Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)
Кафедра Інформатики
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

РОЗРОБКА ВЕБЗАСТОСУНКУ ДЛЯ ВЕТЕРЕНАРНОЇ КЛІНІКИ
(тема)

Виконав:
студент 4 курсу, групи ІТІНФ-20-3

Лиман І.С.
(прізвище, ініціали)

Спеціальності 122 Комп'ютерні науки
(код і повна назва спеціальності)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

Керівник проф. Безсонов О.О.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ Кобилін О.А.
(підпис) (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)

Кафедра Інформатики
(повна назва)

Рівень вищої освіти перший (бакалаврський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Лиману Ігорю Сергійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка вебзастосунку для ветеринарної клініки

затверджена наказом університету від 20 травня 2024 року № 464 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 28 травня 2024 р.

3. Вихідні дані до роботи науково-методична та науково-технічна література, дані інтернет-мережі, оточення розробки Visual Studio, інструменти для контейнеризації Docker.

4. Перелік питань, що потрібно опрацювати в роботі
1. Аналіз предметної області.
2. Аналіз мікросервісної архітектури.
3. Проєктування системи.
4. Написання вебзастосунку.
5. Тестування системи.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) Актуальність проблеми мікросервісної, постановка задачі, схема бази даних, архітектури, алгоритм створення JWT, діаграми класів, тестові зображення.

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	08.04.2024	
2	Аналіз завдання, підбір літератури	08.05.24-16.04.24	
3	Аналіз літератури з досліджуваної проблеми	16.04.24-18.04.24	
4	Аналіз предметної області	19.04.24-22.04.24	
5	Аналіз активностей системи	23.04.24-1.05.24	
6	Програмна реалізація	1.05.24-22.05.24	
7	Оформлення пояснювальної записки	23.05.24-26.05.24	
8	Перевірка на плагіат	28.05.24	
9	Рецензування	29.05.24	
10	Підготовка презентації та доповіді	30.05.24-02.06.24	
11	Занесення роботи в електронний архів	06.06.24	
12	Попередній захист кваліфікаційної роботи	06.06.24	

Дата видачі завдання 8 квітня 2024 р.

Студент _____
(підпис)

Керівник роботи _____ проф. Безсонов О.О.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ/ABSTRACT

Пояснювальна записка до кваліфікаційної роботи: 69 с., 10 табл., 43 рис., 1 дод., 32 джерела.

ВЕБЗАСТОСУНОК, МІКРОСЕРВІСНА АРХІТЕКТУРА, .NET, ASP.NET, REACT, ТОКЕНИ, БЕКЕНД, ФРОНТЕНД, DOCKER.

Об’єктом роботи є система для ветеринарної клініки, яка буде написана мною до кінця дослідження.

Метою роботи є розробка вебзастосунку для ветеринарної клініки з використанням мікросервісної архітектури.

Використано мікросервісну архітектури для дефрагментації системи на невеликі компоненти. Створена база даних. Створена система ідентифікації з використанням JWT. Було створено 5 мікросервісів для бекенду використовуючи ASP.NET і один для фронтенду використовуючи React. Система розгорнута у контейнерному середовищі Docker і протестована.

У результаті роботи здійснена програмна реалізація системи для ветеринарної клініки.

WEB APPLICATION, MICROSERVICE ARCHITECTURE, .NET, ASP.NET, REACT, TOKENS, BACKEND, FRONTEND, DOCKER.

The object of the work is a system for a veterinary clinic, which will be written by me by the end of the research.

The purpose of the work is to develop a web application for a veterinary clinic using microservice architecture.

Microservice architecture is used to defragment the system into small components. The database is created. An identification system using JWT has been created. 5 microservices were created for the backend using ASP.NET and one for the frontend using React. The system is deployed in a Docker container environment and tested.

As a result of the work, the software implementation of the system for the veterinary clinic was carried out.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

API – Application Programming Interface (інтерфейс програмування додатків)

СУБД – Системи керування базами даних

ORM – Object-Relational Mapping (об’єктно-реляційне відображення)

ACID – Atomicity, Consistency, Isolation, Durability (атомарність, узгодженість, ізоляція, надійність)

LINQ – Language Integrated Query (мова інтегрованих запитів)

SQL – Structured Query Language (мова структурованих запитів)

EF – Entity Framework

HTTP – HyperText Transfer Protocol (протокол передачі гіпертексту)

DI – Dependency Injection (введення залежностей)

JWT – JSON Web Token (вебтокен JSON)

JSON – JavaScript Object Notation (нотація об’єктів JavaScript)

UML – Unified Modeling Language (уніфікована мова моделювання)

UTC – Coordinated Universal Time (всесвітній координований час)

CORS – Cross-Origin Resource Sharing

ВСТУП

Зважаючи на важливість ефективного управління ветеринарними клініками та постійне прагнення до вдосконалення медичних сервісів для тварин, вебзастосунки стають ключовим інструментом для поліпшення якості надання послуг. Швидкий розвиток технологій і інструментів для створення ефективних та функціональних вебзастосунків вимагають від розробника користуватися використовувати архітектуру, яка надає можливість розширювати і вдосконалювати систему за потребами клієнтів системи [1]. Тому в роботі будуть використані сучасні технології, такі як .NET, ASP.NET, React та Docker, що дозволить декомпозувати систему на мікросервіси і працювати з ними незалежно [2].

Ця робота пропонує розв’язання проблем створення системи аутентифікації і авторизації що використовує технологію JWT, створення різних гнучких та надійних мікросервісів для бекенду, які будуть забезпечені роботою з базою даних, кешем, і використовувати сервісно-орієнтований підхід до розробки [3, 4]. Створення фронтенду, який буде клієнтом сервісів бекенду, і буде використовувати компоненти для забезпечення адаптивності вебсторінки. Розгортання сисеми в контейнерному середовищі Docker для забезпечення масштабованості та гнучкості.

Актуальність роботи полягає в використанні мікросервісної архітектури для декомпозиції системи що дозволяє створити гнучку та розширювану систему, яка може легко адаптуватися до змін у вимогах клієнтів та технологічному середовищі.

1 ОГЛЯД МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

1.1 Вступ до мікросервісної архітектури

Через постійне збільшення охоплення інтернету як інструменту для вирішення повсякденних і робочих задач, неминуче для кожної такої задачі створюються вебзастосунки [5, 6]. Користувачі обирають ті вебзастосунки, які мають привабливий і простий інтерфейс, а також мають зручну функціональність. Під час розробки сайту для компанії, яка займається продажем товарів або послуг, потрібно пам’ятати що вебзастосунок потрібен не тільки для полегшення взаємодії клієнта і компаній, а й для того, щоб завдяки гало-ефекту компанії було легше продати свої послуги клієнту. Тому швидкість розгортання, швидкість оновлення та потенціал для масштабування вебзастосунків стає важливою вимогою для компаній, які надають послуги через інтернет. Огляд багатьох вебзастосунків для ветеринарних клінік показав, що більшість з них використовують односторінкові сайти, які мають обмежений потенціал для масштабування та розширення функціональності.

В контексті таких вимог, мікросервісна архітектура стає одним із найпопулярніших підходів для великих проєктів. Вибір цієї архітектури, дозволяє розділяти програму на невеликі, незалежні компоненти, відомі як мікросервіси (рис. 1.1 [7]). Така декомпозиція великого проєкту, сприяє спрощенню процесу розгортання, робить легшим процес масштабування, оскільки кожен мікросервіс може розвиватися і масштабуватися незалежно від інших. Цей архітектурний підхід дозволяє застосунку швидко адаптуватися до змін в вимовах користувачів і швидше збільшувати обсяг користувацької бази, або давати змогу іншим розробникам розробляти додатки до системи [8]. Також розробка, яка використовує цей підхід, має переваги в забезпеченні більшої гнучкості та швидкості в процесі розробки. Кожен мікросервіс можна розробляти, не вдаючись в реалізацію інших, що

дозволяє розробнику концентруватися на вирішенні конкретної задачі, не розмірковуючи над взаємодією його рішення з іншим кодом.

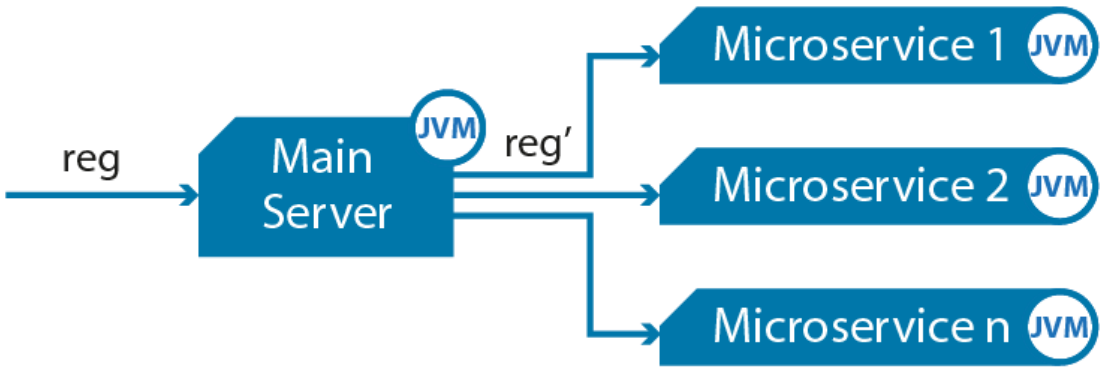


Рисунок 1.1 – Схематичне зображення мікросервісної архітектури

Крім того, мікросервісна архітектура спрощує написання юніт-тестів. Кожин шар мікросервісів може бути протестований незалежно, без необхідності включати в тестування інші частини системи. Це забезпечує високу якість і надійність коду та дозволяє швидко виявляти та виправляти помилки, які з’явились під час розробки або масштабування проєкту. Такий підхід допомагає забезпечити стабільність та надійність системи в цілому.

1.2 Основні принципи мікросервісної архітектури

Впровадження мікросервісної архітектури в систему вимагає розуміння її основних принципів і концепцій. Тому потрібно розглянути ключові аспекти цієї парадигми та їх значення для розробки вебзастосунку для ветеринарної клініки.

Перший і найважливіший аспект мікросервісної архітектури полягає в декомпозиції великої системи на маленькі, самодостатні компоненти. Кожен такий компонент відповідає за конкретну, окрему функціональність або бізнес-логіку. Наприклад, в предметній області «Ветеринарна клініка», один мікросервіс може відповідати за управляння клієнтами, інший – за реєстрацію записів на консультацію, а ще інший за аунтифікацію всіх користувачів. Такий підхід дозволяє різним розробникам системи працювати над окремими мікросервісами паралельно, що значно підвищує швидкість та легкість розробки та забезпечує більшу гнучкість у виборі технологій та інструментів для кожного сервісу. Крім того це збільшує потенціал для оновлення та масштабування, оскільки такі зміни не будуть впливати на решту системи. Наприклад, до системи для ветеринарної клініки можна додати мікросервіс, який буде використовувати нейронні мережі для швидкого аналізу досліджень [9].

Другим ключовим аспектом мікросервісної архітектури є незалежність сервісів. Кожен мікросервіс має бути самодостатнім та повинен не залежати від внутрішньої реалізації інших сервісів в системі. Це означає, що кожен сервіс має власну функціональність власну базу даних, власні залежності, власну взаємодію з компонентами, які знаходяться за функціональністю мікросервісів, такі як кешування або інші API. Незалежність сервісів забезпечує гнучкість та резильєнтність системи. Це означає, що при виявлені проблем з одним з мікросервісом, його нестійкість не буде впливати на решту системи. Кожен з сервісів, може бути розвинений, розгорнутий, масштабований або оновлений незалежно від інших, що забезпечує швидке впровадження змін та дозволяє якісно підтримувати систему в цілому. Забезпечення незалежності сервісів також допомагає розділити відповідальності між різними командами розробників. Кожна команда може відповідати за свій власний сервіс та опрацьовувати його функціональність не залежно від інших команд, що сприяє збільшенню продуктивності та швидкості розробки.

У мікросервісній архітектурі, сервіси взаємодіють між собою через добре визначенні API. Це дозволяє окремим компонентам спілкуватися та обмінюватися даними безпосередньо, незалежно від того, якою мовою програмування або технологією вони користуються. API дає стандартизований спосіб взаємодії між сервісами, що полегшує та пришвидшує розробку, тестування та підтримку систем. Це також дозволяє забезпечити гнучкість та незалежність від реалізації інших сервісів. Наприклад, якщо один сервіс потребує змін у своїй функціональності, це буде зроблено без впливу на інші сервіси, якщо API залишається незмінним. Саме API дозволяє декомпозувати функціональність системи на незалежні компоненти, завдяки чому спрощується розробка.

Наступним аспектом є автоматизація розгортання і масштабування мікросервісів. Ця автоматизація дозволяє швидко та ефективно впроваджувати нові версії сервісів та дозволяє забезпечити консистентність середовища розгортання. Це досягається за допомогою інструментів автоматизації, таких як Docker, Kubernetes, або інших технологій контейнерного розгортання, які дозволяють додавати, інтегрувати, змінювати або масштабувати сервіси у віртуальному середовищі, без великих зусиль розробника. Загалом, автоматизація розгортання та масштабування забезпечує швидке впровадження оновлень та ефективне управління інфраструктурою, що є ключовими для успішної реалізації мікросервісної архітектури.

1.3 Переваги і недоліки мікросервісної архітектури

Під час написання системи, вибір архітектурного підходу, та впровадження цього підходу в проєкт, потрібно розуміти його переваги, недоліків та обмеження. Через це, потрібно розглянути переваги і недоліки використання мікросервісної архітектури в системі для ветеринарної клініки.

Перевагою мікросервісної архітектури в контексті системи для ветеринарної клініки є здатність до швидкого розвитку та гнучкість в розробці. Наприклад, уявімо сервіс для управління медичними записами тварин. Якщо з'являється необхідність додати новий функціонал, наприклад, можливість завантажувати та переглядати результати лабораторних аналізів, розробник може працювати над цим сервісом незалежно від інших частин системи. Це дозволяє прискорити розробку та впровадження нового функціоналу, не чекаючи на зміни в інших компонентах системи.

Також, якщо виникає потреба у оновлені та поліпшення існуючого функціоналу, наприклад, впровадження систем онлайн-запису на прийом до ветеринара, розробник має можливість працювати над цією проблемою окремо від інших сервісів. Це дає переваги коли потрібно оперативно реагувати на потреби клієнтів та швидко впроваджувати зміни в систему без перешкод, що можуть виникнути у більших монолітних системах (рис. 1.2 [10]).

