntity } from 'src/modules/users/user.entity';

export const User = createParamDecorator(
  (data: unknown, ctx: ExecutionContext): Partial<UserEntity> => {
    const request = ctx.switchToHttp().getRequest();
    return request.user;
  },
);

// access-token.guard.ts

import {
  ExecutionContext,
  ForbiddenException,
  Injectable,
  UnauthorizedException,
} from '@nestjs/common';
import { Reflector } from '@nestjs/core';
import { AuthGuard } from '@nestjs/passport';
import { ErrorMessagesEnum } from 'src/common/enums';
import { UserRoleEnum } from 'src/modules/users/enums';
import { UserEntity } from 'src/modules/users/user.entity';

@Injectable()
export class AccessTokenGuard extends AuthGuard('jwt') {
  constructor(private readonly reflector: Reflector) {
    super();
  }

  public handleRequest(
    _err: Error,
    user: UserEntity,
    info: Error,
    ctx: ExecutionContext,
  ): UserEntity | any {
    if (_err || info || !user) {
      throw new UnauthorizedException(ErrorMessagesEnum.UNAUTHORIZED);
    }

    const allowedRoles = this.reflector.get<UserRoleEnum[]>(
      'roles',
      ctx.getHandler(),
    );
```

```
    if (!allowedRoles || allowedRoles.includes(user.role)) return user;
    throw new ForbiddenException(ErrorMessagesEnum.FORBIDDEN);
  }
}

// refresh-token.guard.ts

import { Injectable } from '@nestjs/common';
import { AuthGuard } from '@nestjs/passport';

@Injectable()
export class RefreshTokenGuard extends AuthGuard('jwt-refresh') {}

// access-token.strategy.ts

import { Injectable } from '@nestjs/common';
import { PassportStrategy } from '@nestjs/passport';
import { ExtractJwt, Strategy } from 'passport-jwt';
import { AppConfigService } from 'src/config/app-config.service';
import { AccessJwtPayload } from '../types';
import { UsersService } from 'src/modules/users/users.service';

@Injectable()
export class AccessTokenStrategy extends PassportStrategy(Strategy, 'jwt') {
  constructor(
    private readonly appConfigService: AppConfigService,
    private readonly usersService: UsersService,
  ) {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      secretOrKey: appConfigService.get<string>('JWT_ACCESS_SECRET'),
    });
  }

  async validate({ id, role, refreshTokenId }: AccessJwtPayload) {
    return this.usersService.findOne(
      { id, role, refreshTokens: { id: refreshTokenId } },
      {
        loadEagerRelations: false,
      },
    );
  }
}

// refresh-token.strategy.ts

import { PassportStrategy } from '@nestjs/passport';
import { ExtractJwt, Strategy } from 'passport-jwt';
import { Injectable } from '@nestjs/common';
import { AppConfigService } from 'src/config/app-config.service';
import { RefreshJwtPayload } from '../types';
import { UsersService } from 'src/modules/users/users.service';
import { RefreshTokensService } from 'src/modules/refresh-tokens/refresh-tokens.service';

@Injectable()
export class RefreshTokenStrategy extends PassportStrategy(
  Strategy,
  'jwt-refresh',
) {
  constructor(
    private readonly appConfigService: AppConfigService,
    private readonly usersService: UsersService,
    private readonly refreshTokensService: RefreshTokensService,
  ) {
    super({
```

```
      jwtFromRequest: ExtractJwt.fromBodyField('refreshToken'),
      secretOrKey: appConfigService.get<string>('JWT_REFRESH_SECRET'),
    });
  }

  async validate({ id, role, refreshTokenId }: RefreshJwtPayload) {
    const userData = await this.usersService.findOne(
      { id, role },
      {
        loadEagerRelations: false,
      },
    );
    const tokenData = await this.refreshTokensService.findOne(
      {
        id: refreshTokenId,
        user: { id: userData.id },
      },
      {
        select: { id: true },
        loadEagerRelations: false,
      },
    );

    return { ...userData, refreshTokenId: tokenData.id };
  }
}
```

// auth.controller.ts

```
import {
  Body,
  ClassSerializerInterceptor,
  Controller,
  Get,
  Patch,
  Post,
  UseGuards,
  UseInterceptors,
} from '@nestjs/common';
import { AuthService } from './auth.service';
import {
  CreatePatientDto,
  RefreshTokenDto,
  SingInDto,
  UpdatePasswordDto,
} from './dto';
import { ApiBearerAuth, ApiBody, ApiTags } from '@nestjs/swagger';
import { AccessTokenGuard, RefreshTokenGuard } from 'src/modules/auth/guards';
import { AuthTokens, JwtPayloadUser, RefreshJwtPayload } from './types';
import { UserEntity } from '../users/user.entity';
import { Role, User } from './decorators';
import { UserRoleEnum } from '../users/enums';
import { UpdateProfileDto } from './dto/update-profile.dto';

@ApiTags('auth')
@Controller('auth')
@UseInterceptors(ClassSerializerInterceptor)
export class AuthController {
  constructor(private readonly authService: AuthService) {}

  @Post('sign-in')
  signIn(@Body() data: SingInDto): Promise<AuthTokens> {
    return this.authService.signIn(data);
  }
```

```
  @UseGuards(RefreshTokenGuard)
  @Post('sign-out')
  @ApiBody({ type: RefreshTokenDto })
  signOut(@User() user: RefreshJwtPayload) {
    return this.authService.signOut({ id: user.refreshTokenId });
  }

  @UseGuards(RefreshTokenGuard)
  @Post('tokens/refresh')
  @ApiBody({ type: RefreshTokenDto })
  refreshTokens(@User() user: RefreshJwtPayload): Promise<AuthTokens> {
    return this.authService.refreshTokens(
      { id: user.id },
      { id: user.refreshTokenId },
    );
  }

