Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

Програмна система управління персоналом, завданнями та інвентарем для готельного бізнесу. Серверна частина
(тема)

Виконав:
здобувач 4 року навчання
групи ПЗПІ-21-10

Кирило ГОТВЯНСЬКИЙ
(Власне ім'я, ПРІЗВИЩЕ)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)

Тип програми освітньо-професійна

Освітня програма Програмна інженерія
(повна назва освітньої програми)

Керівник доц. каф. ПІ Наталія РУСАКОВА
(посада, Власне ім'я, ПРІЗВИЩЕ)

Допускається до захисту
Зав. кафедри _____ Кирило СМЕЛЯКОВ
(підпис) (Власне ім'я, ПРІЗВИЩЕ)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
Кафедра програмної інженерії
Рівень вищої освіти перший (бакалаврський)
Спеціальність 121 – Інженерія програмного забезпечення
Тип програми Освітньо-професійна
Освітня програма Програмна Інженерія
(шифр і назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«___» _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Готвянському Кирилу Павловичу
(прізвище, ім'я, по батькові)

1. Тема роботи Програмна система управління персоналом, завданнями та інвентарем для готельного бізнесу. Серверна частина
Затверджена наказом по університету від 19.05. 2025р. № 397 Ст
2. Термін подання студентом роботи до екзаменаційної комісії 11.06.2025
3. Вихідні дані до роботи Розробити серверну частину для програмної системи управління персоналом, завданнями та інвентарем для готельного бізнесу
4. Перелік питань, що потрібно опрацювати в роботі
Вступ, аналіз предметної галузі, формування вимог до програмної системи, архітектура та проектування програмного забезпечення, опис прийнятих програмних рішень, тестування розробленого програмного забезпечення, висновки, додатки.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз предметної галузі	20.05.2025	*виконано*
2	Створення специфікації ПЗ	22.05.2025	*виконано*
3	Проектування ПЗ	24.05.2025	*виконано*
4	Розробка ПЗ	28.05.2025	*виконано*
5	Тестування ПЗ	30.05.2025	*виконано*
6	Оформлення пояснювальної записки	05.06.2025	*виконано*
7	Підготовка презентації та доповіді	06.06.2025	*виконано*
8	Попередній захист	09.06.2025	*виконано*
9	Нормоконтроль, рецензування	09.06.2025	*виконано*
10	Здача роботи у електронний архів	10.06.2025	*виконано*
11	Допуск до захисту у зав. кафедри	11.06.2025	*виконано*

Дата видачі завдання «20» «травня» 2025р.

Здобувач _____
(підпис)

Керівник роботи _____ доц. кафедри ПІ Наталія РУСАКОВА
(підпис) (посада, власне ім’я, ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи бакалавра, 82 стор., 17 рис., 3 таблиці, 12 джерел, 4 додатки

ГОТЕЛЬ, ASP.NET WEB API, C#, CLEAN ARCHITECTURE, MEDIATOR, NEON, POSTGRESQL, RENDER.

Об’єкт розробки – серверна компонента (back-end) програмної системи для управління персоналом, завданнями та інвентарем у готельному бізнесі. Цей розділ включає налаштування бази даних, створення бізнес-логіки для обробки запитів, забезпечення безпеки даних і організацію взаємодії між клієнтською та серверною частинами.

Мета розробки – створення надійної та ефективної серверної платформи, яка автоматизує ключові процеси управління персоналом і завданнями, а також облік інвентарю. Серверна частина повинна надавати засоби для безпечного зберігання й обробки інформації, виконання складних запитів до бази даних та інтеграції з іншими системами й сервісами.

Метод реалізації – використання ASP.NET Core як основи для створення Web API, що забезпечує виконання бізнес-логіки. Для зберігання даних обрано базу даних PostgreSQL, розгорнуту на платформі Neon, що забезпечує високу доступність і захист інформації.

У результаті створено масштабовану та безпечну серверну частину системи, здатну обробляти великі обсяги даних. Це дозволяє готелям ефективно управляти персоналом, відстежувати витрати інвентарю, оптимізувати постачання та підвищувати продуктивність роботи.

ABSTRACT

HOTEL, ASP.NET WEB API, C#, CLEAN ARCHITECTURE, MEDIATOR, NEON, POSTGRESQL, RENDER.

The object of development is the back-end component of a software system for managing personnel, tasks and inventory in the hotel business. This section includes setting up the database, creating business logic for processing requests, ensuring data security, and organising interaction between the client and server parts.