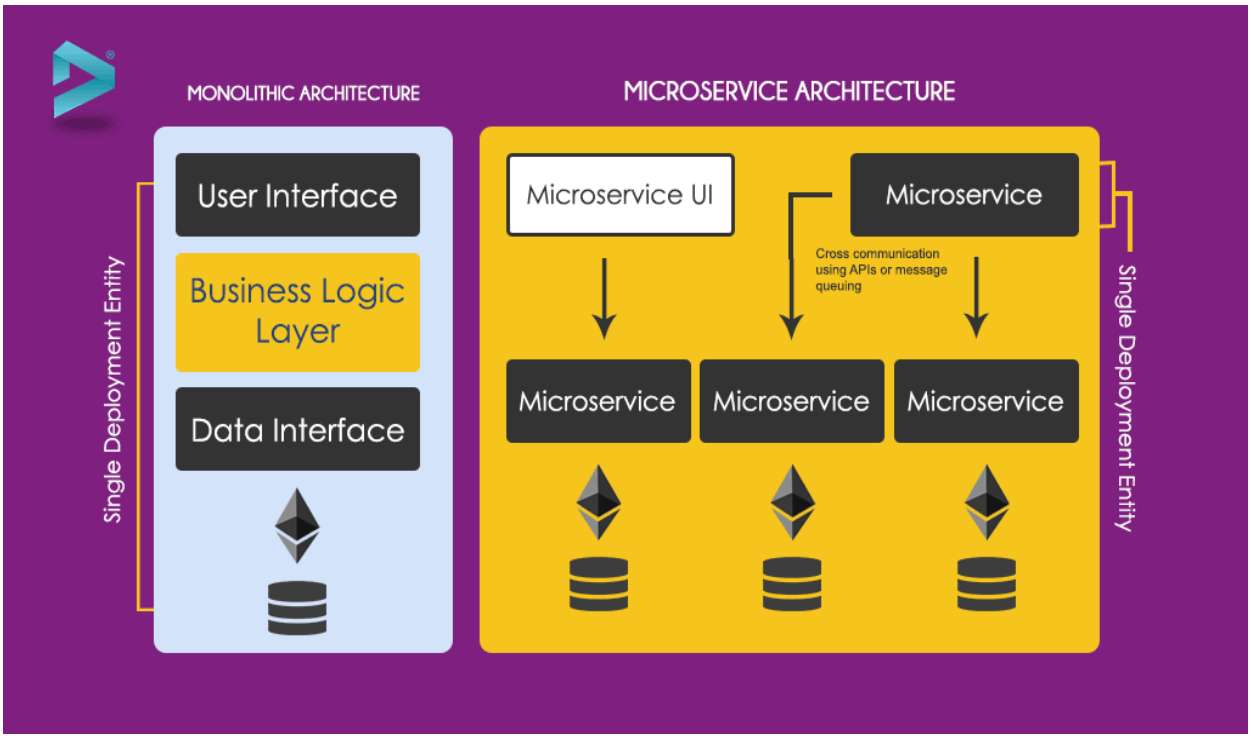


Рисунок 1.2 – Схематичне порівнянню мікросервісної архітектури і монолітної архітектури

Використання мікросервісної архітектури в системі для ветеринарної клініки сприяє її відокремленості та незалежності компонентів. Через те, що кожен мікросервіс відповідає за окрему функціональність, стійкість системи значно підвищується. Кожен сервіс може бути розгорнутий окремо, що знижує вплив можливих проблем на інші частини системи. Наприклад, якщо один сервіс перестає працювати через помилку або навантаження, це не призведе до збою всієї системи. Відповідно, інші сервіси продовжать працювати нормально, забезпечуючи користувачам неперервну роботу.

Більше того, мікросервісна архітектура сприяє гнучкості в управлінні ресурсами та відновленні після збоїв. Наприклад, якщо один сервіс перевантажений або недоступний, можна легко масштабувати або відновити лише цей сервіс без впливу на решту системи. Це дозволяє швидко відновлювати роботу системи після неполадок та забезпечує стабільну та безперервну роботу для користувачів.

Мікросервісна архітектура сприяє технологічному різноманіттю в системі для ветеринарної клініки, оскільки кожен сервіс може бути написаний з використанням різних технологій. Наприклад один сервіс може використовувати бази даних, а інший ні. Різні мікросервіси можуть бути написані на різних мовах програмування, або мати одну мову програмування, але різні її версії. Як приклад, при розробці системи для ветеринарної аптеки, можна передбачити, що мікросервіс який відповідає за аунтифікацію може буте використаний в подальшому для ветеринарної аптеки, або інших системах пов’язаних з попередньою. Якщо проєкт буде розроблятися з використанням монолітної архітектури, то авторизацію або неможливо буде використовувати для цих цілей, або потрібно буде робити великий проєкт, який уособлював в собі і клініку і аптеку.

Також мікросервісна архітектура вирішує проблеми з навантаженням на систему, завдяки тому, що вона дозволяє масштабувати ті сервіси, на які робиться більше запитів.

Якщо говорити про недоліки мікросервісної архітектури, то першим з них буде складність управління. Це не дуже суттєво коли мова йде про декілька компонентів, але зі збільшенням системи можуть виникнути проблеми з моніторингом, керуванням версій, залежностями та розгалуженням коду. Тому на відмінно від розробки з використанням монолітної архітектури, де ніяк не може бути проблем з різними версіями бібліотек, використання монолітної архітектури зобов’язує дивитися на це більш пильно.

Хоча написання юніт тестів в мікросервісній архітектурі і є простішим за написання тих тестів в монолітній архітектурі, їх кількість буде значно вища. Також через розподіленість системи та велику кількість залежностей, може бути проблема з написанням інтегрованих тестів. Але треба зауважити, що надійність мікросервісній архітектурі буде все одно вищою за інші.

Мережеві витрати будуть значно вищими, через те що різні компоненти будуть спілкуватися між собою. Це може стати проблемою в разі частих запитів сервісів між собою. Частково вирішується завдяки кешуванням статичних ресурсах в проксі-серверах таких як Nginx, але треба бути уважним щоб уникнути проблем з кешуванням застарілих даних.

Таким чином хоча мікросервісна архітектура і має проблеми зі складністю керування і тестуванням, вона все ще є кращим варіантом через її гнучкість і легкість під час розробки.

1.4 Вибір технологій для розробки

Наступним етапом в процесі створення системи для ветеринарної клініки з використанням мікросервісної архітектури буде вибір оптимальних технологій розробки. Цей вибір буде спрямований на забезпечення надійності, ефективності та зручності управління системою для користувачів та розробників.

В якості контейнерної технології мною був обраний Docker. Docker – це платформа для автоматизації розгортання та управління програмним забезпеченням в середовищі з підтримкою контейнеризації. Завдяки Docker кожний окремий сервіс, з підтримкою всіх його залежносте, може бути упакований в окремий контейнер. Цей контейнер можна розгорнути в будь якій системі з підтримкою Linux і контрольних груп в ядрах. Docker також надає можливість управляти цими контейнерами за допомогою набору команд, забезпечуючи повну ізоляцію контейнерів на рівні файлової системи, процесорів і мережі.

В якості СУБД було обрано PostgreSQL. Ця СУБД буде раціональним вибором при написанні системи яка використовує мікросервісну архітектуру. PostgreSQL легко інтегрується з контейнерною технологією Docker. Її контейнер швидко створюється, розгортається і керується в віртуальних середовищах, що спрощує розгортання і керування усім проєктом. Також дозволяється легке створення окремих екземплярів бази даних для різних мікросервісів. PostgreSQL відома своєю стабільністю і високою надійністю у роботі. Вона підтримує ACID, що робить її ідеальним вибором для сховища даних у мікросервісній архітектурі, де надійність та стабільність дуже важливі. PostgreSQL налічує велику кількість функціональних рішень, які дозволяють легко розширювати систему і сприяють її більшій гнучкості при розробці системи. Вона може підтримувати багато типів даних, складені запити та індексацію. Отже вибір PostgreSQL для мікросервісної архітектури з використанням з Docker дає системі потужний та надійний інструмент для зберігання та керування даними, що зробить саму систему більш гнучкою та надійною.

В якості платформа для написання backend для ветеринарної клініки мною була вибрана модульна платформа .NET, а в якості мови програмування C#. Платформа .NET дає широкий асортимент бібліотек та інструментів для розробки, включаючи створення вебзастосунків, використання СУБД, написання системи з використанням мікросервісної

архітектури. Платформа .NET відома своєю здатністю працювати з великими обсягами даних та обробляти велику кількість запитів швидко та ефективно. Це робить її ідеальним вибором для створення вебзастосунків, які мають обробляти велику кількість інформації, як це може бути у ветеринарній клініці. Завдяки кросплатформенності .NET систему легко розробляти на Windows або macOS, а разгортати на Linux контейнері через Docker .NET забезпечує високий рівень безпеки та надійності, що є критичним для систем, які працюють з конфіденційними медичними даними. Інструменти та практики .NET допомагають уникати потенційних загроз безпеці та забезпечують стабільну роботу застосунку.

C# це об’єктно орієнтований язик програмування який працює на платформі .NET зі строгою, статичною типізацією. Синтаксис C# схожий на синтаксис C++ або Java, але має більше синтаксичного цукру, такого як властивості або LINQ. Ця мова підтримує поліморфізм, який реалізований через переважання методів, оператори, делегати, події, атрибути, винятки, асинхронність і паралельність, узагальнення. Має Garbage Collection, що значно спрощує керування внутрішньої пам’яті при створені екземпляра класу. Для зовнішньої існує дві механізми, фіналізатор, який автоматично викликається при видаленні об’єкта, і Dispose який може бути викликаний власноруч, якщо розробник впевнений, що екземпляр класу більше не знадобиться. При роботі з колекціями можна використовувати технології LINQ, що значно прискорює написання системи і робить код більш лаконічним і читабельним [11, 12]. Ця технологія або надає синтаксис у вигляді методів розширення, або додає в C# мову запросів що використовує декларативний синтаксис запиту [13]. Альтернативою C# може служити мова програмування Java, яка теж широко використовується для розробки вебзастосунків.

В якості ORM було обрано Entity Framework. Ця бібліотека дає потужний інструмент для роботи з базами даних і полегшує розробку. EF спрощує запити до самої бази через використання високорівневого способу

роботи з об’єкти через об’єктно орієнтовану модель, замість написання SQL запитів і їх подальшому маппінгу. Крім того EF надає велику кількість інструментів які спрощують роботу з базою даних, такі як автоматичне створення та оновлення бази даних, маппінг об’єктів на таблиці та відносини між ними, можливість конвертувати LINQ запроси у SQL. Тож вибір EF є оптимальним при написанні систем на платформі .NET.

ASP.NET був мною вибраний для написання усіх backend мікросервісів. Він являє собою платформу для розробки вебзастосунків і буде гарним вибором для написання web API. ASP.NET надає широкий спектр можливостей для створення надійних, масштабованих та ефективних вебзастосунків [14]. Він підтримує різні протоколи, такі як HTTP, WebSocket, інтегрується з базами даних, що зробить інтеграцію EF і PostgreSQL легкою і зручною. Має вбудовану систему DI, що робить легким інтеграцію шаблонів програмування IServices та IRepositories. ASP.NET дає можливість використання мідлвер для обробки HTTP-запитів та відповідей, дозволяючи вбудовувати різноманітні операції, такі як аутентифікація, авторизація та логування, у потік обробки запитів. Ця платформа через її потужність і надійність буде найкращим варіантом для написання backend для системи для ветеринарної клініки.

При написання користувацького інтерфейсу були використані мови HTML, CSS для розмітки, і Typescript для логіки. В якості фреймворку для Typescript мною був обраний React. Їх інтеграція з сервером на ASP.NET дозволяє забезпечити надійність через однакову типізацію на рівні клієнта і сервера. Розробка з використанням React значно пришвидшує розробку і робить сторінку більш адаптивною. Через використання react-dom можна розробляти компоненти окремо, і завантажувати їх клієнту тільки за потреби, що робить вебсторінку більш швидкою. Таким чином використання технологій HTML, CSS, Typescript, React дозволить написати користувацький інтерфейс швидко, надійно і з гарною інтеграцією ASP.NET сервера [15, 16].

Альтернативою React може служити Angular, який теж використовує компоненти для рендерінгу сторінки [17].

1.5 Постановка задачі

Підбиваючи підсумки, метою дослідження буде створення системи для ветеринарної клініки з використанням мікросервісної архітектури. Метою створення цієї системи буде демонстрація можливостей використання мікросервісної архітектури і демонстрація можливості спрощення доступу до медичних послуг для клієнтів ветеринарної клініки. Це може буте корисним не тільки при розробці систем зі схожою предметною областю, а і для будь яких вебзастосунків, які можуть використовувати мікросервісної архітектури.

Об’єктом роботи є система для ветеринарної клініки, яка буде написана мною до кінця дослідження.

Метою роботи є розробка вебзастосунку для ветеринарної клініки з використанням мікросервісної архітектури.

Для досягнення мети в написанні системи потрібно буде:

– описати бізнес логіку використовуючи PostgreSQL і зробити маппінг даних з EF, або використати code First, і зробити міграції до PostgreSQL;

– розробити написати IRepository і IServices для роботи з предметною областю;

– розробити мікросервіси з використанням технології ASP.NET;

– написати користувацький інтерфейс з використанням технологій HTML, CSS, Typescript, React;

– розгорнути проєкт в Docker.

2 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

2.1 Вступ до предметної області

Першим етапом у створенні вебзастосунку для ветеринарної клініки з використанням мікросервісної архітектури буде аналіз предметної області, декомпозиція системи на менші компоненти і опис цих компонентів. Для цього до початку розробки потрібно чітко уявляти як буде виглядати система з модельної і архітектурної точки зору, і як користувач буде взаємодіяти з цією предметною областю. Для цього було виділено 5 компонентів, які будуть описувати внутрішню логіку системи: сервіс для авторизації і аунтифікації, сервіс зі списком послуг, сервіс для профіля ветеринарів, сервіс для профіля користувачів і їх хатніх тварин, сервіс для керування записами на прийом.

Сервіс який описує логіку авторизації і аунтифікації є ключовим для системи через те, що без його використання складно буде забезпечити безпеку системи і зробити легку ідентифікацію користувачів системи. Його основними функціями буде зберігання акаунтів користувачів системи, їх пошти і ролі, і створення нових користувачів. Дані з цього компонента будуть передаватися завдяки JWT, який буде зберігатися у клієнта і завдяки інорполяції передаватися на інші компоненти, що забезпечить безпечну авторизацію і аунтифікацію у системі [18]. У вебзастосунку буде 3 ролі: користувач, лікар, адмін. Користувач має доступ лише до предметної області, яка пов’язана з ним. Він, зазвичай, виступає як господар домашньої тварини, хоча господарями може бути також і лікар і адмін. Користувач може змінювати свій профіль, реєструвати, оновлювати і видаляти своїх хатніх тварин. Лікар теж має доступ лише до своєї частини системи, це свій профіль, розклад, і за потреби він може реєструвати тварину так-же як і користувач. Адмін має необмежений доступ до системи, якщо це не суперечить бізнес логікі і безпеки.

Наступним є сервіс який надає доступ до каталогу списку послуг і каталогу можливих тварин, з якими може працювати ветеринари в рамках наданого каталогу послуг. Основне призначення цього компоненту – продемонструвати клієнту набір можливих послуг, які можуть бути надані у цій ветеринарної клініці. Доступ до цього каталогу і каталогу видів тварин, з якими може працювати ветеринарний лікар в рамках кожної послуги, може отримати будь хто, незалежно яку роль він має, і чи аунтифікований він чи ні. Додавати, редагувати і видаляти послуги і види тварин може лише адмін.

Сервіс для профілю ветеринарного лікарю дозволяє лікарю надавати інформацію про себе, редагувати її. Також цей сервіс відповідає за розклад ветеринара. Завдяки цьому сервісу лікар може описати себе, надати корисну інформацію щоб користувач міг вибрати потрібного йому лікаря під час запису. Будь хто може подивитися цю інформацію, незалежно від його ролі і стану авторизації. Лікар може редагувати лише свій профіль. Адмін може редагувати профіль будь якого лікаря. Доступ до розкладу надається таким же чином, це дозволяє бачити коли можна записатися на послуги і до якого лікаря, і забезпечує високий рівень безпеки.

Сервіс для профілю користувачів дозволяє будь якому аунтифікованому користувачу незалежно від ролі створити власний акаунт і додати своїх хатніх тварин. Вид хатньої тварини обмежений списком з сервісу для списку послуг. Користувач чи лікар може додавати, редагувати або видаляти лише інформацію що пов’язана з його профілем. Адмін має доступ до будь якого профілю. Використання цього сервісу опційно, через можливість записуватися на прийом не аунтифікованих користувачів, але його використання дозволяє покращити роботу лікаря через можливість довитися додаткову інформацію і історію лікування.

Основним сервісом який надає основну функціональність цієї системи - це сервіс для керування записами на прийом. Записуватися на прийом може будь хто, в залежності від того чи зареєстрований користувач чи ні, його пошта береться або з токіну або користувач надає її сам. Першим чином

користувач повинен вибрати послугу, яку надає сервіс зі списком послуг. Після вибору послугу користувачу надається вибір тварини. Цей вибір обмежений тими тваринами, які зберігаються в вибраної послуги. Після вибору тварини, можна опційно вибрати породу хатньої тварини. У випадку якщо користувач аунтифікований і має зареєстрованих тварин, він може вибрати свою тварину, і її дані підтягнуться автоматично. Далі, незалежно від попередніх виборів, користувачу надається можливість вибрати ветеринарного лікаря, який буде проводити обрану послугу. Після, система буде аналізувати його розклат і зайнятість і надає списом можливих днів, в які користувач може отримати послуги. Після, користувач може обрати час, який аналізований таким же чином. За потреби користувач може надати додаткову текстову інформацію. Сам користувач може редагувати або видалити цей запит. Лікар може прийняти або скасувати запит. Адмін має доступ до кожного запиту.