  @ApiBearerAuth()
  @UseGuards(AccessTokenGuard)
  @Get('profile')
  getProfile(@User() user: JwtPayloadUser): Promise<UserEntity> {
    return this.authService.findUser({ id: user.id });
  }

  @ApiBearerAuth()
  @UseGuards(AccessTokenGuard)
  @Patch('profile')
  updateProfile(
    @User() user: JwtPayloadUser,
    @Body() data: UpdateProfileDto,
  ): Promise<UserEntity> {
    return this.authService.updateUser({ id: user.id }, data);
  }

  @ApiBearerAuth()
  @UseGuards(AccessTokenGuard)
  @Patch('password')
  updatePassword(
    @User() user: JwtPayloadUser,
    @Body() data: UpdatePasswordDto,
  ): Promise<AuthTokens> {
    return this.authService.updatePassword({ id: user.id }, data.password);
  }

  @ApiBearerAuth()
  @UseGuards(AccessTokenGuard)
  @Role(UserRoleEnum.ADMIN)
  @Post('patients')
  createPatient(
    @Body() data: CreatePatientDto,
  ): Promise<{ user: UserEntity; password: string }> {
    return this.authService.createPatient(data);
  }
}
```

// diagnostic-types.service.ts

```
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { BaseService } from 'src/common/services';
import { Repository } from 'typeorm';
import { diagnosticTypesServiceErrorMessages } from './diagnostic-types.constants';
import { DiagnosticTypeEntity } from './diagnostic-type.entity';

@Injectable()
```

```
export class DiagnosticTypesService extends BaseService<DiagnosticTypeEntity> {
  constructor(
    @InjectRepository(DiagnosticTypeEntity)
    private readonly diagnosticTypeEntityRepository: Repository<DiagnosticTypeEntity>,
  ) {
    super(diagnosticTypeEntityRepository, diagnosticTypesServiceErrorMessages);
  }
}
```

// diagnostic-type.entity.ts

```
import { ApiProperty } from '@nestjs/swagger';
import { Entity, Column } from 'typeorm';
import { CommonEntity } from 'src/common/entities';

@Entity({ name: 'diagnostic-types' })
export class DiagnosticTypeEntity extends CommonEntity {
  @ApiProperty({ type: 'string', maxLength: 128 })
  @Column({ length: 128, nullable: false })
  name: string;

  @ApiProperty({
    type: 'string',
    maxLength: 500,
    required: false,
    nullable: true,
  })
  @Column({ length: 500, nullable: true })
  description: string;
}
```

// diagnostic-models.service.ts

```
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { BaseService } from 'src/common/services';
import {
  EntityManager,
  FindManyOptions,
  FindOneOptions,
  FindOptionsWhere,
  Repository,
} from 'typeorm';
import { diagnosticModelsServiceErrorMessages } from './diagnostic-models.constants';
import { DiagnosticModelEntity } from './diagnostic-model.entity';
import { IMultipartFile } from 'src/systems/storage/interfaces';
import { DiagnosticModelVersionEntity } from '../diagnostic-model-versions/diagnostic-model-
version.entity';
import { FilesService } from '../files/files.service';
import { join } from 'node:path';
import { MODELS_DIRECTORY } from 'src/systems/storage/storage.constants';
import { DiagnosticModelVersionsService } from '../diagnostic-model-versions/diagnostic-
model-versions.service';
import { DiagnosticModelStatus } from './enums';
import { DiagnosticModelVersionStatus } from '../diagnostic-model-versions/enums';

@Injectable()
export class DiagnosticModelsService extends BaseService<DiagnosticModelEntity> {
  constructor(
    @InjectRepository(DiagnosticModelEntity)
    private readonly diagnosticModelEntityRepository: Repository<DiagnosticModelEntity>,
    private readonly filesService: FilesService,
    private readonly diagnosticModelVersionsService: DiagnosticModelVersionsService,
  ) {
    super(
```

```
    diagnosticModelEntityRepository,
    diagnosticModelsServiceErrorMessages,
  );
}

async findOneWithVersions(
  conditions: FindOptionsWhere<DiagnosticModelEntity>,
  options?: FindOneOptions<DiagnosticModelEntity>,
  transactionManager?: EntityManager,
): Promise<DiagnosticModelEntity> {
  return this.findOne(
    conditions,
    {
      ...options,
      loadEagerRelations: false,
      relations: { versions: { file: true } },
      order: { versions: { version: 'DESC' } },
    },
    transactionManager,
  );
}

async findAllWithVersions(
  options?: FindManyOptions<DiagnosticModelEntity>,
  transactionManager?: EntityManager,
): Promise<DiagnosticModelEntity[]> {
  return this.findAll(
    {
      ...options,
      loadEagerRelations: false,
      relations: { versions: { file: true } },
      order: { versions: { version: 'DESC' } },
    },
    transactionManager,
  );
}

async selectAvailableModels(
  options?: FindManyOptions<DiagnosticModelEntity>,
  transactionManager?: EntityManager,
): Promise<DiagnosticModelEntity[]> {
  const models = await this.findAll(
    {
      ...options,
      where: {
        status: DiagnosticModelStatus.ENABLED,
        versions: {
          status: DiagnosticModelVersionStatus.ENABLED,
        },
      },
      relations: { versions: { file: true } },
      order: {
        versions: {
          version: 'DESC',
        },
      },
    },
    transactionManager,
  );

  const filteredModels = models.filter((model) => {
    if (model.versions && model.versions.length > 0) {
      model.versions = [model.versions[0]];
      return true;
    }
```