The goal of the development is to create a reliable and efficient server platform that automates key processes of personnel and task management, as well as inventory accounting. The server part should provide tools for secure storage and processing of information, execution of complex database queries and integration with other systems and services.

The method of implementation is to use ASP.NET Core as the basis for creating a Web API that ensures the execution of business logic. For data storage, the PostgreSQL database deployed on the Neon platform was chosen, which ensures high availability and data protection.

As a result, a scalable and secure server part of the system has been created, capable of processing large amounts of data. This allows hotels to efficiently manage staff, track inventory costs, optimise supplies and increase productivity.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

1NF – Перша нормальна форма (First Normal Form)

2FA – Двофакторна аутентифікація (Two-Factor Authentication)

2NF – Друга нормальна форма (Second Normal Form)

3NF – Третя нормальна форма (Third Normal Form)

ACID – Атомарність, узгодженість, ізольованість, стійкість (Atomicity, Consistency, Isolation, Durability)

API – Інтерфейс програмування додатків (Application Programming Interface)

CI/CD – Безперервна інтеграція/безперервне розгортання (Continuous Integration/Continuous Deployment)

CRUD – Створення, читання, оновлення, видалення (Create, Read, Update, Delete)

EF Core – Entity Framework Core (Entity Framework Core)

HTTPS – Протокол передачі гіпертексту з безпекою (HyperText Transfer Protocol Secure)

JWT – JSON Web Token (JSON Web Token)

ООП – Об’єктно-орієнтоване програмування (Object-Oriented Programming)

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

PMS – Система управління готелем (Property Management System)

ПЗ – Програмне забезпечення (Software)

SOLID – Принципи проектування програмного забезпечення (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion)

СУБД – Система управління базами даних (Database Management System)

UML – Уніфікована мова моделювання (Unified Modeling Language)

ВСТУП

У сучасних умовах, коли бізнеси постійно зростають і розвиваються, питання автоматизації рутинних завдань набуває дедалі більшої актуальності. Одним із таких процесів є контролювання співробітників та виконання ними завдань, ведення стану та історії готельного інвентарю.

Метою цієї роботи є створення програмної системи з використанням сучасної архітектури розробки ПЗ, що автоматизує виконання вищеописаних завдань. В основі розробки лежатиме мова програмування C#[1] у поєднанні з фреймворком ASP.NET Web API[2]. Для забезпечення швидкої та стабільної взаємодії між шарами у архітектурі буде використаний патерн Mediator[3]. Для зберігання даних буде використана СУБД PostgreSQL через свою швидкість, сучасність та масштабованість.

У ході дослідження буде проведено аналіз наявних методів контролю робочих процесів, які вже використовуються. Буде визначено вимоги до майбутньої системи та розглянуто можливості інтеграції з іншими сервісами. Особлива увага буде приділена вирішенню проблем оптимізації.

Архітектура системи базуватиметься на Onion Architecture, що сприятиме її гнучкості та масштабованості — важливим характеристикам для сучасних бізнес-процесів. При масштабуванні через велике навантаження на систему, ця архітектура дозволяє у короткі строки перенести додаток на мікросервісну архітектуру. Для того, щоб кожен шар архітектури був легко тестованим та простим для підтримки, у кожному шарі використовується підхід до організації коду Clean Architecture[4].

Таким чином, у результаті роботи буде створено програмну систему, яка автоматизує процеси керування інвентарем та персоналом, використовуючи C#, ASP.NET, патерн Mediator та СУБД PostgreSQL. Це дозволить суттєво підвищити ефективність процесу, скоротити час його виконання та покращити точність обробки даних.

Основні завдання дослідження включають:

- аналіз наявних конкурентів, їхні переваги та недоліки;
- розробку системи, що підвищить ефективність роботи;
- застосування Onion Architecture разом із Mediator для оптимізації комунікації між модулями системи;
- реалізацію системи на основі C# та ASP.NET;
- тестування функціоналу та продуктивності розробленого рішення.

У процесі розробки системи особлива увага буде приділена створенню інтуїтивно зрозумілого інтерфейсу користувача, що забезпечить зручність використання для співробітників різного рівня підготовки. Це дозволить мінімізувати час на навчання персоналу та підвищити загальну продуктивність роботи. Крім того, система передбачатиме можливість генерування детальних звітів та аналітики, що сприятиме прийняттю обґрунтованих управлінських рішень. Для забезпечення безпеки даних буде впроваджено сучасні методи шифрування та автентифікації. У перспективі планується розширення функціоналу системи шляхом інтеграції з хмарними сервісами та IoT-пристроями для автоматизації моніторингу інвентарю в реальному часі. Таким чином, розроблена система стане універсальним рішенням, яке відповідає сучасним вимогам бізнесу та відкриває можливості для подальшого вдосконалення.