Таким чином логіка системи буде поділена на 5 компонентів, що дає змогу розробляти систему незалежно, надає високий рівень безпеки, пришвидшує розробку і робить програму більш адаптивною.

2.2 Опис предметної області

Наступним етапом розробки вебзастосунку для ветеринарної клініки буде виділення моделей, які будуть повністю описувати систему [19]. Це дозволить правильно уявляти ті сутності які потім будуть використовуватися в базі даних і в коді, полегшить розробку і дозволить побачити можливі проблеми до початку розробки, якщо сутності будуть не повністю описувати предметну область.

Сервіс для аунтифікації і авторизації користувачів буде описаний завдяки двох сутностей: IdentityUser і IdentityRole та таблиці для зв’язку між ними IdentityUserRole (табл. 2.1, 2.2). Через використання платформи

розробки ASP.NET можна не писати власну реалізацію сутностей а скористуватися вже даними сутностями і сервісами для ідентифікації. Ця реалізація повністю задовольняє цілям системи а саме створення акаунтів, зберігання їхньої пошти, використання 3 ролей: користувач, лікар, адмін і надання користувачам цих ролей.

Таблиця 2.1 – Таблиця для IdentityUser

Поле	**Тип даних**	**Опис**
1	**2**	**3**
Id	Рядок	Унікальний ідентифікатор користувача
UserName	Рядок	Ім’я користувача
NormalizedUserName	Рядок	Ім’я користувача в нормалізованому форматі
Email	Рядок	Електронна пошта користувача
NormalizedEmail	Рядок	Електронна пошта користувача в нормалізованому форматі
EmailConfirmed	Булево	Підтвердження електронної пошти користувача
PasswordHash	Рядок	Хеш паролю користувача
SecurityStamp	Рядок	Унікальний ідентифікатор безпеки користувача
ConcurrencyStamp	Рядок	Унікальний ідентифікатор конкурентності користувача
PhoneNumber	Рядок	Номер телефону користувача
PhoneNumberConfirmed	Булево	Підтвердження номера телефону користувача

Продовження таблиці 2.1

1	2	3
TwoFactorEnabled	Булево	Включення двофакторної аутентифікації для користувача
LockoutEnd	Дата	Час закінчення блокування облікового запису користувача
LockoutEnabled	Булево	Включення блокування облікового запису користувача
AccessFailedCount	Ціле число	Кількість невдалих спроб входу в систему

Таблиця 2.2 – Таблиця для IdentityRole

Поле	**Тип даних**	**Опис**
Id	Рядок	Унікальний ідентифікатор ролі
Name	Рядок	Назва ролі
NormalizedName	Рядок	Назва ролі в нормалізованому форматі
ConcurrencyStamp	рядок	Унікальний ідентифікатор конкурентності ролі

Сервіс для каталогу послуг описується завдяки сутностям VetAid і AnimalType і проміжною сутністю VetAidAnimalType (табл. 2.3-2.5). Сутність AnimalType дозволяє зберігати можливі типи хатніх тварин, з якими можуть працювати ветеринарні лікарі, і дозволяє робити послуги як для всіх тварин, так і лише для окремих. Сутність VetAid являє собою послугу, і зберігає орієнтовану тривалість, ціну, тип послуги та її опис. Це дозволить користувачам зразу бачити ці параметри. Через зв’язок багато до багатьох між двома таблицями, створено додаткову таблицю VetAidAnimalType.

Таблиця 2.3 – Таблиця для VetAid

Поле	**Тип даних**	**Опис**
Id	Ціле число	Унікальний ідентифікатор послуги
Name	Рядок	Назва послуги
Description	Рядок	Опис послуги
ServiceType	Рядок	Тип послуги («консультація», «діагностика», «лікування»)
Duration	Час	Тривалість послуги (відведений час)
Price	Десяткове	Ціна послуги

Таблиця 2.4 – Таблиця для AnimalType

Поле	**Тип даних**	**Опис**
Id	Ціле число	Унікальний ідентифікатор типу тварини
Name	Рядок	Назва типу

Таблиця 2.5 – Таблиця для VetAidAnimalType

Поле	**Тип даних**	**Опис**
VetAidId	Ціле число	Унікальний ідентифікатор послуги
TypeId	Ціле число	Унікальний ідентифікатор типу тварини

Сервіс для профілю ветеринарних лікарів описується завдяки сутностям VetProfile і VetAvailability (табл. 2.6, 2.7). Сутність VetProfile описує ту інформацію про ветеринарного лікаря, котру потрібно бачити користувачам під час прийняття рішення про вибір лікаря. Завдяки

електронної пошти ветеренара, сутність можна зв’язати з сутністю IdentityUser. Для описання часів роботи лікаря і його розкладу буде використана сутність VetAvailability. У випадку коли часи роботи у лікаря роздроблені, можна створити декілька екземплярів сутності з однією датою. Зв’язок один до багатьох, тому допоміжна сутність не потрібна.

Таблиця 2.6 – Таблиця для VetProfile

Поле	**Тип даних**	**Опис**
Id	Ціле число	Унікальний ідентифікатор ветеренара
FullName	Рядок	Повне ім’я ветеринара
Email	Рядок	Електронна пошта ветеринара
Specialization	Рядок	Спеціалізація ветеринара

Таблиця 2.7 – Таблиця для VetAvailability

Поле	**Тип даних**	**Опис**
Id	Ціле число	Унікальний ідентифікатор доступності
Date	Дата	Дата доступності
StartTime	Час	Початковий час доступності
EndTime	Час	Кінцевий час доступності
ProfileId	Ціле число	Ідентифікатор профілю ветеринара

Сервіс для профілю користувачів описується завдяки сутностям OwnerProfile і Pet (табл. 2.8, 2.9). Користувач може зареєструвати свою тварину завдяки сутності Pet, і додати корисну інформацію про неї. Зв’язок один до багатьох, тому допоміжна сутність не потрібна.

Таблиця 2.8 – Таблиця для OwnerProfile

Поле	**Тип даних**	**Опис**
Id	Ціле число	Унікальний ідентифікатор
FullName	Рядок	Повне ім’я власника
Email	Рядок	Електронна пошта власника

Таблиця 2.9 – Таблиця для Pet

Поле	**Тип даних**	**Опис**
Id	Ціле число	Унікальний ідентифікатор тварини
Name	Рядок	Ім’я тварини
Age	Ціле число	Вік тварини
AnimalType	Рядок	Тип тварини (наприклад, собака, кіт тощо)
Breed	Рядок	Порода тварини
OwnerProfileId	Ціле число	Ідентифікатор профілю власника

Останній сервіс для керування запису на прийом описується лише сутністю Booking (табл. 2.10). Користувач може вибрати послугу, потім або вид тварини або його зареєстровану тварину (тоді вид і порода тварини підтягується автоматично). Після можна вибрати ветеринарного лікаря, дату зустрічі і її час. Лікар може підтвердити запит або відмовити його. Ідентифікатор зареєстрованого користувача або тимчасовий ідентифікатор незареєстрованого користувача і його пошта зберігаються автоматично. Доктор може змінити статус прийому підтвердив або скасувавши його. Користувач теж може скасувати запит.

Таблиця 2.10 – Таблиця для Booking

Поле	**Тип даних**	**Опис**
Id	Ціле число	Унікальний ідентифікатор бронювання
AppointmentDateTime	Дата і час	Дата та час запису на прийом
ClientEmail	Рядок	Електронна пошта клієнта
ClientId	Ціле число	Унікальний ідентифікатор клієнта
DoctorId	Ціле число	Ідентифікатор лікаря
PetId	Ціле число	Ідентифікатор домашнього улюбленця (опційно)
AnimalTypeId	Ціле число	Ідентифікатор типу тварини
PetBreed	Рядок	Порода тварини
AppointmentStatus	Перерахування	Статус запису на прийом
BookingNote	Рядок	Примітки щодо запису

2.3 Аналіз логіки системи

Більша частина логікі для сервісу для аунтифікації і авторизації буде використовувати сервіси з бібліотеки AspNetCore.Identity, що дозволяє не писати складну логіку для роботи з акаунтами а скористатися вже існуючою функціональністю в контролері (рис. 2.1). Клас, який відповідає за базу даних потрібно буде лише успадкувати від *IdentityDbContext*. Усі моделі будуть доступні в успадкованому контексті бази даних. Написання сервісів і репозиторієв для сервісу не є обов’язкові, у випадку якщо функціональність не потрібно буде адаптувати. Основні методи, які потрібно буде написати для

правильного функціювання системи це метод для входу, реєстрації і виходу з системи. Для авторизації потрібно створити модель *LoginModel* а для реєстрації створити *RegisterModel*. Також потрібно зробити допоміжний метод для генерації JWT. Ця логіка надає базову функціональність для аунтифікації і авторизації користувачів є досить простою і має можливість для розширення.

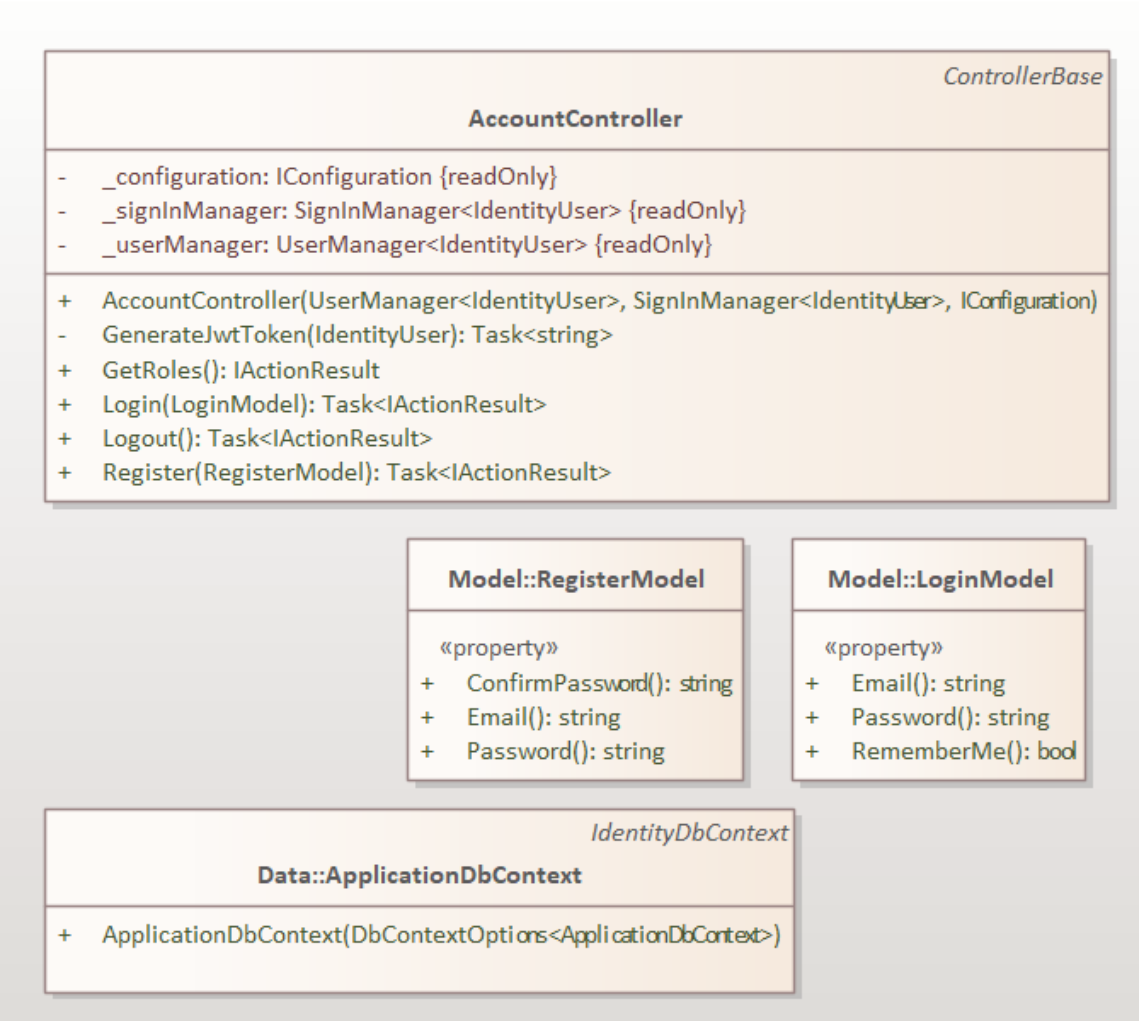


Рисунок 2.1 – Діаграма Auth серверу

Для сервісу який надає список послуг потрібно буде створити IRepository і IService для роботи з послугами і типами тварин і реалізувати їх для використання в подальшому, як на рисунку 2.2. Написання логіки зараз і в подальшому буде використовувати принцип інверсії залежності, що дозволяє проєктувати на рівні абстракції а не реалізації. Сам сервіс реалізує

лише логіку для додавання, зміни, видалення, отримання всіх елементів і отримання елемента по ідентифікатору. Сутності для сервісу були розглянути раніше. Сутність яка являє репозиторій буде використана для логіки сервісу, а сутність сервісу буде використана для логіку контексту усього мікросервісу (рис. 2.3).