```
    return false;
  });

  return filteredModels;
}

async selectAvailableModelVersion(
  conditions: FindOptionsWhere<DiagnosticModelEntity>,
  options?: FindOneOptions<DiagnosticModelEntity>,
  transactionManager?: EntityManager,
): Promise<DiagnosticModelEntity> {
  const model = await this.findOneWithVersions(
    {
      ...conditions,
      status: DiagnosticModelStatus.ENABLED,
      versions: { status: DiagnosticModelVersionStatus.ENABLED },
    },
    options,
    transactionManager,
  );

  model.versions = [model.versions[0]];
  return model;
}

async uploadModelVersion(
  conditions: FindOptionsWhere<DiagnosticModelEntity>,
  file: IMultipartFile,
  versionData: Partial<DiagnosticModelVersionEntity>,
): Promise<DiagnosticModelEntity> {
  return this.diagnosticModelEntityRepository.manager.transaction(
    async (transaction) => {
      const { id, versions = [] } = await this.findOneWithVersions(
        conditions,
        undefined,
        transaction,
      );

      const filePath = this.getPathToModelVersionFile(id);
      const newVersionFile = await this.filesService.createOne(
        file,
        {},
        filePath,
        transaction,
      );

      const lastVersion = versions?.[0]?.version ?? 0;

      await this.diagnosticModelVersionsService.createOne(
        {
          ...versionData,
          version: lastVersion + 1,
          model: { id },
          file: { id: newVersionFile.id },
        },
        transaction,
      );

      return this.findOneWithVersions({ id }, undefined, transaction);
    },
  );
}

async updateVersionStatus(
  conditions: FindOptionsWhere<DiagnosticModelVersionEntity>,
```

```
    versionData: Partial<DiagnosticModelVersionEntity>,
  ): Promise<DiagnosticModelVersionEntity> {
    return this.diagnosticModelVersionsService.updateOne(
      conditions,
      versionData,
    );
  }

  getPathToModelVersionFile(modelId: string): string {
    return join(MODELS_DIRECTORY, modelId);
  }
}
```

// diagnostic-models.controller.ts

```
import {
  Body,
  ClassSerializerInterceptor,
  Controller,
  Delete,
  Get,
  Param,
  Patch,
  Post,
  UseGuards,
  UseInterceptors,
} from '@nestjs/common';
import { DiagnosticModelsService } from './diagnostic-models.service';
import { DiagnosticModelEntity } from './diagnostic-model.entity';
import { IdDto } from 'src/common/dto';
import { CreateGenreDto, UpdateGenreDto, UploadModelVersionDto } from './dto';
import {
  ApiBearerAuth,
  ApiBody,
  ApiConsumes,
  ApiParam,
  ApiTags,
} from '@nestjs/swagger';
import { plainToInstance } from 'class-transformer';
import { AccessTokenGuard } from '../auth/guards';
import { Role } from '../auth/decorators';
import { UserRoleEnum } from '../users/enums';
import { DiagnosticModelVersionEntity } from '../diagnostic-model-versions/diagnostic-model-
version.entity';
import { UpdateModelVersionStatusDto } from './dto/update-model-version-status.dto';
import { UpdateModelStatusDto } from './dto/update-model-status.dto';

@ApiTags('diagnostic-models')
@Controller('diagnostic-models')
@UseGuards(AccessTokenGuard)
@ApiBearerAuth()
@UseInterceptors(ClassSerializerInterceptor)
export class DiagnosticModelsController {
  constructor(
    private readonly diagnosticModelsService: DiagnosticModelsService,
  ) {}

  @Role(UserRoleEnum.ADMIN)
  @Get()
  findAll(): Promise<DiagnosticModelEntity[]> {
    return this.diagnosticModelsService.findAllWithVersions();
  }

  @Get('available')
  selectAvailableModels(): Promise<DiagnosticModelEntity[]> {
```

```
  return this.diagnosticModelsService.selectAvailableModels();
}

@Get(':id/available')
@ApiParam({ name: 'id', description: 'Model id' })
selectAvailableModelVersion(
  @Param() conditions: IdDto,
): Promise<DiagnosticModelEntity> {
  return this.diagnosticModelsService.selectAvailableModelVersion(conditions);
}

@UseGuards(AccessTokenGuard)
@Role(UserRoleEnum.ADMIN)
@Get(':id')
findOne(@Param() conditions: IdDto): Promise<DiagnosticModelEntity> {
  return this.diagnosticModelsService.findOneWithVersions(conditions);
}

@Role(UserRoleEnum.ADMIN)
@Post()
createOne(
  @Body() createEntityDto: CreateGenreDto,
): Promise<DiagnosticModelEntity> {
  const model = plainToInstance(DiagnosticModelEntity, createEntityDto);

  return this.diagnosticModelsService.createOne(model);
}

@Role(UserRoleEnum.ADMIN)
@Patch(':id')
updateOne(
  @Param() conditions: IdDto,
  @Body() updateEntityDto: UpdateGenreDto,
): Promise<DiagnosticModelEntity> {
  const model = plainToInstance(DiagnosticModelEntity, updateEntityDto);

  return this.diagnosticModelsService.updateOne(conditions, model);
}

@Role(UserRoleEnum.ADMIN)
@Delete(':id')
deleteOne(@Param() conditions: IdDto): Promise<DiagnosticModelEntity> {
  return this.diagnosticModelsService.deleteOne(conditions);
}