Отже, ця робота є актуальною та перспективною в вищеописанному контексті. Запропонована система має широкі можливості для застосування у різних сферах бізнесу та промисловості.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Аналіз предметної галузі

Досліджуючи предметну область, пов’язану з розробкою програмної системи для управління персоналом та інвентарем для готельного бізнесу, можна зробити висновок, що ця проблема є актуальною для багатьох організацій. Управління персоналом та інвентарем відіграє ключову роль у плануванні поставок предметів та робочого часу персоналу, бухгалтерському обліку, вимагаючи значних ресурсів та часу.

На сьогоднішній день існує чимало підходів та інструментів для вирішення вищеописаних проблем, однак вони мають певні недоліки. Наприклад, ведення документації «на папері» та ручне опрацювання можуть спричиняти помилки та займати надто багато часу. Електронні методи (як «Excel») також не є ідеальними, адже їх впровадження та підтримка потребують значних зусиль, а ручне введення інформації підвищує ризик виникнення помилок.

Тому створення програмної системи, що допомагає автоматизувати ці фактори та витіснити «людський фактор» з рівняння, є важливим кроком для підвищення ефективності та точності цього процесу. Таке рішення дасть змогу зменшити витрати часу та трудовитрати, в той же час забезпечуючи більшу надійність.

На ринку вже представлені деякі програмні продукти, однак вони не завжди відповідають специфічним потребам власників готелів. Розробка власного рішення на основі Onion Architecture з використанням фреймворка ASP.NET Web API[2] надасть можливість створити масштабовану та гнучку систему, яка буде відмовостійкою, гарантуючи при цьому високу продуктивність та гарантії цілісності даних.

Отже, створення програмної системи для управління персоналом та інвентарем для готельного бізнесу є актуальним та перспективним завданням. Використання сучасного підходу до проектування архітектури разом із новітніми

фреймворками сприятиме створенню продуктивного та гнучкого рішення, що забезпечить виконання необхідних бізнесу завдань.

1.2 Виявлення та вирішення проблем

Спрощення процесу управління персоналом та інвентарем для готельного бізнесу є критично важливим для подібного бізнесу. Виконання цього завдання, як і багатьох інших, вручну займає багато часу та може призводити до помилок. Впровадження програмних рішень для цих задач дозволяє не тільки оптимізувати цей процес, а і підвищити швидкість обробки інформації, зменшити час на реакцію на нове завдання від персоналу та зменшити кількість неточностей у звітах про використання інвентарю.

Аналізуючи предметну галузь, можна виділити кілька основних проблем, пов’язаних з виконанням завдання за темою предметної області:

- ризик втрати записів про стан інвентарю або помилкове внесення даних: це може статися через людський фактор, що, у свою чергу, може спричинити розбіжності та потребувати додаткових ресурсів на виправлення;
- складність контролювання великої кількості персоналу: якщо персоналу багато, становиться дуже важко стежити за виконанням завдань, за статусом завдань;
- можливість втрати інформації: ручне введення даних не гарантує їх збереження, оскільки через недбалість або технічні збої інформація може бути втрачена.

Щоб уникнути цих проблем, доцільно застосовувати програмні рішення, які дозволяють автоматизовано виконувати наступні дії. Такі системи забезпечують швидкість, точність і надійність збору даних, а також надають можливість електронного зберігання даних.

Для успішного впровадження подібної системи вона повинна бути інтуїтивно зрозумілою та простою у використанні, що сприятиме її ефективному

застосуванню співробітниками та менеджерами. Крім того, важливим аспектом є можливість експортування даних та наявність графіків, що забезпечить зручність аналізу витрат та захисту даних.

Однак, при розробці можуть виникнути певні труднощі, серед яких:

- складність інтеграції з уже існуючими готельними додатками, що можна вирішити через API-інтерфейси;
- система повинна масштабуватись та повинна підходити як для малих готелів (не перегружаючи інтерфейс непотрібними функціями), так і для великих готелів з великим навантаженням на систему;
- неправильно розподілені ролі нелогічним шляхом (напр., адміністратор не повинен керувати інвентарем, покоївка не повинна бачити всіх співробітників і т. д).