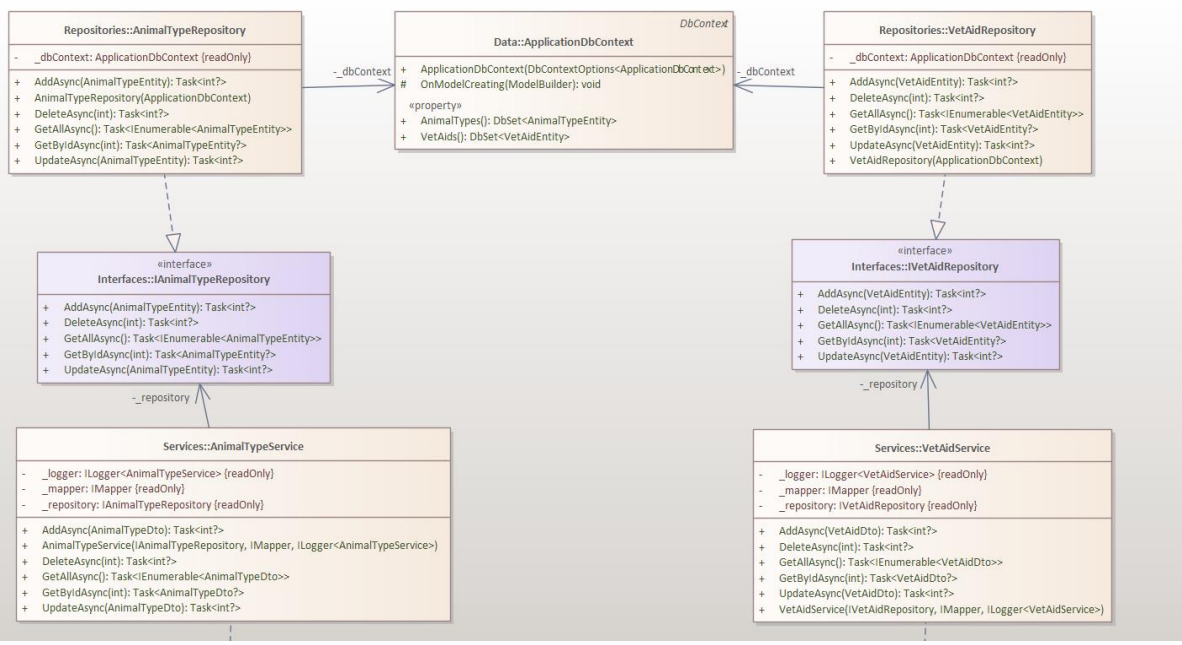


Рисунок 2.2 – Діаграма 1 для сервера VetAid

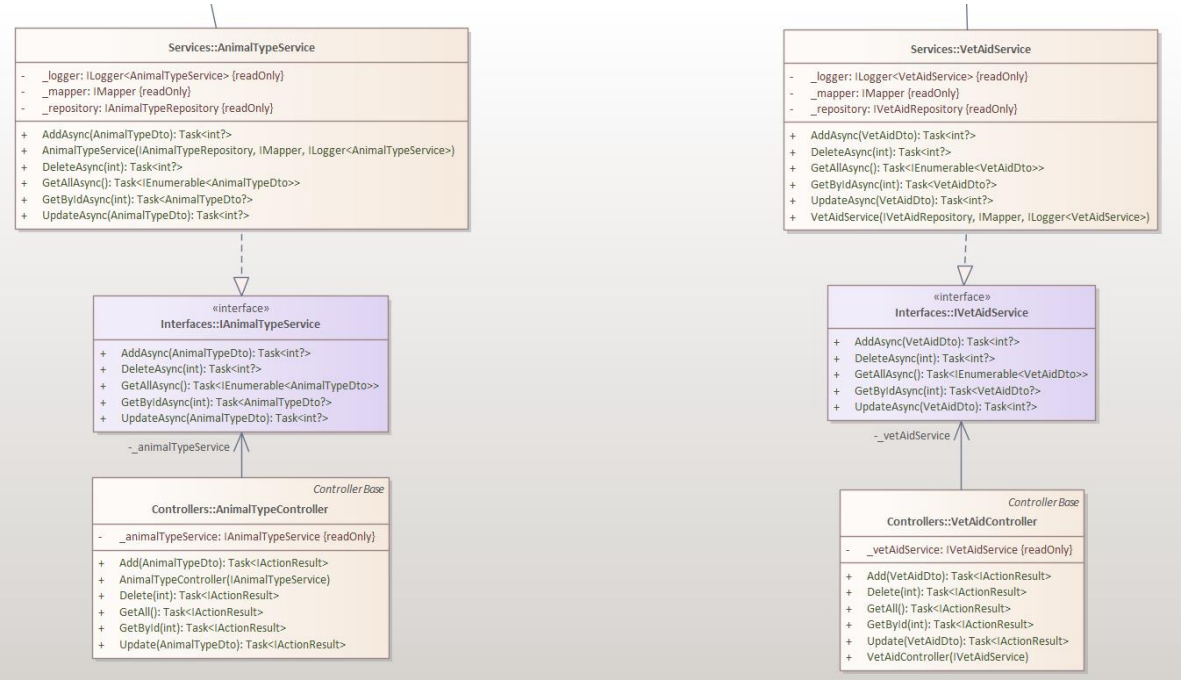


Рисунок 2.3 – Діаграма 2 для сервера VetAid

Сервер VetProfile теж буде написаний з використанням IService і IRepository (рис. 2.4, 2.5) [20]. Окрім логіки для роботи з профілем лікарів і їхньою зайнятістю присутня логіка для кешування і логіка, яка відповідає за перевірку доступу до ресурсу. Логіка для перевірки доступу використовується в обох контейнерах. Окрім звичайної логіки для отримання даних за ідентифікатором є методи на отримання розкладу з різними фільтрами і профіля за ідентифікатором.

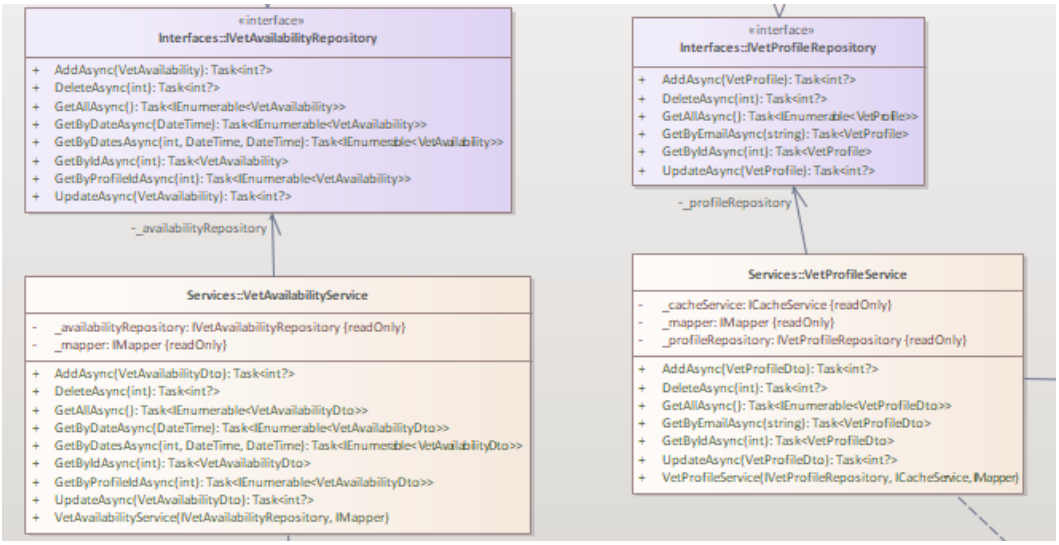


Рисунок 2.4 – Діаграма 1 для сервера VetProfile

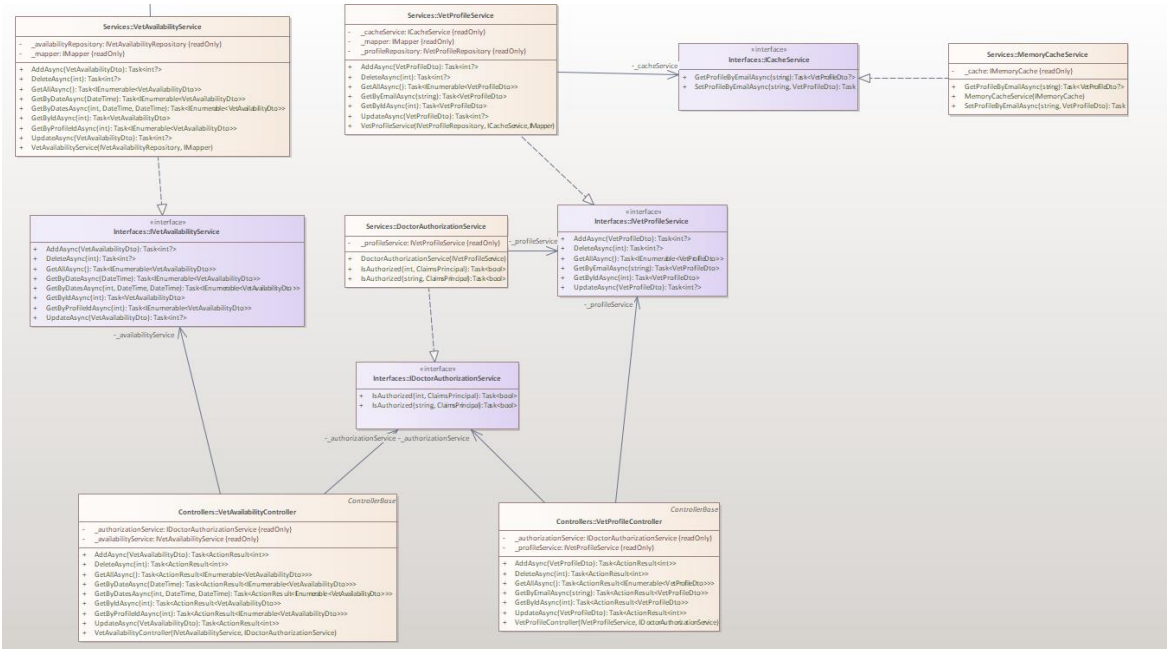


Рисунок 2.5 – Діаграма 2 для сервера VetProfile

Сервіс PetManagement надає доступ до профілів господарів хатніх тварин. Його реалізація схожа на VetProfile, він теж має IService і IRepository, теж використовує кешування і перевірку за доступом. Структура проєкта схожа, відрізняються лише методи (рис. 2.6, 2.7).

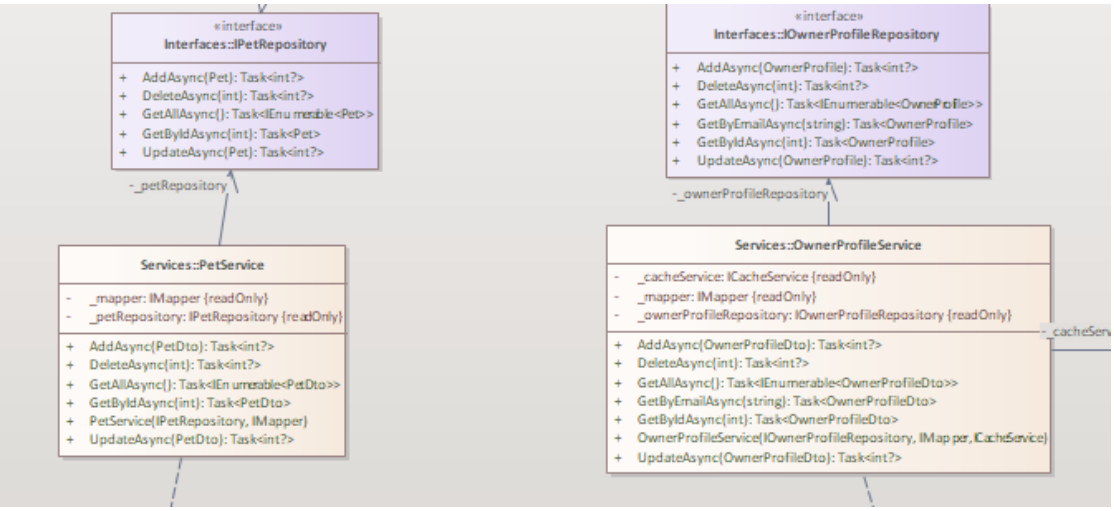


Рисунок 2.6 – Діаграма 1 для сервера PetManagement

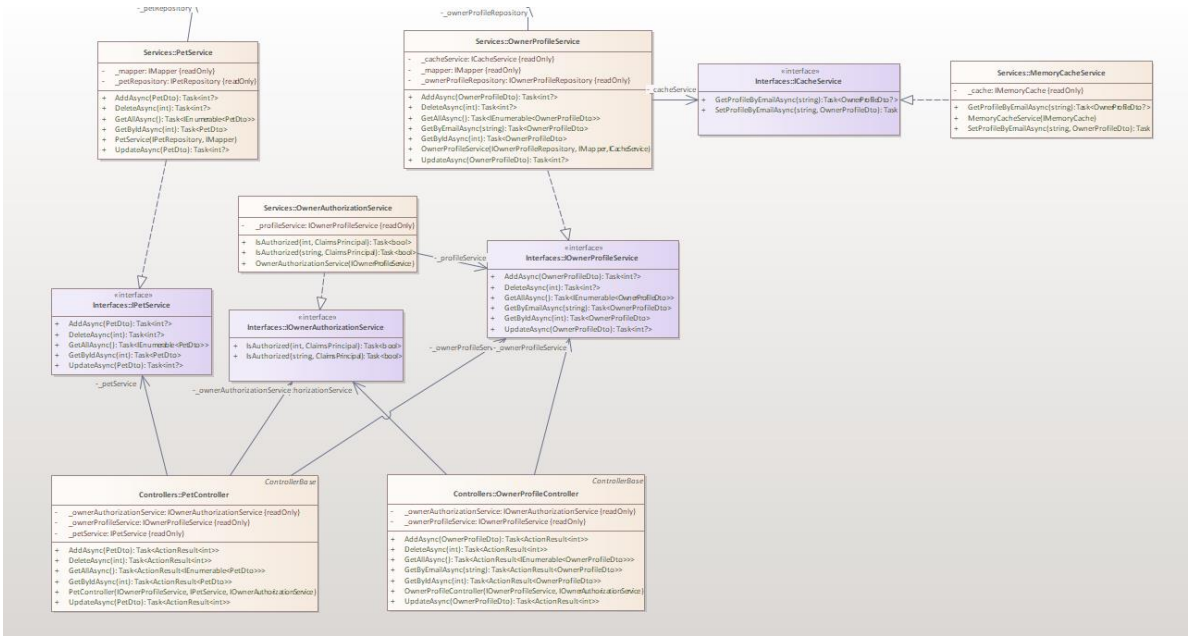


Рисунок 2.7 – Діаграма 2 для сервера PetManagement

Сервіс Appointment надає доступ до записів на прийом до лікаря. Через необхідність дивитися інформацію з інших він є клієнтом сервіса VetProfile для отримання розкладу лікаря, що буде реалізовано завдяки додатковому

класу *DoctorClient* (рис. 2.8). Appointment повинен мати можливість об’єднувати дані з двох сервісів, для отримання можливості користувачам системи отримувати вільні часу прийому. Окрім цього сам контролер має більше методів такі як отримання результатів завдяки фільтрації, або зміна статусу запису на прийом.

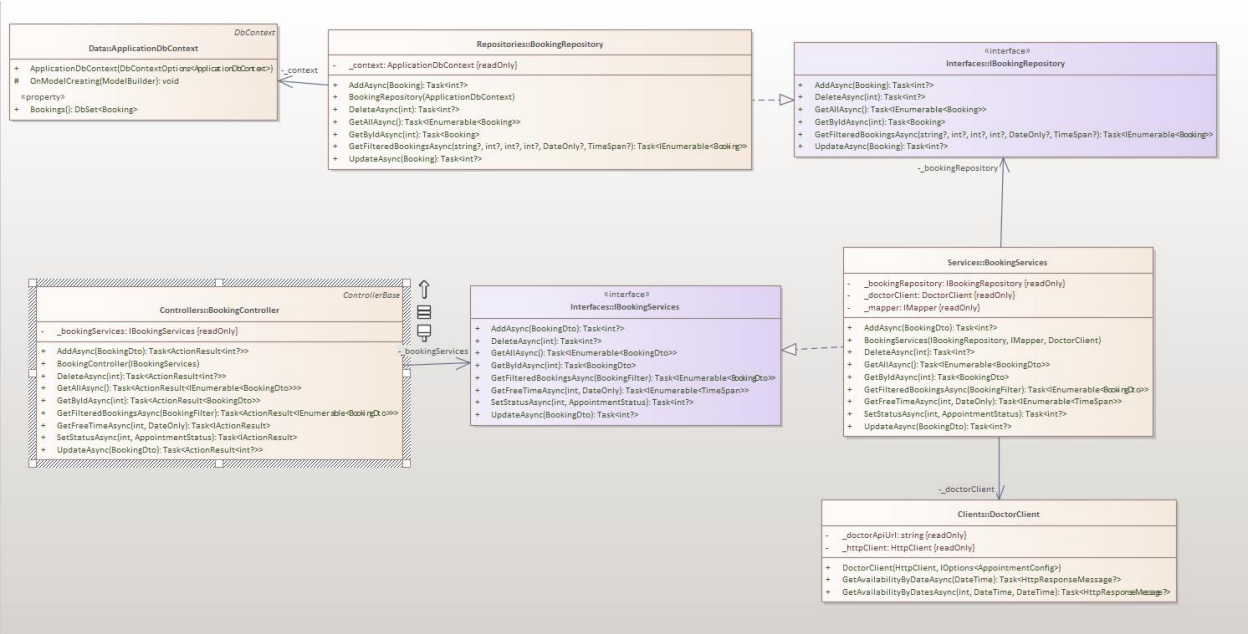


Рисунок 2.8 – Діаграма для сервера Appointment

2.4 Аналіз активностей системи

Після визначення основної частини логіки якою буде описана сереврна частина системи для ветеринарної клініки, потрібно визначити основні активності, тобто можливі послідовні дії починаючи від запиту клієнта і закінчуючи взаємодією з базою даних. Для наглядної ілюстрації можливих активностей системи буде використані UML-діаграми. UML – це мова для графічного опису об’єктного моделювання в галузі розробки програмного забезпечення, для моделювання бізнес-процесів, системного проєктування та відображення організаційних структур. Створення UML-діаграм перед

розробкою буде гарним кроком, для забезпечення правильної роботи системи.