@Role(UserRoleEnum.ADMIN)
@Post(':id/versions')
@ApiBody({ type: UploadModelVersionDto })
@ApiConsumes('multipart/form-data')
uploadModelVersion(
  @Param() conditions: IdDto,
  @Body() data: UploadModelVersionDto,
): Promise<DiagnosticModelEntity> {
  const { file, ...versionData } = data;

  return this.diagnosticModelsService.uploadModelVersion(
    conditions,
    file,
    versionData,
  );
}

@Role(UserRoleEnum.ADMIN)
@Patch(':id/status')
@ApiParam({ name: 'id', description: 'Model id' })
```

```
  updateModelStatus(
    @Param() conditions: IdDto,
    @Body() data: UpdateModelStatusDto,
  ): Promise<DiagnosticModelEntity> {
    return this.diagnosticModelsService.updateOne(conditions, data);
  }

  @Role(UserRoleEnum.ADMIN)
  @Patch('versions/:id/status')
  @ApiParam({ name: 'id', description: 'Version id' })
  updateVersionStatus(
    @Param() conditions: IdDto,
    @Body() data: UpdateModelVersionStatusDto,
  ): Promise<DiagnosticModelVersionEntity> {
    return this.diagnosticModelsService.updateVersionStatus(conditions, data);
  }
}
```

// diagnostic-model.entity.ts

```
import { ApiHideProperty, ApiProperty } from '@nestjs/swagger';
import { Entity, Column, JoinColumn, ManyToOne, OneToMany } from 'typeorm';
import { CommonEntity } from 'src/common/entities';
import { DiagnosticTypeEntity } from '../diagnostic-types/diagnostic-type.entity';
import { DiagnosticModelVersionEntity } from '../diagnostic-model-versions/diagnostic-model-
version.entity';
import { DiagnosticModelStatus } from './enums';

@Entity({ name: 'diagnostic-models' })
export class DiagnosticModelEntity extends CommonEntity {
  @ApiProperty({ type: 'string', maxLength: 128 })
  @Column({ length: 128, nullable: false })
  name: string;

  @ApiProperty({
    type: 'string',
    maxLength: 500,
    required: false,
    nullable: true,
  })
  @Column({ length: 500, nullable: true })
  description: string;

  @ApiProperty({
    type: 'MELANOMA_CUSTOM_MODEL_QUEUE',
    maxLength: 128,
    required: true,
    nullable: false,
    uniqueItems: true,
  })
  @Column({ length: 128, nullable: false, unique: true })
  queueName: string;

  @JoinColumn()
  @ApiProperty({
    type: () => DiagnosticTypeEntity,
    nullable: false,
    required: true,
  })
  @ManyToOne(() => DiagnosticTypeEntity, {
    onDelete: 'CASCADE',
    eager: false,
    nullable: false,
  })
  type: Partial<DiagnosticTypeEntity>;
```

```
  @ApiProperty({
    enum: DiagnosticModelStatus,
    default: DiagnosticModelStatus.ENABLED,
  })
  @Column({
    type: 'enum',
    enum: DiagnosticModelStatus,
    default: DiagnosticModelStatus.ENABLED,
  })
  public readonly status: DiagnosticModelStatus;

  @ApiHideProperty()
  @OneToMany(() => DiagnosticModelVersionEntity, ({ model }) => model, {
    nullable: true,
  })
  versions: DiagnosticModelVersionEntity[];
}
```

// diagnostic-model-versions.service.ts

```
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { BaseService } from 'src/common/services';
import { Repository } from 'typeorm';
import { diagnosticModelVersionsServiceErrorMessages } from './diagnostic-model-
versions.constants';
import { DiagnosticModelVersionEntity } from './diagnostic-model-version.entity';

@Injectable()
export class DiagnosticModelVersionsService extends
BaseService<DiagnosticModelVersionEntity> {
  constructor(
    @InjectRepository(DiagnosticModelVersionEntity)
    private readonly diagnosticModelVersionEntityRepository:
Repository<DiagnosticModelVersionEntity>,
  ) {
    super(
      diagnosticModelVersionEntityRepository,
      diagnosticModelVersionsServiceErrorMessages,
    );
  }
}
```

// diagnostic-model-version.entity.ts

```
import { ApiProperty } from '@nestjs/swagger';
import { Entity, Column, JoinColumn, ManyToOne, OneToOne } from 'typeorm';
import { CommonEntity } from 'src/common/entities';
import { DiagnosticModelEntity } from '../diagnostic-models/diagnostic-model.entity';
import { FileEntity } from '../files/file.entity';
import { DiagnosticModelVersionStatus } from './enums';

@Entity({ name: 'diagnostic-model-versions' })
export class DiagnosticModelVersionEntity extends CommonEntity {
  @ApiProperty({ type: 'string', maxLength: 128 })
  @Column({ length: 128, nullable: false })
  name: string;

  @ApiProperty({
    type: 'string',
    maxLength: 500,
    required: false,
    nullable: true,
  })
```

```
  @Column({ length: 500, nullable: true })
  description: string;

  @ApiProperty({ type: 'integer', required: true, example: 1 })
  @Column({ type: 'integer', nullable: false })
  public version: number;

  @JoinColumn()
  @ApiProperty({
    type: () => FileEntity,
    required: true,
    nullable: false,
  })
  @OneToOne(() => FileEntity, {
    onDelete: 'CASCADE',
    nullable: false,
    eager: true,
  })
  public readonly file: Partial<FileEntity>;

  @ApiProperty({
    enum: DiagnosticModelVersionStatus,
    default: DiagnosticModelVersionStatus.ENABLED,
  })
  @Column({
    type: 'enum',
    enum: DiagnosticModelVersionStatus,
    default: DiagnosticModelVersionStatus.ENABLED,
  })
  public readonly status: DiagnosticModelVersionStatus;