Крім того, розробка подібного програмного забезпечення має враховувати такі аспекти:

- забезпечення безпеки даних шляхом використання механізмів шифрування та автентифікації користувачів;
- гарантування масштабованості системи через впровадження контейнеризації (наприклад, Docker), що дозволить стабільно працювати при збільшенні навантаження;
- реалізація «чистої» архітектури та використання Mediator для швидкого, зрозумілого та логічного обміну даними між сервісами.

Ще одним викликом для такого продукту є конкуренція. Аналіз ринку показує, що такі програми у широкому доступі існують і також великі мережі готелів мають свої власні програми.

RoomRaccoon [5] – це комплексна система управління готелем, призначена для незалежних готелів, апартаментів та гостьових будинків, яка надає різноманітні функції, включаючи інтегрований механізм бронювання, онлайн реєстрацію та автоматизовані додаткові продажі. Приклад інтерфейсу наведений на рисунку 2.1.

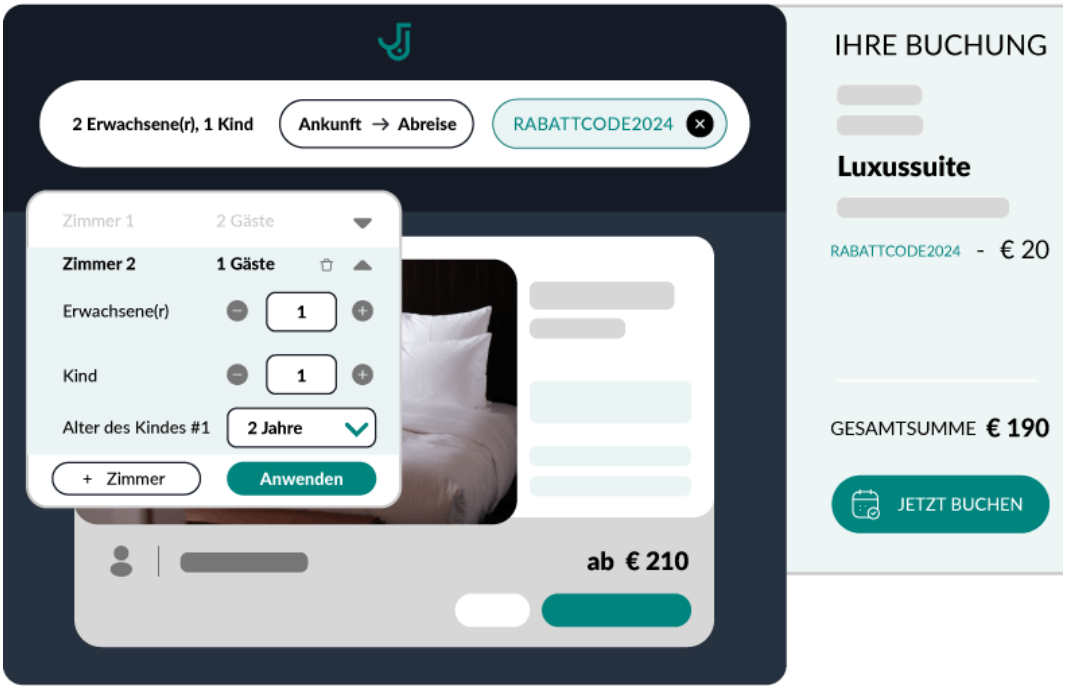


Рисунок 1.1 – Інтерфейс RoomRaccoon [5]

Переваги:

- RoomRaccoon має синхронізацію бронювань у реальному часі, що допомагає уникнути овербукінгу та покращує управління номерами;
- RoomRaccoon надає можливості для персоналізованого спілкування з гостями, що підвищує якість обслуговування та задоволеність клієнтів.

Недоліки:

- нові користувачі можуть зіткнутися з певною складністю при освоєнні системи, що вимагає додаткового часу на навчання;
- система може не мати стільки варіантів персоналізації, які є «базовими» (напр., взяття персоналом предметів зі складу), як деякі інші системи управління готелями, що робить її не гнучкою.

Hotelogix [6] – це комплексна система управління готелем (PMS), яка має як переваги, так і недоліки, згідно з відгуками користувачів та експертів. Приклад інтерфейсу наведений на рисунку 2.2.