Наглядною демонстрацією взаємодії користувача з системою буде UML діаграма користувача (рис. 2.9). Користувач може аунтифікуватися в системі, використовуючи запити до Auth мікросервісу, завдяки чому отримає JWT. Цей токен буде доданий до наступних запитів завдяки інтерполяції, що дозволяє не робити зв'язок усієї системи з Auth мікросервісом, а перевіряти токін через секретний ключ. Наступною дією, яку може зробити користувач системи буде перегляд ветеринарних послуг. Системі потрібно буде зробити запит до мікросервісу VetAid для того щоб отримати список можливих послуг, які можуть бути надані ветеринарною клінікою. В подальшому буде можливість фільтрувати список послуг по виду тварини і типу послуги. Далі, користувач може додати свою хатню тварину, щоб використовувати її в подальшому при записі на зустріч, і щоб лікарям було легше завдяки можливості переглянути історію хвороби. Наступною дією, яку може виконати користувач буде запис на зустріч. Після заповнення, можна буде переглянути список зустрічей, на яких користувач був зареєстрований.

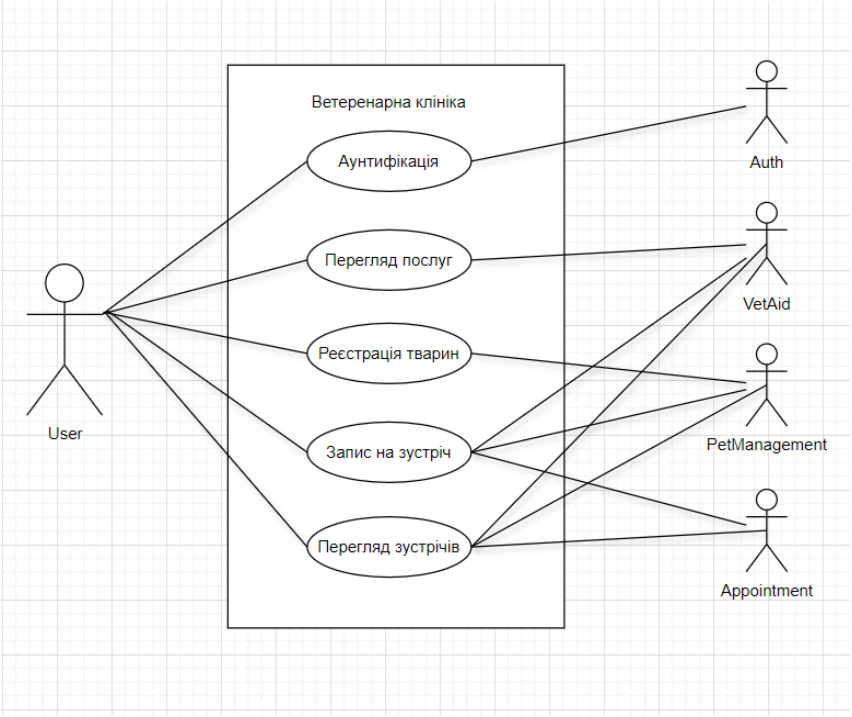


Рисунок 2.9 – UML діаграма користувача

Наступною наглядною демонстрацією логіки системи буде UML діаграма лікаря (рис. 2.10). Аунтифікація працює однаково для всіх ролей, тому її логіка така-ж як і у користувача. Додавання розкладу дозволяє лікарю додати інформацію що до його часів роботи. При роздробленому часі можна додавати розклад декілька разів в одну дату. Якщо запит на зустріч складені некоректно цей запит можна скасувати, або підтвердити якщо все правильно. Далі можна переглянути всі зустрічі, які провів або проведе цей лікар.

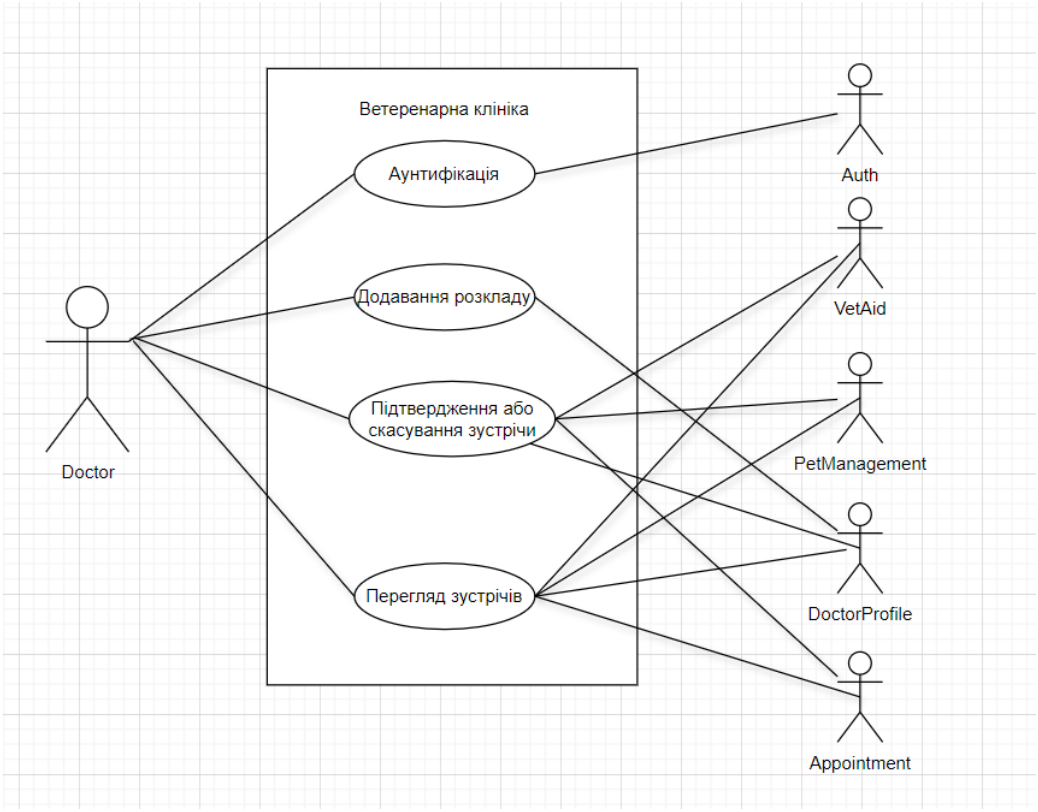


Рисунок 2.10 – UML діаграма лікаря

3 НАПИСАННЯ ВЕБЗАСТОСУНКУ

3.1 Підготовка до написання вебзастосунку

Першим етапом в написанні вебзастосунку для ветеринарної клініки буде вставлення всіх необхідних програм, таких як середа розробки, контейнерне середовище, менеджери пакетів, база даних і платформи на яких буде проходити розробка.

Платформою для розробки бекенду буде .NET Core. Це дозволить мені писати системи як під Linux так і під Windows. .Net Core підтримує декілька мов програмування такі як C#, Visual Basic і F#.

В якості середи розробки для бекенду мною було вибрано Microsoft Visual Studio 2022. Microsoft Visual Studio – це інтегроване середовище розробки (IDE), розроблене компанією Microsoft. Воно призначене для створення різноманітних програмних продуктів, включаючи програми для операційних систем Windows, вебсайти, вебзастосунки, мобільні застосунки, ігри, сервіси хмарної інфраструктури та багато іншого. Visual Studio надає широкий набір інструментів для розробки, таких як редактори коду, компілятори, відлагоджувачі, симулятори, дизайнери і багато інших, що полегшують роботу розробників і допомагають вирішувати різноманітні завдання.

Для написання бекенду з використанням технології ASP.NET в робочі Microsoft Visual Studio 2022 треба додати робоче навантаження ASP.NET and web development для можливості створення проєкту ASP.NET, використання веб компонентів таких як контролери, запуск проєктів з підтримкою Docker або протоколів HTTP або HTTPS.

Для управління пакетами бекенду було використано NuGet, який встановлений в Microsoft Visual Studio як основний менеджер пакетів. Ця система призначена для управління бібліотек для .NET Core і .NET Framework. Пакет NuGet є одним ZIP-файл із розширенням імені .nupack або

.nupkgta, що містить збірки .NET і потрібні для них файли з файлом маніфесту. Розробники можуть створювати ці пакети за допомогою клієнтської програми NuGet і публікувати їх у приватних чи громадських сховищах. Встановлення пакетів і бібліотек пакетів проводиться за допомогою графічного інтерфейсу або за допомогою командної строки.

В якості середи розроби для фроненда було вибрано Visual Studio Code. Visual Studio Code – це безкоштовний і легкий у використанні текстовий редактор, розроблений компанією Microsoft. Цей редактор призначений для редагування і написання коду для різних мов програмування, включаючи JavaScript, TypeScript, Python, PHP, C++, Java, HTML, CSS і багато інших.

Для написання фронтенду на React потрібно встановити платформу Node.js. Це дозволить використовувати JavaScript код поза браузером, що є потрібним для хостингу системи на сервері. Node.js дає пристроям можливість працювати з пристроями вводу-виводу через свій API, написаний на C++, підключати інші зовнішні бібліотеки, написані різними мовами, забезпечуючи виклики до них з JavaScript-коду.

Для пакетів для фронтенду було використано менеджер пакетів npm (Node Package Manager), який надає доступ до бібліотек і пакетів, що входять в склад Node.js. Встановлення пакетів проводиться за допомогою командної строки. При використанні менеджера залежностей для локального проєкту npm може встановити однією командою всі залежності проєкту через файл package.json.

Для запуску систему через контейнери потрібно встановити контейнерне середовище Docker. Це дозволить безпечно розгортувати мікросерівси, робити зв’язок між ними через API запроси. Для роботи з декільком сервісами буде використовуватися інструмент Docker Compose. Його конфігурація описується завдяки файлу docker-compose.yml, де знаходиться опис всіх контейнерів, їх залежності, налаштувань та інших параметрів.

В якості бази даних було використано PostgreSQL. Для полегшення системи запустити базу даних можна використовуючи Docker-контейнер, що містить базовий образ PostgreSQL. Для цього в docker-compose.yml потрібно описати сервер ім’ям, версією образу, паролем і портом на якому буде розгортатися сервіс (рис. 3.1). Тепер для використання PostgreSQL в іншому сервісі потрібно буде лише додати залежності і зміну середовища.

```
doctor_profile.api:
  container_name: doctor_profile.api
  build:
    context: .
    dockerfile: DoctorProfile/Dockerfile
  environment:
    - ASPNETCORE_ENVIRONMENT=Development
    - ASPNETCORE_URLS=http://+:5002
    - ConnectionString=server=postgres;port=5432;database=DoctorProfile;uid=postgres;password=postgres;
  depends_on:
    - postgres
  ports:
    - 5002:5002
postgres:
  image: postgres
  container_name: postgres
  environment:
    POSTGRES_PASSWORD: postgres
  ports:
    - 5433:5432
networks:
  default:
    driver: bridge
```

Рисунок 3.1 – Реєстрація PostgreSQL як Docker-контейнеру

3.2 Розробка backend

Розробка backend для системи ветеринарної клініки починається зі створення ASP.NET Core Web API проєкту. Версія фреймоврка була вибрана .NET 8.0, тип аунтифікації вибрано *None* через те що далі буде реалізована власна аунтифікація через JWT. Підключений докер і використання синтаксису високого рівня. Проєкт з такою конфігурацією буде здатен працювати через Docker-контейнерами або через HTTP або HTTPS протокол, буде мати всі переваги .NET 8.0 і не буде мати зайвого коду. Ці параметри будуть для всіх 5 сервісів.

Першим сервісом який був реалізований в рамках системи для ветеринарної клініки це *IdentityServer*. Перед розробкою потрібно встановити пакети і бібліотеки які знадобляться для написання системи. Для інтеграції з базою даних PostgreSQL буде використана бібліотека *Npgsql.EntityFrameworkCore.PostgreSQL*, для створення моделі і логіки буде використані бібліотеки *Microsoft.AspNetCore.Identity.EntityFrameworkCore* і *Microsoft.Extensions.Identity.Core*, для міграцій моделі до бази даних за технологією CodeFirst використовується *Microsoft.EntityFrameworkCore.Design* і для створення JWT буде використано *System.IdentityModel.Tokens.Jwt*. Після завантаження всіх цих бібліотек через графічну або консольну версію NuGet можна починати розробку сервісу.

Для використання ORM (Object Role Model) потрібно буде лише реалізувати власний контекст бази даних від *IdentityDbContext*, що дозволить використовувати поля користувачів і ролей, що є достатньо для системи ветеринарної клініки [21]. В разі якщо потрібно реалізувати власні класи які описують користувачів, ролей і потреби можна зробити це через узагальнені при наслідуванні власного ORM від *IdentityDbContext*.

Створення сутностей в базі даних на основі сутностей з коду називається CodeFirst. Першим етапом буде створення Initial міграції командою *dotnet ef migrations add Initial*. Для створення бази даних на основі цієї міграції потрібно щоб EF розумів за яким посиланням знаходиться база даних. Для інтеграції ASP.NET і PostgreSQL гарним варіантом буде зареєструвати фабрику через *AddDbContextFactory* і додати посилання на базу і її атрибути, такі як ідентифікатор клієнта бази даних та його пароль через параметри.

Для ініціалізації, засівання і описання обмежень буде створений клас *DbInitializer* з методом Initialize який приймає контекст власної бази даних, менеджер для користувачів, менеджер для ролів і налаштування для ідентифікації (рис. 3.2). В ньому спочатку конфігуруються параметри для паролю, потім параметри блокування. Далі конфігуруються можливі

символи, які будуть використані в імені користувача. Метод *EnsureCreated* створює базу даних якщо вона не існує і не створює якщо існує. Далі, за потребі, створюється ролі і користувачі які будуть використовувати ці ролі [22]. Тепер залишилося лише додати виклик цього метода в *Program.cs*, і база даних буде створена за обраною конфігурацією.

```
public static class DbInitializer
{
    1 reference
    public static async Task Initialize(ApplicationDbContext context,
        UserManager<IdentityUser> userManager,
        RoleManager<IdentityRole> roleManager,
        IdentityOptions identitiOptions)
    {
        identitiOptions.Password.RequireDigit = false;
        identitiOptions.Password.RequireLowercase = false;
        identitiOptions.Password.RequireNonAlphanumeric = false;
        identitiOptions.Password.RequireUppercase = false;
        identitiOptions.Password.RequiredLength = 4;

        identitiOptions.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(30);
        identitiOptions.Lockout.MaxFailedAccessAttempts = 5;
        identitiOptions.Lockout.AllowedForNewUsers = true;

        identitiOptions.User.AllowedUserNameCharacters =
            "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-._@+";
        identitiOptions.User.RequireUniqueEmail = true;


        context.Database.EnsureCreated();

        if (!context.Roles.Any())...

        if (!context.Users.Any())...
    }
}
```

Рисунок 3.2 – Ініціалізатор бази даних

Після написання бази даних потрібно додати в Program.cs зв’язок між сервісами *IdentityUser*, *IdentityRole* і *ApplicationDbContext*, для того щоб сервіси могли працювати з створеною раніше ORM. Конфігурація в ASP.NET проходить через виклик методів розширення, для створення зв’язку між базою даних і сервісами сервіси через узагальнення потрібно передати в метод *AddIdentity* а базу даних в *AddEntityFrameworkStores*. Тепер сервіси для ідентифікації, користувачів і ролей можна використовувати через DI.

Далі було створено контролер для роботи з акаунтами користувачів системи. Успадкуємо цей контролер від *ControllerBase*, додаємо атрибут для роутінга і атрибут *ApiController*. В конструкторі потрібно приймати

UserManager і *SignInManager* з узагальненням *IdentityUser*. Також передаємо конфігурацію для використання JWT у подальшому. Строюємо приватні поля лише для читання для всіх трьох параметрів і ініціалізуємо їх у конструкторі Це дозволить користуватися DI контейнером для їх ініціалізації.