  @JoinColumn()
  @ApiProperty({
    type: () => DiagnosticModelEntity,
    nullable: false,
    required: true,
  })
  @ManyToOne(() => DiagnosticModelEntity, {
    onDelete: 'CASCADE',
    eager: false,
    nullable: false,
  })
  model: Partial<DiagnosticModelEntity>;
}


// diagnostics.service.ts

import { BadRequestException, Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { BaseService } from 'src/common/services';
import {
  Repository,
  EntityManager,
  FindOptionsWhere,
  FindManyOptions,
} from 'typeorm';
import { diagnosticsServiceErrorMessages } from './diagnostics.constants';
import { DiagnosticEntity } from './diagnostic.entity';
import { UserEntity } from '../users/user.entity';
import { IMultipartFile } from 'src/systems/storage/interfaces';
import { join } from 'path';
import { DIAGNOSTIC_IMAGE_DIRECTORY } from 'src/systems/storage/storage.constants';
import { randomUUID } from 'crypto';
import { FilesService } from '../files/files.service';
```

```
import { DiagnosticResultsService } from '../diagnostic-results/diagnostic-results.service';
import { DiagnosticModelsService } from '../diagnostic-models/diagnostic-models.service';
import { ErrorMessagesEnum } from 'src/common/enums';
import { DiagnosticModelEntity } from '../diagnostic-models/diagnostic-model.entity';
import { RabbitMQPublisherService } from 'src/systems/message-queue/rabbit-mq-
publisher.service';
import { DiagnosticResultData } from './types';

@Injectable()
export class DiagnosticsService extends BaseService<DiagnosticEntity> {
  constructor(
    @InjectRepository(DiagnosticEntity)
    private readonly diagnosticEntityRepository: Repository<DiagnosticEntity>,
    private readonly filesService: FilesService,
    private readonly diagnosticResultsService: DiagnosticResultsService,
    private readonly diagnosticModelsService: DiagnosticModelsService,
    private readonly rabbitMQPublisherService: RabbitMQPublisherService,
  ) {
    super(diagnosticEntityRepository, diagnosticsServiceErrorMessages);
  }

  async selectOneDetails(
    conditions: FindOptionsWhere<DiagnosticEntity>,
    transactionManager?: EntityManager,
  ) {
    const diagnostic = await this.findOne(
      conditions,
      {
        loadEagerRelations: false,
        relations: {
          image: true,
          results: { modelVersion: { model: { type: true } } },
        },
      },
      transactionManager,
    );

    return this.transformDiagnosticToDetailsFormat(diagnostic);
  }

  private transformDiagnosticToDetailsFormat(
    diagnostic: Partial<DiagnosticEntity>,
  ) {
    const { results, ...diagnosticData } = diagnostic;

    const types = {};
    for (const result of results) {
      const { modelVersion, ...resultData } = result;
      const { model, ...versionData } = modelVersion;
      const { type, ...modelData } = model;

      if (!types[type.id]) {
        types[type.id] = {
          ...type,
          models: {},
        };
      }
      const typeInCollection = types[type.id];

      if (!typeInCollection.models[model.id]) {
        types[type.id].models[model.id] = {
          ...modelData,
          versions: {},
        };
      }
```

```
    const modelInCollection = typeInCollection.models[model.id];
    if (!modelInCollection.versions[modelVersion.id]) {
      types[type.id].models[model.id].versions[modelVersion.id] = {
        ...versionData,
        results: {},
      };
    }
    const versionInCollection = modelInCollection.versions[modelVersion.id];

    versionInCollection.results[result.id] = resultData;
  }

  return { ...diagnosticData, types };
}

async createDiagnostic(
  user: Partial<UserEntity>,
  file: IMultipartFile,
  entity: Partial<DiagnosticEntity>,
  modelIds: string[],
) {
  return this.diagnosticEntityRepository.manager.transaction(
    async (transaction) => {
      const diagnosticId = randomUUID();
      const actualName = entity?.name ?? this.generateDefaultNameByDate();

      const filePath = this.getPathToDiagnosticImageFile(
        user.id,
        diagnosticId,
      );
      const image = await this.filesService.createOne(
        file,
        {},
        filePath,
        transaction,
      );

      await this.createOne(
        {
          ...entity,
          name: actualName,
          id: diagnosticId,
          image: { id: image.id },
          user: { id: user.id },
        },
        transaction,
      );

      await this.uploadDiagnosticByModels(
        diagnosticId,
        modelIds,
        image.pathInStorage,
        transaction,
      );

      return this.selectOneDetails({ id: diagnosticId }, transaction);
    },
  );
}

async uploadDiagnosticByModels(
  diagnosticId: string,
  modelIds: string[],
  imagePathInStorage: string,
```

```
  transactionManager?: EntityManager,
): Promise<void> {
  const uniqueModelIds = new Set(modelIds); // remove duplicate values
  let uploadedModels = 0;

  for (const modelId of uniqueModelIds) {
    const model = await this.diagnosticModelsService
      .selectAvailableModelVersion({ id: modelId }, null, transactionManager)
      .catch(() => null);

    const availableVersion = model?.versions?.[0];

    if (!availableVersion) {
      continue;
    }

    const result = await this.diagnosticResultsService.createOne(
      {
        diagnostic: { id: diagnosticId },
        modelVersion: { id: availableVersion.id },
      },
      transactionManager,
    );

    await this.rabbitMQPublisherService.addMessageToQueue(model.queueName, {
      resultId: result.id,
      imagePathInStorage,
    });

    uploadedModels++;
  }

  if (uploadedModels === 0) {
    throw new BadRequestException(
      ErrorMessagesEnum.NO_VALID_MODELS_FOR_DIAGNOSIS_SPECIFIED,
    );
  }
}