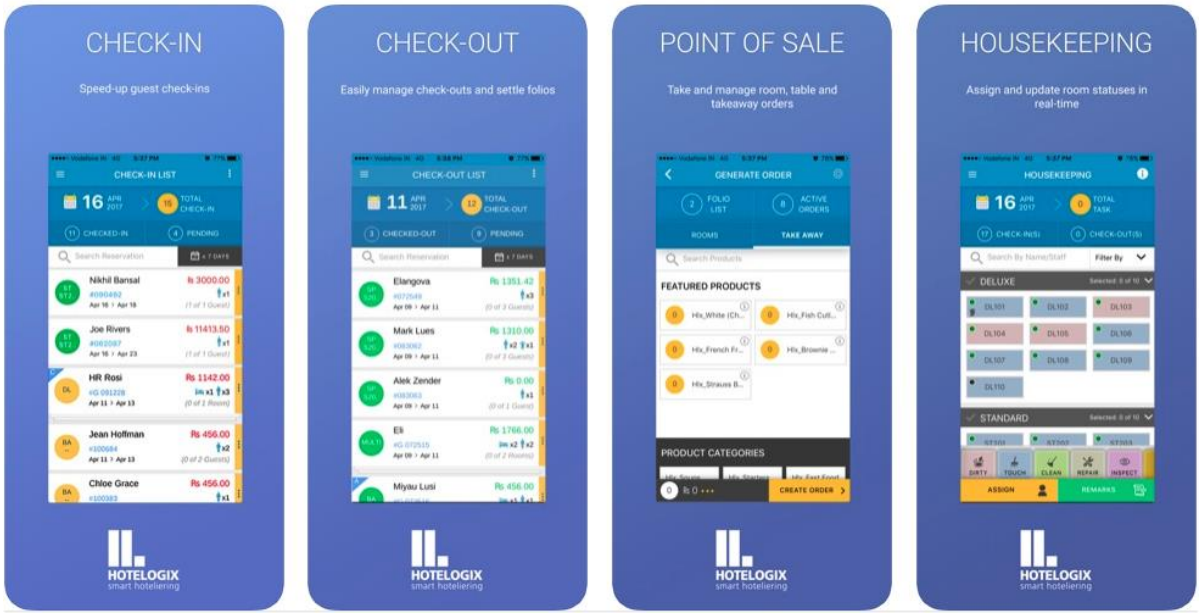


Рисунок 1.2 – Інтерфейс Hotelogix [6]

Переваги:

- Hotelogix пропонує багатий набір функцій, включаючи управління бронюваннями, інтеграцію з каналами продажів та детальну аналітику, що допомагає оптимізувати операційну діяльність готелю;
- наявна цілодобова підтримка клієнтів;
- високий рівень шифрування даних.

Недоліки:

- великі проблеми з інтеграцією з уже існуючими сервісами через застарілість;
- обмежений функціонал мобільної версії порівняно з комп'ютерною. Мобільна версія також є додатково оплачуваною.

Розробка програмної системи для управління персоналом та інвентарем у готельному бізнесі є надзвичайно актуальною з огляду на сучасні потреби галузі. Аналіз предметної області показує, що традиційні методи управління, такі як паперова документація чи використання електронних таблиць, є неефективними через високий ризик помилок, втрату даних і значні витрати часу. Існуючі програмні рішення, такі як RoomRaccoon чи Hotelogix, хоча й пропонують певні

функціональні можливості, мають обмеження, зокрема складність інтеграції, недостатню гнучкість для специфічних потреб готелів, обмежений функціонал мобільних версій та труднощі в освоєнні для нових користувачів.

Створення нової системи на основі сучасних технологій, таких як Onion Architecture та ASP.NET Web API, дозволить подолати ці недоліки, забезпечивши високу продуктивність, масштабованість, безпеку даних і зручність використання. Автоматизація процесів управління персоналом та інвентарем зменшить вплив людського фактору, оптимізує операційні процеси, підвищить точність звітності та швидкість реагування на завдання. Така система відповідатиме специфічним потребам готельного бізнесу, буде гнучкою для малих і великих закладів, а також забезпечить інтуїтивний інтерфейс і надійне зберігання даних.

Таким чином, розробка подібної системи є перспективною, оскільки вона не лише вирішує актуальні проблеми готельної індустрії, але й пропонує конкурентоспроможне рішення, здатне адаптуватися до сучасних викликів і потреб бізнесу.

1.3 Постановка задачі

Метою проекту є оптимізація управління персоналом, створення звітності, керування інвентарем та вживанням предметів на складі, а також надання зручного та ефективного інтерфейсу.