Контролер для акаунтів повинен мати методи *Login*, *Logout*, *Register*. Метод Logout не має нічого приймати і просто визиває *SignOutAsync* у класа менеджера входу і повертає інформацію про те що вихід з системи пройшов успішно. Методи *Login* і *Register* будуть складніше через валідацію моделі, можливості ввести неправильні дані і через те що методи будуть повинні повертати токен (рис. 3.3). Для написання цих методів першим чином потрібно створити моделі *LoginModel* і *RegiserModel*, які будуть описувати ті параметри, яких буде достатньо для ідентифікації або реєстрації. Таким чином *LoginModel* буде описуватися поштою, паролем і прапорець вибору чи запам’ятовувати користувача, а *RegiserModel* буде описуватися поштою, паролем і повтором паролю. За потреби можна додати атрибути з DataAnnotations для валідації форми. Тепер в методі *Login* залишилося лише перевірити *LoginModel* на валідність, зробити запит через *PasswordSignInAsync* і отримати користувача з бази даних якщо запит пройде успішно. Далі потрібно згенерувати токен, він буде розглянути наступним, і зробити різне повернення для 3 різних сценарієв. Для реєстрації ще потрібно буде розпарсити *RegiserModel* в *IdentityUser*, створити користувача методом *CreateAsync* і увійти передавши користувача у метод *SignInAsync*.

```
[HttpPost("Login")]
0 references
public async Task<IActionResult> Login(LoginModel model)
{
    if (ModelState.IsValid)
    {
        var result = await _signInManager.PasswordSignInAsync(model.Email, model.Password, model.RememberMe, lockoutOnFailure: false);
        if (result.Succeeded)
        {
            var user = await _userManager.FindByEmailAsync(model.Email);
            var token = await GenerateJwtToken(user);
            return Ok(new { success = true, message = "Login successful", token });
        }
        return BadRequest(new { success = false, message = "Invalid username or password" });
    }
    return BadRequest(new { success = false, message = "Invalid model state" });
}
```

Рисунок 3.3 – Метод для входу

JWT – це стандарт для безпечної передачі інформації у форматі JSON між двома сторонами. Він складається з трьох частин: заголовка (header), корисної навантаження (payload) і підпису (signature). Заголовок містить метадані токена, такі як тип токена та тип шифрування, використовувані для створення токена. Зазвичай це містить два поля: *typ*, який вказує на тип токена (зазвичай JWT), і *alg*, який вказує на алгоритм, який використовується для підпису токена. Корисна навантаження це JSON-об’єкт, який містить корисну інформацію, яку потрібно передати зашифрованим токеном. Цей вміст може містити будь-яку корисну інформацію, яку ви хочете включити в токен, наприклад, ідентифікатор користувача, термін дії токена, дозволи тощо. Підпис це результат підписування заголовка та корисного навантаження токена за допомогою певного алгоритму підпису. Цей підпис використовується для перевірки автентичності токена при отриманні і для переконання в тому, що вміст токена не було змінено під час передачі. Для забезпечення безпеки потрібно і на стороні клієнта і на стороні сервер, який випускає цей токен, потрібно мати однаковий секретний ключ, який зазвичай зберігається в менеджері ключів. Також для забезпечення безпеки можна додати дані про те хто випустив токен і для кого він випускається [23]. Життєвий цикл токену повинен бути не дуже довгим, від декількох хвилин до місяця. Більше буде шкодити безпеці через можливість того, що токен буде скомпрометований.

Для написання випуску таких токенів в рамках C# і ASP.NET потрібно створити корисне навантаження через колекцію з класу Claims (рис. 3.4). Кориснє навантаження токену, який буде випускатися цим мікросервісом буд налічувати ім’я, пошта і ролі користувача. Далі токен підписується через семітичне шифрування використовуючи секретний ключ, який зберігається в конфігурації. Потрібно зауважити, що зберігати секретний ключ в конфігурації можливо лише в демонстраційних цілях, для цього краще підходить менеджер ключів. Після отримання ключа завдяки ньому потрібно створити *SigningCredentials*, який буде використовуватися для підписування

токену [24]. Для створення самого токену потрібно додати в конструктор сервіс який випускає токен, сервіс який є клієнтом токену, час створення токену в форматі UTC, claims який хочемо передавати клієнтам токену, скільки часу можно користуватися токеном і клас для підписування токену. Тепер використовуючи приватний метод *GenerateJwtToken* можна створити токен, тривалістю в 31 день. В реальних проєктах краще робити тривалість меншою. За потреби цей метод можно винести, створивши окремий сервіс для створення токенів.

```
2 references
private async Task<string> GenerateJwtToken(IdentityUser user)
{
    var claims = new List<Claim>
    {
        new Claim(ClaimTypes.Name, user.UserName),
        new Claim(ClaimTypes.Email, user.Email)
    };

    var roles = await _userManager.GetRolesAsync(user);
    foreach (var role in roles)
    {
        claims.Add(new Claim(ClaimTypes.Role, role));
    }

    var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(_configuration["Jwt:Secret"]));
    var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);


    var jwt = new JwtSecurityToken(
            issuer: _configuration["Jwt:Issuer"],
            audience: _configuration["Jwt:Audience"],
            notBefore: DateTime.UtcNow,
            claims: claims,
            expires: DateTime.UtcNow.AddDays(31),
            signingCredentials: creds);

    var encodedJwt = new JwtSecurityTokenHandler().WriteToken(jwt);
    return encodedJwt;
}
```

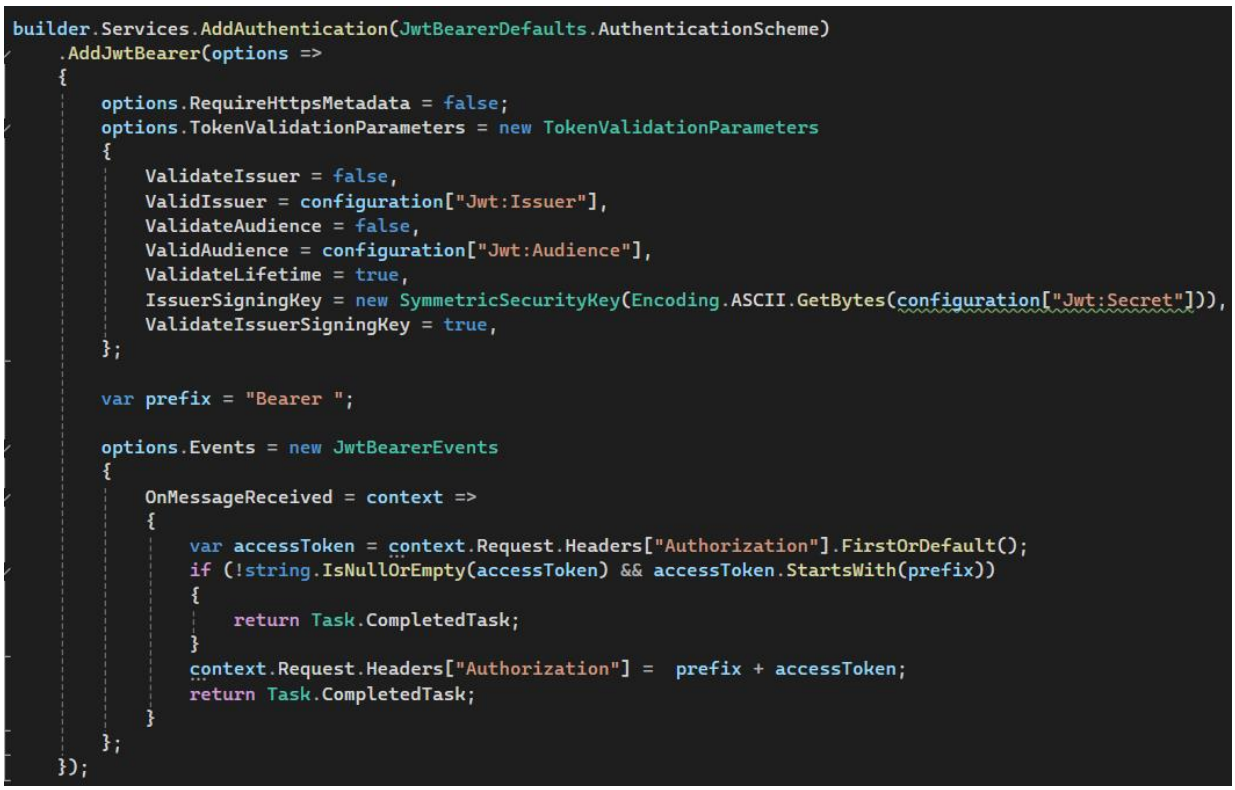
Рисунок 3.4 – Метод для створення JWT

Перед тестовим запуском сервісу потрібно додати деякі налаштування в *Program.cs*. В сервісах потрібно налаштувати політику CORS використовуючи метод *AddCors*. Додати контроллери методом *AddControllers*, які будуть знайдені завдяки атрибутам. Додати Swagger через *AddSwaggerGen*. Далі потрібно додати middleware для Swagger через методи *UseSwagger* і *UseSwaggerUI*. Тепер треба додати аунтифікацію і можна білдити проєкт.

Для тестування роботи потрібно запустити сервіс через HTTP протокол в Microsoft Visual Studio або розгорнути сервіс в контейнері через Docker. Після запуску, використовуючи Swagger, було перевірено роботу методів. Для отримання токену було введено правильний логін і пароль. Отриманий токен, система буде далі використовувати для ідентифікації через інтерполяцію запитів (рис. 3.5).

Рисунок 3.5 – Запуск сервісу через Swagger

Для використання токену для аутентифікації і авторизації в інших 4 мікросервісах, в них потрібно додати налаштування аутентифікації через метод *AddAuthentication* зі схемою через JwtBearer (рис. 3.6). В налаштуваннях схеми було додано налаштування, які описують те, що токен не вилідується за тим хто випустив токен і тим хто його отримав, але він валідується за часом життя токену і його підписом. Через конфігурацію проєкта отримується ім’я аудитора і того хто випустив токена, і ще ключ за яким перевіряється підпис. Також було додано подію, яка буде додавати префікс Bearer у випадку коли токен не має цього префіксу. Це робиться для полегшення використання токену в тестах, або при використанні Swagger. Тепер при отриманні в запиті до мікросервіса токена, він буде використовуватися для ідентифікації клієнта.



```
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.RequireHttpsMetadata = false;
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuer = false,
            ValidIssuer = configuration["Jwt:Issuer"],
            ValidateAudience = false,
            ValidAudience = configuration["Jwt:Audience"],
            ValidateLifetime = true,
            IssuerSigningKey = new SymmetricSecurityKey(Encoding.ASCII.GetBytes(configuration["Jwt:Secret"])),
            ValidateIssuerSigningKey = true,
        };

        var prefix = "Bearer ";

        options.Events = new JwtBearerEvents
        {
            OnMessageReceived = context =>
            {
                var accessToken = context.Request.Headers["Authorization"].FirstOrDefault();
                if (!string.IsNullOrEmpty(accessToken) && accessToken.StartsWith(prefix))
                {
                    return Task.CompletedTask;
                }
                context.Request.Headers["Authorization"] =  prefix + accessToken;
                return Task.CompletedTask;
            }
        };
    });
```

Рисунок 3.6 – Налаштування аутентифікації

Наступний мікросервіс, який був створений в рамках системи ветеринарної клініки – це *VetAid*. Цей сервіс надає користувачам список послуг і можливих хатніх тварин. Перед його написанням потрібно встановити бібліотеки для EntityFrameworkCore, для аутентифікації через JWT і автомапер.

Перед написання контексту для бази даних потрібно створити сутності, які описують ветеринарну послугу і вид хатньої тварини. Далі для кожної сутності потрібно написати конфігурацію, або через реалізувавши власний клас від *IEntityTypeConfiguration* з узагальненням у вигляді сутності, або прямо в контексті бази даних. Між сутностями *VetAid* і *AnimalType* потрібен зв’язок багато до багатьох, через те що як послуги можуть мати декілька видів творин, так і тварини мають декілька послуг. Після, в створеному контексті бази потрібно додати сутності через *DbSet* і конфігурацію в перевантажений метод *OnModelCreating*. Залишилося лише написати ініціалізатор, який на відмінно від попереднього сервісу буде приймати лише *ApplicationDbContext*.

Для роботи з самою базою даних можна скористатися підходом IRepository, тобто створити інтерфейси для роботи з сутностями *VetAid* і *AnimalType*. Для списку послуг і для типу тварин буде достатньо методів для отримання всіх сутностей, отримання сутності по ідентифікатору, додавання сутності, оновлення сутності і видалення сутності. Після написання інтерфейсів треба їх реалізувати. В конструкторі класу, завдяки DI, отримується контекст бази даних і зберігається в приватному, незмінному полі. Далі потрібно реалізувати всі контракти, якими описується інтерфейс, і використовуючи контекст бази даних реалізувати всю необхідну логіку. Вся логіка для взаємодії з базою даних через EntityFrameworkCore можна реалізувати завдяки операторам LINQ. При використанні цього оператора формується дерево запитів, чий виклик буде лише після матеріалізації через цикл foreach, або операторів матеріалізації. На відмінно від роботи IEnumerable, запити якого обробляються на рівні .NET коду *IQueryable* обробляються на рівні джерела даних, такого як база даних або ORM фреймворк, що може використовувати LINQ-to-SQL або LINQ-to-Entities для генерації та виконання оптимізованих SQL-запитів. Таким чином запити треба робити через LINQ, з обов’язковим запитом на матеріалізацію даних.

Перед написання основної логіки мікросервіса через підхід IService треба створити DTO (Data Transfer Object), і зробити мапінг між ними і сутностями з бази даних. Найкращим способом мапінгу в .NET буде використання автомаперу, який дозволяє описати процес мапінгу один раз і використовувати його в подальшому (рис. 3.7). Для описання цього процесу потрібно створити власний профіль і успадкуватися від *Profile*. В конструкторі цього класу потрібно створити зв’язок між класами через метод *CreateMap* і 2 узагальнень. Перше показує з якого класу буди проходити мапінг а друге у який клас він буде перетворений. Можна зробити реверс цього процесу методом *ReverseMap*. У випадку коли перетворення класів буде не один до одного потрібно використовувати метод *MapFrom*. В рамках цієї системи сутність *VetAidEntity* має колекцію сутностей типів тварин і її

потрібно перетворити в колекцію з типом *AnimalTypeDto*. Після реєстрації маперу в program.cs, його можна отримати в будь якому класі через DI. Для мапінгу достатньо визвати у екземпляра класу метод Map з узагальненням, параметром якого буде виступати той клас, в який потрібно перетворити об’єкт. Також можна мапити колекції, просто передавши їх через узагальнення.

Рисунок 3.7 – Налаштування профілю мапінга

В рамках реалізації основної логіки сервісу, було створено 2 інтерфейса для сервіса *IAnimalTypeService* і *IVetAidService*. Контракт цих сервісів передбачає, що кожен метод буде повертати Task, що робить можливість розробляти реалізацію у вигляді асинхронних методів. Самі методи використовують DTO, і налічують 5 методів: отримання всіх елементів, отримання елементів по ідентифікатору, додавання елемента, оновлення елемента, видалення елемента. Після написання контрактів було створено реалізацію інтерфейсів, які завдяки DI отримують репозіторій і мапер. Сама логіка сервіса являє собою мапинг об’єктів у правильний вигляд і використання методів з репозиторія для роботи з базою даних.

Перед написанням контролера потрібно додати транзиторні залежності між інтерфейсами і класами через метод *AddTransient* і передачу типів у вигляді узагальнень. Додати JWT аутентифікацію (рис. 3.6), налаштувати фабрику для бази даних. Зробити міграцію і створити на її основі сутності в PostgreSQL [25].

В мікросервісі для списку послуг і типів тварин має бути 2 різних кінцеві точки. Тому, було створено 2 контролери *AnimalTypeController* і

VetAidController. В перший контроллер було введена залежність від інтерфейсу *IAnimalTypeService* а в другом від *IVetAidService*. В кожному контроллері було зроблено по 5 методів, які будуть використовувати методи з сервісів. Через те, що отримувати список послуг і хатніх тварин може будь хто, перевірка авторизації для цих методів зроблена не була. Для інших, була реалізована перевірка через атрибут *Authorize* з параметрами *Roles = "admin"*, що буде означати, що лише адміни будуть здатні додавати і редагувати ці сутності.

Наступний мікросервіс, який був створений в рамках цієї роботи, це сервіс який керує профілем докторів і їх розкладом. Набір бібліотек і пакетів, які потрібно буде встановити перед написанням мікросервіса буде ідентичним з мікросервісом для каталогу послуг. Кроки зі створенням бази даних, ініціалізатора, створення міграцій і фабрики для бази даних будуть схожі окрім декількох змін. В конфігурації потрібно змінити зв’язок на один до багатьох. Завдяки цьому не потрібно створювати додаткову віртуально таблицю. Пошта повинна бути обов’язковою через її зв’язок з профілем. Далі, була створена міграція і по цій міграції було створена база даних за підходом CodeFirst.