getPathToDiagnosticImageFile(userId: string, diagnosticId: string): string {
  return join(DIAGNOSTIC_IMAGE_DIRECTORY, userId, diagnosticId);
}

generateDefaultNameByDate(): string {
  const now = new Date();
  const year = now.getFullYear();
  const month = String(now.getMonth() + 1).padStart(2, '0');
  const day = String(now.getDate()).padStart(2, '0');
  const hours = String(now.getHours()).padStart(2, '0');
  const minutes = String(now.getMinutes()).padStart(2, '0');
  const seconds = String(now.getSeconds()).padStart(2, '0');

  return `diagnostic_${year}${month}${day}_${hours}${minutes}${seconds}`;
}

async selectModelQueueNames(
  options?: FindManyOptions<DiagnosticModelEntity>,
  transactionManager?: EntityManager,
): Promise<string[]> {
  const models = await this.diagnosticModelsService.selectAvailableModels(
    options,
    transactionManager,
  );

  return models.map(({ queueName }) => queueName);
```

```
  }

  async handleDiagnosticResult(data: DiagnosticResultData): Promise<void> {
    const { resultId, ...resultData } = data;
    await this.diagnosticResultsService.updateOne({ id: resultId }, resultData);
  }
}
```

// diagnostics.controller.tsts

```
import {
  Body,
  ClassSerializerInterceptor,
  Controller,
  Get,
  Param,
  Post,
  UseGuards,
  UseInterceptors,
} from '@nestjs/common';
import { DiagnosticsService } from './diagnostics.service';
import { DiagnosticEntity } from './diagnostic.entity';
import { IdDto } from 'src/common/dto';
import { CreateDiagnosticDto } from './dto';
import { ApiBearerAuth, ApiBody, ApiConsumes, ApiTags } from '@nestjs/swagger';
import { AccessTokenGuard } from '../auth/guards';
import { Role, User } from '../auth/decorators';
import { UserRoleEnum } from '../users/enums';
import { UserEntity } from '../users/user.entity';

@ApiTags('diagnostics')
@Controller('diagnostics')
@UseGuards(AccessTokenGuard)
@ApiBearerAuth()
@UseInterceptors(ClassSerializerInterceptor)
export class DiagnosticsController {
  constructor(private readonly diagnosticsService: DiagnosticsService) {}

  @Get()
  findAll(@User() user: Partial<UserEntity>): Promise<DiagnosticEntity[]> {
    return this.diagnosticsService.findAll({
      where: { user: { id: user.id } },
      loadEagerRelations: true,
    });
  }

  @Get(':id')
  findOne(@User() user: Partial<UserEntity>, @Param() conditions: IdDto) {
    return this.diagnosticsService.selectOneDetails({
      ...conditions,
      user: { id: user.id },
    });
  }

  @Post()
  @ApiBody({ type: CreateDiagnosticDto })
  @ApiConsumes('multipart/form-data')
  uploadModelVersion(
    @User() user: Partial<UserEntity>,
    @Body() data: CreateDiagnosticDto,
  ) {
    const { file, modelIds, ...diagnosticData } = data;

    return this.diagnosticsService.createDiagnostic(
      user,
```

```
      file,
      diagnosticData,
      modelIds,
    );
  }
}

// diagnostic.entity.ts

import { ApiHideProperty, ApiProperty } from '@nestjs/swagger';
import {
  Entity,
  Column,
  ManyToOne,
  OneToOne,
  JoinColumn,
  OneToMany,
  VirtualColumn,
} from 'typeorm';
import { CommonEntity } from 'src/common/entities';
import { UserEntity } from '../users/user.entity';
import { FileEntity } from '../files/file.entity';
import { DiagnosticStatus } from './enums';
import { DiagnosticResultEntity } from '../diagnostic-results/diagnostic-result.entity';
import { getDiagnosticStatusQuery } from './diagnostics.helper';

@Entity({ name: 'diagnostics' })
export class DiagnosticEntity extends CommonEntity {
  @ApiProperty({ type: 'string', maxLength: 128 })
  @Column({ length: 128, nullable: false })
  name: string;

  @ApiProperty({
    type: 'string',
    maxLength: 500,
    required: false,
    nullable: true,
  })
  @Column({ length: 500, nullable: true })
  description: string;

  @ApiProperty({ type: () => UserEntity, required: true, nullable: false })
  @ManyToOne(() => UserEntity, {
    onDelete: 'CASCADE',
    eager: false,
    nullable: false,
  })
  @JoinColumn()
  user: Partial<UserEntity>;

  @JoinColumn()
  @ApiProperty({
    type: () => FileEntity,
    required: true,
    nullable: false,
  })
  @OneToOne(() => FileEntity, {
    onDelete: 'CASCADE',
    nullable: false,
    eager: true,
  })
  image: Partial<FileEntity>;

  @ApiProperty({ enum: DiagnosticStatus })
  @VirtualColumn({
```

```
    type: 'enum',
    query: getDiagnosticStatusQuery,
  })
  status: DiagnosticStatus;

  @ApiHideProperty()
  @OneToMany(() => DiagnosticResultEntity, ({ diagnostic }) => diagnostic, {
    nullable: true,
  })
  results: DiagnosticResultEntity[];
}
```

// diagnostics.helper.ts