В мікросервісі було створено 2 інтерфейси *IVetProfileRepository* і *IVetAvailabilityRepository*, для роботи з профілем користувача і для роботи з його розкладом. Перший інтерфейс містить 6 методів, окрім звичайних, отримання всіх елементів, отримання елементу за ідентифікатором, додавання елементу, його зміна та видалення, був створений контракт на отримання елемента за поштою. Пошту можна отримати з JWT токену, і цей метод дозволить робити перевірку на аутентифікацію і авторизацію перш ніж робити зміни в базі даних. В другому інтерфейсі окрім 5 звичайних CRUD (Create Read Rpdate Delete) методів, було реалізовано отримання розкладу на день за датою, розклад за ідентифікатором доктора і розклад доктора за ідентифікатором, початковою і кінцевою датою. Реалізація цих інтерфейсів проходить так само, отримується *ApplicationDbContext* через DI і робляться

запити через нього і з використанням LINQ. Було додано транзиторну залежність між *IVetProfileRepository* і *VetAvailabilityRepository* та *IVetProfileRepository* і *VetProfileRepository*.

Далі були створені DTO і створений профіль для мапінгу між ними. Першим інтерфейсом для сервісів був створений інтерфейс *ICacheService*, який має зберігати профіль ветеринарного лікаря з ключом у вигляді його пошти, і видавати їх за потреби. Реалізація контрактів інтерфейсу *ICacheService* буде виконана завдяки менеджеру кешування *IMemoryCache*. Для його підключення потрібно додати його сервіс в program.cs командою *AddMemoryCache* [26]. За потреби можна написати реалізацію з використанням redis або інших сервісів. Тут було реалізовано звичайне кешування, де в якості ключа виступає пошта, а в якості змінної виступає DTO об'єкт (рис. 3.8). Через можливість використання зовнішніх систем для кешування, при розширенні проєкту, методи повинні бути асинхронними. Зберігання кешу 2 години.

```
public class MemoryCacheService : ICacheService
{
    0 references
    public MemoryCacheService(IMemoryCache cache)...
    private readonly IMemoryCache _cache;

    2 references
    public async Task<VetProfileDto?> GetProfileByEmailAsync(string email) =>
        await Task.FromResult(_cache.Get<VetProfileDto>(email));
    2 references
    public async Task SetProfileByEmailAsync(string email, VetProfileDto profile)
    {
        _cache.Set(email, profile, TimeSpan.FromHours(2));
        await Task.CompletedTask;
    }
}
```

Рисунок 3.8 – Реалізація кешування

Далі були створені інтерфейси *IVetProfileService* і *IVetAvailabilityService*. Реалізація першого сервіса буде використовувати сервіс для кешування (рис. 3.9). При отриманні ветеринарного лікаря по

електронній пошті, спочатку буде запит до кешу, якщо запит проходить, повернеться профіль з кешу. Якщо запит не проходить то запит йде вже через репозіторій до PostgreSQL, і результат записується в кеш. Для додавання, оновлення і видалення профілю можна або оновити дані в кеші, або просто перезавантажити його відправивши null замість профілю. Інші методи будуть просто використовувати схожі методи з репозіторію, з додаванням мапінгу і перевірок за потреби. Реалізація другого сервіса вже не буде використовувати кешування, а лише буде відбравляти запроси через репозиторій.

```
public async Task<VetProfileDto> GetByEmailAsync(string email)
{
    var cachedProfile = await _cacheService.GetProfileByEmailAsync(email);
    if (cachedProfile != null)
    {
        return cachedProfile;
    }
    var profile = _mapper.Map<VetProfileDto>(await _profileRepository.GetByEmailAsync(email));
    if (profile != null)
    {
        await _cacheService.SetProfileByEmailAsync(email, profile);
    }
    return profile;
}
```

Рисунок 3.9 – Отримання профілю з кешу або з бази даних

Останній сервіс, який потрібно реалізувати в рамках цього мікросервісу – це сервіс який перевіряє чи має користувач доступ до логіки, яку він намагається виконати. Для цього було створено інтерфейс *IDoctorAuthorizationService* який має 2 контракти на методи *IsAuthorized* з різними параметрами. Перший метод приймає ідентифікатор користувача і *ClaimsPrincipal*, а другий його пошту і *ClaimsPrincipal*. Реалізація буде використовувати *IVetProfileService*, щоб отримати лікаря за поштою (рис. 3.10). В разі якщо роль користувача адмін, його дія буде схвалена в будь якому разі, якщо це доктор, дія буде схвалена лише у разі коли ідентифікатор лікаря в сутності збігається з ідентифікатором лікаря який отримується завдяки *GetByEmailAsync*. Другий метод з поштою теж схвалює дію, якщо її

робить адмін, але тепер пошту можна отримати з *ClaimsPrincipal* і перевірити завдяки неї.

```
6 references
public async Task<bool> IsAuthorized(int id, ClaimsPrincipal user)
{
    if (user.IsInRole("admin"))
    {
        return true;
    }
    else if (user.IsInRole("doctor"))
    {
        var email = user.Claims.FirstOrDefault(c => c.Type == ClaimTypes.Email)?.Value;
        if (email != null)
        {
            var profile = await _profileService.GetByEmailAsync(email);
            return profile != null && profile.Id == id;
        }
    }
    return false;
}
```

Рисунок 3.10 – Перевірка доступу

Далі було додано транзетивні залежності між *ICacheService* і *MemoryCacheService*, *IVetProfileService* і *VetProfileService*, *IVetAvailabilityService* і *VetAvailabilityService*, *IDoctorAuthorizationService* і *DoctorAuthorizationService*. Створено 2 контролери *VetAvailabilityController* і *VetProfileController*, перший контролер має використовувати сервіс *IVetProfileService*, другий використовувати *IVetAvailabilityService*. Обидва використовують *IDoctorAuthorizationService*. Для додавання, оновлення та змін, перед виконанням цієї логіки йде перевірка на доступ, у випадку якщо є пошта (наприклад додавання) – через пошту, у випадку коли пошти немає (наприклад видалення) – через ідентифікатор. В контролери *VetAvailabilityController* пошту можна отримати через зв’язок між сутностями, тобто спочатку знайти розклад через *GetByIdAsync*, а потім отримати з неї пошту, або ідентифікатор.

Наступний мікросервіс *PetManagement* дуже схожий на попередній. Для нього був створений *ApplicationDbContext* з моделями *OwnerProfile* і *Pet*, додана конфігурація і зв’язок один до багатьох, ініціалізатор. Репозиторії

цього класу виконують дефолтні CRUD операції, окрім *GetByEmailAsync* в репозіторії для власників тварин. Таким чином було створено інтерфейси *IPetRepository* і *IOwnerProfileRepository* і їх реалізація. Додана транзетивна залежність в program.cs. Цей мікросервіс теж має сервіс для кешування і перевірки доступу, і реалізовані так само. В контролерах теж є перевірка на рівень доступу. Загалом цей мікросервіс має дуже схожу логіку з попереднім, тому його створення проходить ідентичним чином, тому його детального опису не буде. За потреби кешування і перевірку можна узагальнити і винести в файл бібліотек.

Останній мікросервіс для бекенду надає логіку для запису на зустріч. Бібліотеки і пакети, які потрібні для цього сервісу ті самі, що і для попереднього. Для сервіса було створено *ApplicationDbContext*. Цей сервіс буде мати лише сутність Booking і конфігурацію для неї. В цієї сутності використовуються узагальнення, тому при написанні клієнта цього сервісу потібно звернути на це увагу. В репозиторіях окрім звичайних CRUD операцій був створений метод *GetFilteredBookingsAsync* для отримання значення за фільтрами (рис. 3.11). В цьому методі був використаний механізм лінивого виконання операцій, що дозволяє створити дерево запитів і лише потім сформувати запит повністю і передати його в базу даних. В методі створюється змінна типу *IQueryable* з узагальненням *Booking*, яка просто має запрос на отримання всіх записів на прийом. Далі перевіряється наявність кожного фільтра, у випадку коли фільтр наявний до запиту додається перевірка за цим фільтром [27]. Таким чином список записів на прийом фільтрується за поштою, ідентифікатором ветеринарного лікаря, ідентифікатором користувача, ідентифікатором послуги, ідентифікатором зареєстрованої хатньої тварини, датою запису і часом запису [28]. Перевірка для дати і часу запису проходять через перевірки чи знаходиться потрібна змінна в обраному діапазоні. Якщо якась змінна або більшість змінних будуть null, тоді пошуку по ним не буде. В кінці проходить матеріалізація колекції через метод *ToListAsync*.

```
public async Task<IEnumerable<Booking>> GetFilteredBookingsAsync(string? clientEmail, int? doctorId, int? vetAidId,
{
    var query = (IQueryable<Booking>)_context.Bookings;

    if (!string.IsNullOrEmpty(clientEmail))
    {
        query = query.Where(b => b.ClientEmail == clientEmail);
    }
    if (doctorId.HasValue)
    {
        query = query.Where(b => b.DoctorId == doctorId);
    }
    if (vetAidId.HasValue)
    {
        query = query.Where(b => b.VetAidId == vetAidId);
    }
    if (petId.HasValue)
    {
        query = query.Where(b => b.PetId == petId);
    }
    if (date.HasValue)
    {
        DateTime startOfDay = date.Value.ToDateTime( TimeOnly.MinValue ).ToUniversalTime();
        DateTime endOfDay = date.Value.ToDateTime(TimeOnly.MaxValue).ToUniversalTime();
        query = query.Where(b => b.AppointmentDateTime >= startOfDay && b.AppointmentDateTime <= endOfDay);
    }

    if (time.HasValue)...

    return await query.ToListAsync();
}
```

Рисунок 3.11 – Метод GetFilteredBookingsAsync

Цей мікросервіс виступає клієнтом для мікросервісу *DoctorProfile*, тому потрібно створи допоміжний клас *DoctorClient*, який буде приймати у конструкторі *HttpClient* і конфігурацію, з якої можна отримати посилання на сервіс *DoctorProfile* (рис. 3.12). З сервісу *DoctorProfile* потрібно отримати розклад докторів за вибраний день і розклад доктора за вибраний проміжок часу, тому було створено два методи. В першому методі була зроблена інтерполяція з *_doctorApiUrl* і шляхом за яким знаходиться метод для отримання розкладу. Для реалізації складеного запиту було використано клас *StringContent* який дозволяє додати параметри у вигляді рядка в контент з кодуванням UTF8. Після формування запит відправляється до *DoctorProfile* методом *SendAsync* і з нього отримується відповідь. Цю відповідь і повертає цей перший метод. Другий метод, на відмінно від першого, буде відправляти всі параметри в самому рядку посланні. Тут робиться інтерполяція між ідентифікатором доктора, датою початку і датою закінчення. Далі отримується фільтрований результат методом *GetAsync*. Цей результат повертає другий метод.

```
private readonly HttpClient _httpClient;
private readonly string _doctorApiUrl;

0 references
public async Task<HttpResponseMessage?> GetAvailabilityByDateAsync(DateTime date)
{
    var url = $"{_doctorApiUrl}/api/VetAvailability/ByDate";

    var request = new HttpRequestMessage(HttpMethod.Get, url);
    request.Content = new StringContent($"{{\"date\":\"{date:yyyy-MM-ddTHH:mm:ssZ}\"}}",
                                        System.Text.Encoding.UTF8,
                                        "application/json");

    var response = await _httpClient.SendAsync(request);

    return response;

}

1 reference
public async Task<HttpResponseMessage?> GetAvailabilityByDatesAsync(int profileId, DateTime startDate, DateTime endDate)
{
    var url = $"{_doctorApiUrl}/api/VetAvailability/ByDates?profileId={profileId}" +
        $"&startDate={startDate:yyyy-MM-dd}&endDate={endDate:yyyy-MM-dd}";

    var response = await _httpClient.GetAsync(url);

    return response;
}
```

Рисунок 3.12 – DoctorClient

Сервіс для цього інтерфейсу буде мати, окрім 5 дефолтних CRUD запитів, запит на отримання фільтрованих записів, встановлення статусу прийому (підтверджена, скасована, відбулася), і отримання вільного часу для зустрічи. Для запита на отримання фільтрованих записів було створено окремий файл *BookingFilter*, який має всі фільтри для отримання. Для встановлення статусу прийому, через особливості реалізації, достатньо лише викликати метод *UpdateAsync* і відправити в нього сутність лише з полями для ідентифікації і статусу. За потреби реалізації оновлення іншими способами можна додати ще один метод в репозиторій.

Для реалізації метода для отримання можливого вільного часу для запитів у доктора було отримано всі записи до цього лікаря но конкретну дату через отримання з фільтрацією (рис. 3.13). Наступне, було отримано розклад лікаря у форматі *HttpResponseMessage* і перетворено на звичайний контейнер. Для запису вільного часу було створено колекцію цього можливого часу. Проходимось по колекції розкладу, для кожного *availability* проходимось з кроком в 30 хвилин. У випадку коли нема жодної зустрічі на цей час додаємо час у колекцію. Повертаєм колекцію часу.

```
public async Task<IEnumerable<TimeSpan>> GetFreeTimeAsync(int id, DateOnly date)
{
    var bookings = await GetFilteredBookingsAsync(new BookingFilter() { DoctorId = id, Date = date });
    var availabilitysResponse = await _doctorClient.GetAvailabilityByDatesAsync(id, date.ToDateTime(new()), date.ToDateTime(new()));
    var availabilitys = await availabilitysResponse.Content.ReadFromJsonAsync<IEnumerable<VetAvailabilityDto>>();
    var availableTimes = new List<TimeSpan>();

    foreach (var availability in availabilitys)
    {
        while ((availability.EndTime.Value - availability.StartTime.Value) >= TimeSpan.FromMinutes(30))
        {
            if (!bookings.Any(b =>
                b.AppointmentDateTime.Value.TimeOfDay <= availability.StartTime.Value &&
                b.AppointmentDateTime.Value.AddHours(1).AddTicks(-1).TimeOfDay >= availability.StartTime.Value
            ))
            {
                availableTimes.Add(availability.StartTime.Value);
            }
            availability.StartTime = availability.StartTime.Value.Add(TimeSpan.FromMinutes(30));
        }
    }
    return availableTimes;
}
```

Рисунок 3.13 – Метод *GetFreeTimeAsync*

Далі було створено контролер *BookingController* який приймає *IBookingServices*. В контроллері реалізовані всі методи, які є в сервісі. Перед запуском потрібно додати всі залежності в program.cs.

3.3 Розробка frontend

Створення React проєкту відбувається командою *npx create-react-app* з параметром у вигляді назви проєкту. Для написання коду на typescript потрібно додати параметр *--template typescript* в команду. В проєкт були додані залежності від axios, react-router-dom, redux, jsonwebtoken, bootstrap. Були налаштовані базові стилі.

Для роботи з бекендом і ін’єкції токена в запити було зроблено інтерполяцію запита з використанням бібліотеки axios (рис. 3.14). Було створено функцію request, яка приймає путь до API, метод який цей API буде виконувати і контент запиту. Файл, в якому міститься функція має рядок baseURL, який відповідає за IP сервера. Робиться інтерполяція строк і отримується url сервісу. Далі формується *AxiosRequestConfig* з параметрами метод, місце знаходження ресурсу і контент, який має запит. З локального середовища отримується токен. Якщо цей токен не null, робиться інтерполяція запиту і додається в заголовок поле авторизації з параметром

Bearer + JWT. Запит відправляється і отримується результат. Для кожного з 5 мікросервісів для бекенду було додано метод, який спрощує запити до них.

```
const request = async (path: string, method: string, data?: any) => {
    try {
        const url = `${baseURL}${path}`;
        const config: AxiosRequestConfig = {
            method,
            url,
            data
        };
        const token = localStorage.getItem('token');
        if (token) {
            config.headers = { Authorization: `Bearer ${token}` };
        }
        const response = await axios(config);
        return response.data;
    } catch (error) {···
    }
};

export const authRequest = async (path: string, method: string, data?: any) => {
    return await request(`5000/api/${path}`, method, data);
};
```

Рисунок 3.14 – Інтерполяція запиту

Для створення адаптивного вебзастосунку було створені компоненти *Header* і *Footer*. *Footer* просто надає контактну інформацію ветеринарної клініки. *Header* надає навігацію, тобто дає список клікабельних посилань на сторінки системи. Меню для цього компонента є адаптивним і воно згортається в бургер на маленькому екрані. Компоненти було додано в функцію App. Доданий роутінг між сторінками вебзастосунку.