```
import { DiagnosticResultStatus } from '../diagnostic-results/enums';
import { DiagnosticStatus } from './enums';

export const getDiagnosticStatusQuery = (alias: string) => `
CASE
  WHEN (SELECT COUNT(*) FROM "diagnostic-results" dr WHERE dr."diagnosticId" = ${alias}.id
AND dr.status = '${DiagnosticResultStatus.FAILED}') > 0 THEN
'${DiagnosticStatus.COMPLETED_WITH_ISSUES}'
  WHEN (SELECT COUNT(*) FROM "diagnostic-results" dr WHERE dr."diagnosticId" = ${alias}.id
AND dr.status != '${DiagnosticResultStatus.COMPLETED}') = 0 THEN
'${DiagnosticStatus.COMPLETED}'
  ELSE '${DiagnosticStatus.PENDING}'
END
`;
```

// diagnostic-results.service.ts

```
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { BaseService } from 'src/common/services';
import { Repository } from 'typeorm';
import { diagnosticResultsServiceErrorMessages } from './diagnostic-results.constants';
import { DiagnosticResultEntity } from './diagnostic-result.entity';

@Injectable()
export class DiagnosticResultsService extends BaseService<DiagnosticResultEntity> {
  constructor(
    @InjectRepository(DiagnosticResultEntity)
    private readonly diagnosticResultEntityRepository: Repository<DiagnosticResultEntity>,
  ) {
    super(
      diagnosticResultEntityRepository,
      diagnosticResultsServiceErrorMessages,
    );
  }
}
```

// diagnostic-result.entity.ts

```
import { ApiProperty } from '@nestjs/swagger';
import { Entity, Column, ManyToOne, JoinColumn } from 'typeorm';
import { CommonEntity } from 'src/common/entities';
import { FileEntity } from '../files/file.entity';
import { DiagnosticResultStatus } from './enums';
import { DiagnosticEntity } from '../diagnostics/diagnostic.entity';
import { DiagnosticModelVersionEntity } from '../diagnostic-model-versions/diagnostic-model-
version.entity';
import { Transform } from 'class-transformer';

@Entity({ name: 'diagnostic-results' })
export class DiagnosticResultEntity extends CommonEntity {
```

```
  @ApiProperty({
    type: () => DiagnosticEntity,
    required: true,
    nullable: false,
  })
  @ManyToOne(() => DiagnosticEntity, {
    onDelete: 'CASCADE',
    eager: false,
    nullable: false,
  })
  @JoinColumn()
  diagnostic: Partial<DiagnosticEntity>;

  @JoinColumn()
  @ApiProperty({
    type: () => FileEntity,
    required: true,
    nullable: false,
  })
  @ManyToOne(() => DiagnosticModelVersionEntity, {
    onDelete: 'CASCADE',
    nullable: false,
    eager: false,
  })
  modelVersion: Partial<DiagnosticModelVersionEntity>;

  @ApiProperty({
    enum: DiagnosticResultStatus,
    default: DiagnosticResultStatus.PENDING,
  })
  @Column({
    type: 'enum',
    enum: DiagnosticResultStatus,
    default: DiagnosticResultStatus.PENDING,
  })
  status: DiagnosticResultStatus;

  @Transform(({ value }) => parseFloat(value))
  @ApiProperty({ type: Number, required: false, example: 23, nullable: true })
  @Column({ type: 'numeric', scale: 2, precision: 5, nullable: true })
  diseaseProbability: number;

  @ApiProperty({
    type: 'string',
    maxLength: 500,
    required: false,
    nullable: true,
  })
  @Column({ length: 500, nullable: true })
  resultDescription: string;
}
```

// main.py

```
import logging
from fastapi import FastAPI
from app.services.rabbitmq_connection_service import RabbitMQConnectionService
from app.services.rabbitmq_consumer_service import RabbitMQConsumerService
from app.services.rabbitmq_publisher_service import RabbitMQPublisherService
from app.services.storage_service import StorageService
from app.services.disease_analyzer_service import DiseaseAnalyzerService

logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger(__name__)
```

```
app = FastAPI()

storage_service = StorageService()
disease_analyzer_service = DiseaseAnalyzerService(storage_service)
rabbitmq_connection_service = RabbitMQConnectionService()
rabbitmq_publisher_service = RabbitMQPublisherService(rabbitmq_connection_service)
rabbitmq_consumer_service = RabbitMQConsumerService(rabbitmq_connection_service,
rabbitmq_publisher_service, storage_service, disease_analyzer_service)
```

// disease_analyzer_service.py

```
import tensorflow as tf
import io
import os
import logging
from app.services.storage_service import StorageService
from dotenv import load_dotenv
import tempfile

load_dotenv()

class DiseaseAnalyzerService:
    def __init__(self, storage_service: StorageService):
        self.logger = logging.getLogger(__name__)
        self.storage_service = storage_service
        self.model = self.load_model(os.getenv('MODEL_PATH'))

    def load_model(self, model_path):
        try:
            self.logger.debug(f"Loading model from: {model_path}")
            buffer = self.storage_service.read_file_to_buffer(model_path)
            with tempfile.NamedTemporaryFile(delete=False) as temp_model_file:
                temp_model_file.write(buffer.getvalue())
                temp_model_path = temp_model_file.name
            model = tf.keras.models.load_model(temp_model_path)
            self.logger.info("Model loaded successfully")
            os.remove(temp_model_path)
            return model
        except Exception as e:
            self.logger.error(f"Failed to load model: {e}")
            raise e

    def analyze_disease(self, image_buffer):
        try:
            image = tf.io.decode_image(image_buffer.getvalue(), channels=3)
            image = tf.image.resize(image, [224, 224])
            image = tf.expand_dims(image, axis=0)
            prediction = self.model.predict(image)
            self.logger.debug(f"PREDICTION: {prediction}")
            disease_probability = prediction[0][0]
            self.logger.debug(f"Disease probability: {disease_probability}")
            return disease_probability
        except Exception as e:
            self.logger.error(f"Failed to analyze disease: {e}")
            raise e
```

// rabbitmq_connection_service.py

```
import os
import pika
from dotenv import load_dotenv
import logging
```

```
load_dotenv()

class RabbitMQConnectionService:
    def __init__(self):
        self.logger = logging.getLogger(__name__)
        self.rabbitmq_host = os.getenv('RABBIT_MQ_HOST')
        self.rabbitmq_port = os.getenv('RABBIT_MQ_PORT')
        self.rabbitmq_user = os.getenv('RABBIT_MQ_DEFAULT_USER')
        self.rabbitmq_password = os.getenv('RABBIT_MQ_DEFAULT_PASS')
        self.connection = None
        self.channel = None


    def set_up_connection(self):
        self.connection = pika.BlockingConnection(
            pika.ConnectionParameters(
                host=self.rabbitmq_host,
                port=self.rabbitmq_port,
                credentials=pika.PlainCredentials(
                    self.rabbitmq_user,
                    self.rabbitmq_password
                )
            )
        )
        self.channel = self.connection.channel()
        self.channel.basic_qos(prefetch_count=1)

    def assert_queue(self, queue: str):
        self.logger.debug(f"Asserting queue - {queue}")
        self.channel.queue_declare(queue=queue, durable=True)

    def get_channel(self):
        return self.channel
```

// rabbitmq_publisher_service.py

```
import json
import logging
import pika
from app.services.rabbitmq_connection_service import RabbitMQConnectionService

class RabbitMQPublisherService:
    def __init__(self, rabbitmq_connection_service: RabbitMQConnectionService):
        self.logger = logging.getLogger(__name__)
        self.rabbitmq_connection_service = rabbitmq_connection_service

    def add_message_to_queue(self, queue: str, message_data: dict):
        channel = self.rabbitmq_connection_service.get_channel()
        self.rabbitmq_connection_service.assert_queue(queue)
        self.logger.debug(f"Sending message to queue: {queue}. Data: {message_data}")
        channel.basic_publish(
            exchange='',
            routing_key=queue,
            body=self.convert_data(message_data),
            properties=pika.BasicProperties(delivery_mode=2)
        )
        print(f"Message sent to queue ({queue}) with task data {message_data}")

    def convert_data(self, data: dict) -> bytes:
        return json.dumps(data).encode('utf-8')
```

// rabbitmq_consumer_service.py

```
import json
import os
import logging
from app.services.rabbitmq_connection_service import RabbitMQConnectionService
from app.services.rabbitmq_publisher_service import RabbitMQPublisherService
from app.services.storage_service import StorageService
from app.services.disease_analyzer_service import DiseaseAnalyzerService

class RabbitMQConsumerService:
    def __init__(self, rabbitmq_connection_service: RabbitMQConnectionService,
                 rabbitmq_publisher_service: RabbitMQPublisherService, storage_service:
StorageService,
                 disease_analyzer_service: DiseaseAnalyzerService):
        self.logger = logging.getLogger(__name__)
        self.rabbitmq_connection_service = rabbitmq_connection_service
        self.rabbitmq_publisher_service = rabbitmq_publisher_service
        self.storage_service = storage_service
        self.disease_analyzer_service = disease_analyzer_service
        self.queue_name = os.getenv('RABBIT_MQ_QUEUE_NAME')
        self.result_queue_name = os.getenv('RABBIT_MQ_RESULT_QUEUE_NAME')
        self.rabbitmq_connection_service.set_up_connection()
        channel = self.rabbitmq_connection_service.get_channel()
        self.set_up_analyze_disease_handler(channel)


    def set_up_analyze_disease_handler(self, channel):
        self.rabbitmq_connection_service.assert_queue(self.queue_name)
        self.logger.debug(f"Starting to consume from queue: {self.queue_name}")
        channel.basic_consume(
            queue=self.queue_name,
            on_message_callback=self.handle_analyze_disease_message,
            auto_ack=False,
        )
        channel.start_consuming()

    def handle_analyze_disease_message(self, ch, method, properties, body):
        self.logger.debug(f"Received message: {body}")
        data = self.get_message_data(body)
        try:
            self.logger.debug(f"Data: {data}")
            self.handle_analyze_disease_result(data)
            ch.basic_ack(delivery_tag=method.delivery_tag)
        except Exception as err:
            self.handle_error(data, ch, method, body, str(err))

    def get_message_data(self, body):
        return json.loads(body)

    def handle_analyze_disease_result(self, data):
        result_id = data.get('resultId')
        image_path_in_storage = data.get('imagePathInStorage')
        if not result_id or not image_path_in_storage:
            raise ValueError("Missing required data in message")
        buffer = self.storage_service.read_file_to_buffer(image_path_in_storage)
        if not buffer:
            raise ValueError(f"File by specified path (${image_path_in_storage}) not
found!")
        diseaseProbability = self.disease_analyzer_service.analyze_disease(buffer)
        result_data = {
            'resultId': result_id,
            'status': 'COMPLETED',
            'diseaseProbability': diseaseProbability,
        }
        self.rabbitmq_publisher_service.add_message_to_queue(self.result_queue_name,
result_data)
```

```
    def handle_error(self, data, ch, method, body, err_message):
        self.logger.debug(f"Finishing validation with FAIL. Message body: {body}. Reason:
{err_message}")
        result_id = data.get('resultId', 'unknown')
        failed_message_data = {
                'resultId': result_id,
                'status': 'FAILED'
            }
        self.rabbitmq_publisher_service.add_message_to_queue(self.result_queue_name,
failed_message_data)
        ch.basic_ack(delivery_tag=method.delivery_tag)
```