Було створена сторінка для входу *LoginPage* яка реалізує аунтифікацію в систему. Сторінка має хук для отримання навігації, щоб після входу переадресувати користувача на потрібну сторінку, хук для логіну, хук для паролю і хук для помилки. Метод *handleSubmit*, який визивається після натисканням користувача на кнопку входу і отримує дані з хуків для логіну і паролю. Ці дані він угруповує в інтерфейс *LoginData* і передає в функцію для входу (рис. 3.15). Ця функція асинхронно визиває метод *authRequest*

(рис. 3.14), з параметрами: путь до функції та її назва, тип метода і форма для входу. У відповіді повертається токен, який буде доданий до кожного наступного запиту. Сама сторінка являє собою форму, яка визиває метод *handleSubmit* при натисканні кнопки типу *submit* і має два поля, для пошти і для пароля. Кожне поле зв'язано з сетером хука.

```
export const login = async (loginData: LoginData) => {
    try {
        const response = await authRequest('Account/Login', 'POST', loginData);
        console.log(response);

        const token = response.token;

        localStorage.setItem('token', token);
        return token;
    } catch (error) {
        console.error('Error logging in:', error);
        throw new Error('Failed to log in');
    }
};
```

Рисунок 3.15 – Запит для входу

Наступною сторінкою є список всіх доступних послуг, якими може скористатися клієнт. Форма має фільтрацію за типом тварини і за типом самої послуги. Для реалізації цього функціоналу було використано 5 хуків, перші 3 відповідають за списки послуг, типів тварин і типів послуг а інші 2 за обрані тип тварини і тип послуги. Був створений метод *filteredServices*, який фільтрує сервіси. Згори сторінки знаходяться 2 випадаючи списки, в яких можна обрати вид фільтрування. Якщо не обирати нічого фільтр працювати не буде. Сам список на сторінці це розмаплений список послуг, які отримуються після фільтрування. Він показує такі дані як назва послуги, його опис, тип послуги, його тривалість і ціна. Також кожна така послугу моє кнопку, яка веде на сервіс для запису на прийом.

Сторінка, яка реалізує запис на прийом являє собою форму, в якій потрібно вибрати послугу, тип тварини, написати породу тварини, вибрати

доктора, вибрати дату і вибрати час зустрічі. Вибір типу тварини і породу надається лише після вибору послуги. Вибір дати лише після вибору доктора. Вибір години лише після вибору дати. Сторінка має хуки, які зберігають вибрані відповіді, хуки які надають варіанти відповідей, і хук який відповідає за помилку. Після вибору послуги, отримується список тварин для вибраної послуги. Після вибору доктора, отримується його розклад на місяць. Після отримання дати зустрічі, отримується його вільний час на день. Всі поля угруповуються, за потреби додається пошта і відправляється на бекенд.

Сторінка, яка реалізує список зустрічей являє собою список з відфільтрованими зустрічами за декількох параметрів. Через те, що до цієї сторінки може доєднатися як лікар так і користувач, вона буде мати різну поведінку для різних ролей. Доктор може фільтрувати список за будь якою кількості днів, підтверджувати і скасовувати зустрічі. Користувач може видалити запис, або змінити додаткову інформацію. У випадку якщо користувач не авторизований використовується тимчасовий ідентифікатор.

3.4 Розгортання проєкту

Для розгортання контейнерів через технологію docker compose потрібно створити Dockerfile для кожного сервіса (рис. 3.16). В цьому файлі треба взяти базовий образ платформи, для .NET це .NET SDK для NodeJs це node:17-alpine. Другий етап – це стадія сбірки, потрібно скопіювати поточний каталог командою COPY. Потрібно встановити поточний робочі каталог командою WOEKDIR. Командою RUN створюється програма та публікується в режимі Release в папці app. Третій етап – використання базового зображення для запуску програм ASP.NET Core. Налаштування середовища виконання на папку app. Копіювання всього середовища виконання. Встановлена точка входу в програму. Цей Dockerfile потрібно створити в кожному мікросервісі, з потрібними параметрами. Для фронтенда

базове зображення запущене через *npm install*, запуск відбувається через команду в CMD.

```
1  FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
2  COPY . /src
3  WORKDIR /src/Appointment
4  RUN dotnet publish -c Release -o /app
5
6  FROM mcr.microsoft.com/dotnet/aspnet:8.0
7  WORKDIR /app
8  COPY --from=build /app ./
9  ENTRYPOINT ["dotnet", "Appointment.dll"]
```

Рисунок 3.16 – Dockerfile

Було створено файл docker-compose.yml який визначає різні мікросервіси для розгортання застосунків в середовищі контейнерів Docker [29]. В нього були додані налаштування і посилання на Dockerfile для кожного мікросервісу системи і *postgres* який був розглянутий раніше. Перший мікросервіс, який відповідає за авторизацію буде розгорнуто як контейнер з ім’ям auth.api. Конфігурація для збірки контейнера береться з Dockerfile в папці IdentityServer. Встановлені середовищні змінні ASP.NET Core, такі як середа розробки, URL за яким працює сервіс і посилання на базу даних. Цей сервіс залежить від служби *postgres* і прослуховує порт 5000. Другий, третій і четвертий мікросервіси мають схожу конфігурацію і прослуховують потри 5001-5003. Мікросервіс, який відповідає за запис на прийом має таку-ж конфігурацію, але в його навколишнє середовище додані посилання на сервіси клієнтом яких він є. Прослуховує порт 5004. Останній мікросервіс відповідає за фронтенд, прослуховує порт 3000 і має посилання на кожен з мікросервісів для бекенду. Для об’єднання контейнерів між собою був створений міст. Будування проєкту відбувається командою *docker-compose build* а його запуск командою *docker-compose up*.

3.5 Тестування програми

Для тестування системи був збудований проєкт в режимі debug і розгорений в контейнер середовищі Docker. Першим чином була перевірена робота логіки бекенду через Swagger. Було відправлено запит на вхід і отримано токен (рис. А.1). Далі було перевірено запит на реєстрацію, де потрібно ще підтвердити пароль (рис. А.2). Перед перевіркою методів далі потрібно додати токен адміна для можливості робити будь який запит, який не суперечить логіки системи. З токеном адміна був зроблений запит на додавання виду тварини (рис. А.3). Після була отримана додана тварина (рис. А.4). Далі була додана і отримана послуга (рис. А.5, А.6). На рисунку А.7 був отриманий список всіх лікарів. Можна використати токен лікаря або токен адміна для запитів на додавання розкладу для цього лікаря (рис. А.8, А.9). На рисунку А.10 було додано запис на консультацію до лікаря а на рисунку А.11 отримання цього запису.

Далі був протестований фронтенд системи [30-32]. На рисунку А.12 зображено вхід в систему користувача. Після входу був отриманий і відфільтрований список послуг за типом тварини і видом послуги (рис. А.13). Була обрана послуга, тип тварини, написана порода тварини, обрано лікаря, обрана дата і час прийому (рис. А.14). Був перевірений список записів, де знаходиться запис сформований раніше (рис. А.15).

ВИСНОВКИ

У рамках кваліфікаційної роботи був розроблений вебзастосунок для ветеринарної клініки. У ході роботи було опрацьовано матеріали пов’язані з розробкою бекенду на платформі ASP.NET а фронтенду з використанням фреймворку React.

Завдяки детальному аналізу предметної області було створено сутності, які в достатній мірі описують роботу ветеринарної клініки. Була розгорнута база даних і використовуючи міграції вона була заповнена сутностями і даними.

Було створено сервіс для ідентифікацію з використанням JWT, який забезпечує можливість встановлення обмежень в логікі системи за ролями і за аутентифікацією. Було додано логіку для реєстрації і входу в систему.

Були створені мікросервіси для бекенду системи на платформі ASP.NET Core, які використовувати ідентифікацію через JWT і можуть спілкуватися між собою. В реалізації мікросервісів було використано сервісно-орієнтований підхід з використанням IRepository, IService для логіки системи і IClient для спілкування між різними компонентами системи. Також сервіси можуть взаємодіяти з базою даних і з кешем.

Було створено мікросервіс для фронтенду з використанням Bootstrap і React. Мікросервіс є клієнтом всіх мікросервісів для бекенду і надає можливість користувачам переглядати список послуг і записуватися на ці послуги. Сама сторінка використовує компоненту моделі, і рендерить кожен елемент сторінки окремо.

Систему було розгорнуто з використанням контейнерної технології Docker. Через використання мікросервісної архітектури кожен сервіс був розгорнутий в окремому контейнері. Окрім мікросервісів для бекенду і мікросервісу для фронтенду було розгорнуто базу даних яка використовує технологію PostgreSQL.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Творошенко, І.С. (2021). Технології прийняття рішень в інформаційних системах: навч. посібник. Харків: ХНУРЕ.

2. Liu, Z., Yu, H., Fan, G., & Chen, L. (2021, December). Reliability modeling and analysis of hospital information system based on microservices. In 2021 IEEE International Conference on Progress in Informatics and Computing (PIC) (pp. 313-318). IEEE.

3. Adam, S. I., Moedjahedy, J. H., & Maramis, J. (2020, October). RESTful Web Service Implementation on Unklab Information System Using JSON Web Token (JWT). In *2020 2nd International Conference on Cybernetics and Intelligent System (ICORIS)* (pp. 1-6). IEEE.

4. Nugraha, A. F., Kabetta, H., Buana, I. K. S., & Hadiprakoso, R. B. (2023, August). Performance and Security Comparison of Json Web Tokens (JWT) and Platform Agnostic Security Tokens (PASETO) on RESTful APIs. In *2023 IEEE International Conference on Cryptography, Informatics, and Cybersecurity (ICoCICs)* (pp. 15-22). IEEE.

5. Тітов, С. В., & Тітова, О. В. (2013). Web-сайти органів місцевого самоврядування як складова впровадження е-урядування в Україні.

6. Tvoroshenko, I., & Andrieieva, A. (2021). Development of web applications for remote learning of English.

7. Frycenty. (2016). Spring applications deployed on JLupin Next Server as a couple of JLupin Microservices [Image]. Wikimedia Commons. URL: https://commons.wikimedia.org/wiki/File:JLupin_wiki_fig3.png (дата звернення 22.04.2024).

8. Yakovleva, O., Kovač, M., Ardasov, V., & Yeremenko, I. (2023). Study on adding functionality to the Zoom online conference system for monitoring the participant activities. Public Administration and Regional Development, 19(1), 158-184.

9. Zeleniy, O., Rudenko, D., Lyubchenko, V., & Lyashenko, V. (2022). Image Processing as an Analysis Tool in Medical Research.

10. Nehra, M. (2019). Monolithic vs Microservice Architecture. Decipher Zone. URL: https://www.decipherzone.com/blog-detail/Monolithic-vs-Microservice-Architecture (дата звернення 22.04.2024).

11. Katona, J. (2021). Analyse the readability of LINQ code using an eye-tracking-based evaluation. Acta Polytech. Hung, 18, 193-215.

12. Katona, J., Kovari, A., Heldal, I., Costescu, C., Rosan, A., Demeter, R., ... & Stefanut, T. (2020, September). Using eye-tracking to examine query syntax and method syntax comprehension in LINQ. In 2020 11th IEEE International Conference on Cognitive Infocommunications (CogInfoCom) (pp. 000437-000444). IEEE.

13. Кобзев, І. В., Руденко, Д. О., & Яковлева, І. О. (2009). Мінімізація результату з'єднання відношень у запитах реляційної бази даних. Право і Безпека, (2), 255-259.

14. Microsoft. (n.d.). ASP.NET Overview. URL: https://learn.microsoft.com/en-us/aspnet/overview (дата звернення 20.04.2024).

15. Тітов, С. В., & Тітова, О. В. (2015). Оцінка юзабіліті освітніх сайтів: методи і технології. Вісник Харківської державної академії культури. Серія: Соціальні комунікації, (47), 127-134.

16. Aggarwal, S. (2018). Modern web-development using reactjs. International Journal of Recent Research Aspects, 5(1), 133-137.

17. Кочкін, А., & Руденко, Д. (2018). Використання Фреймворку Angular для Розробки Веб-застосунків.

18. Sun, J., Khan, F., Li, J., Alshehri, M. D., Alturki, R., & Wedyan, M. (2021). Mutual authentication scheme for the device-to-server communication in the Internet of medical things. IEEE Internet of Things Journal, 8(21), 15663-15671.

19. Гороховатський, В. О., & Творошенко, І. С. (2021). Методи інтелектуального аналізу та оброблення даних: навч. посібник.

20. Gorokhovatskyi V., Tvoroshenko I., Yakovleva O., Hudáková M., and Gorokhovatskyi O. (2024) Application a committee of Kohonen neural networks to training of image classifier based on description of descriptors set, IEEE Access, vol. 12, pp. 73376-73385.

21. Microsoft. (n.d.). Microsoft.AspNetCore.Identity.EntityFrameworkCore.IdentityDbContext Class (Microsoft.AspNetCore.Identity.EntityFrameworkCore) - ASP.NET Core. URL: https://learn.microsoft.com/ru-ru/dotnet/api/microsoft.aspnetcore.identity.entityframeworkcore.identitydbcontext?view=aspnetcore-8.0 (дата звернення 21.04.2024).

22. Yanholenko, O., Cherednichenko, O., Yakovleva, O., & Arkatov, D. (2020). A Model for Estimating the Security Level of Mobile Applications: A Fuzzy Logic Approach (Doctoral dissertation).

23. Lyashenko, V., Kobylin, O., & Minenko, M. (2018, October). Tools for Investigating the Phishing Attacks Dynamics. In 2018 International Scientific-Practical Conference Problems of Infocommunications. Science and Technology (PIC S&T) (pp. 43-46). IEEE.

24. Microsoft. (n.d.). SigningCredentials Class (System.IdentityModel.Tokens) - .NET Framework 4.8.1. URL: https://learn.microsoft.com/en-us/dotnet/api/system.identitymodel.tokens.signingcredentials?view=netframework-4.8.1 (дата звернення 21.04.2024).

25. Nguyen, A. T., Nguyen, T. T., & Nguyen, T. N. (2015, November). Divide-and-conquer approach for multi-phase statistical migration for source code (t). In 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (pp. 585-596). IEEE.

26. Microsoft. (n.d.). Caching in ASP.NET Core - .NET Core. URL: https://learn.microsoft.com/en-us/dotnet/core/extensions/caching (дата звернення 22.04.2024).

27. Microsoft. (n.d.). LINQ (Language-Integrated Query) - C#. URL: https://learn.microsoft.com/en-us/dotnet/csharp/linq/ (дата звернення 21.04.2024) (дата звернення 22.04.2024).

28. Ситников, Д. Э., Титов, С. В., & Титова, Е. В. (2011). Семантические свойства информативности ассоциативных зависимостей между признаками объектов в базах данных.

29. Reis, D., Piedade, B., Correia, F. F., Dias, J. P., & Aguiar, A. (2021). Developing docker and docker-compose specifications: A developers' survey. *Ieee Access*, *10*, 2318-2329.

30. Iryna, T., & Heorhii, M. (2021). TO THE QUESTION OF ANALYSIS OF EXISTING MECHANISMS OF WEB APPLICATION TESTING. Problems of modern science and practice, 1, 403. 10, 2318-2329.

31. Tvoroshenko, I., & Maksimenko, H. (2021). Research of regression and modular testing of web applications., Tvoroshenko, I. S., & Maksimenko, H. (2021). To the question of analysis of existing mechanisms of web application testing.

32. Tvoroshenko, I., & Kuznetsov, M. (2021). Research results of functional, white box and smoke testing methods for mobile applications.