Для забезпечення найліпшого досвіду користування додатком та для розуміння потреб, на першому етапі необхідно чітко визначити вимоги до системи, які відповідатимуть запитам і очікуванням користувачів. Це включає окреслення цілей і прогнозованих результатів проекту, вимог до функціональності, надійності та продуктивності. Особливу увагу слід приділити вимогам до надійності, оскільки система міститиме конфіденційну інформацію персоналу (напр., номер телефону, пошта, ПІБ) і має бути безпечною та стабільною, включаючи механізми перевірки на помилки та відновлення. Також необхідно окреслити вимоги до продуктивності, такі як швидкість роботи та масштабованість.

Сформовані вимоги слід перевірити на відповідність очікуванням користувачів і цілям проекту. Важливо зазначити, що вони можуть коригуватися або доповнюватися в ході розробки.

Розробка плану створення системи допоможе забезпечити виконання завдань відповідно до запланованого графіка. У ньому необхідно передбачити всі етапи процесу, які можуть знадобитися. План має відображати послідовність реалізації проекту, строки виконання кожного етапу та залежності між ними.

Проєктування структури бази даних є ключовим етапом у розробці системи. Потрібно вибрати технологію, яка найбільше відповідає нашим вимогам. Це передбачає вибір відповідної СУБД та інших технічних рішень. Після визначення технологій необхідно грамотно спроєктувати структуру бази даних. Грамотне проєктування сприятиме покращенню продуктивності системи та дозволить уникнути проблем при додаванні функціоналу в майбутньому. Важливо розуміти, як використовуватиметься кожна таблиця, щоб правильно підібрати індекси, які будуть застосовуватися, створити представлення (Views) для спрощення роботи з даними, визначити ролі користувачів і доступні їм функції.

Після завершення проєктування бази даних необхідно реалізувати її. Для цього буде застосовано підхід Code-First з використанням фреймворку Entity Framework [7]. Він дозволяє швидко й ефективно створювати всі необхідні таблиці, описувати їх взаємозв’язки та завдяки вбудованому механізму міграцій підтримувати й розширювати базу даних. Для більш гнучкого налаштування будуть використовуватися SQL-скрипти, які також інтегруватимуться у міграції.

Після створення бази даних необхідно перейти до проєктування додатка. Головне завдання на цьому етапі – визначити всі вимоги, як функціональні, так і нефункціональні, а також встановити ролі користувачів і способи їхньої взаємодії із системою.

Оскільки обрано Onion Architecture, необхідно розподілити функціональність між окремими шарами.

Після визначення шарів слід спроєктувати взаємодію між ними. Важливо розуміти, які шари повинні знати один про одного та як вони будуть обмінюватися даними.

Після завершення проєктування загальної структури можна переходити до розробки окремих шарів. Для кожного з них слід чітко визначити функції, доступні методи, об’єкти, а також внутрішню структуру. Дотримуючись принципів ООП і SOLID, слід розробити класи та визначити їхні зв’язки.

Після завершення проєктування системи можна переходити до її реалізації. Для цього буде використано сучасне середовище розробки JetBrains Rider [8]. Це потужний інструмент із широким функціоналом, що задовольняє потреби розробників при створенні складних веб-додатків.

Коли додаток буде створено, слід подбати про його розгортання. Для його розгортки буде використано контейнеризацію. Це дозволить створювати окремі ізольовані середовища. Один раз налаштувавши середовище виконання, його можна легко розгорнути на будь-якому сервері. Для локальної розробки використовуватиметься Docker [9] – інструмент контейнеризації, що спрощує налаштування взаємодії між сервісами. У продакшн-середовищі буде використано Render [10] – сервіс для розгортання додатків у контейнері.

Заключний етап – тестування навантаженням. Для цього можна використовувати різні інструменти або створити власний скрипт, що асинхронно викликатиме API багаторазово. Це дозволить виявити вузькі місця, які можуть вплинути на продуктивність при значному навантаженні.

З урахуванням теми роботи та всіх перелічених аспектів, можна підсумувати перелік завдань:

− формування вимог до продукту;

− створення плану розробки;

− проєктування структури бази даних за допомогою СУБД PostgreSQL;

− створення бази даних;

− розробка загальної архітектури за допомогою C# та IDE Rider;

− визначення відповідальностей шарів;

− проєктування взаємодії між шарами;

− створення структури шарів;

− реалізація шарів;

− розгортання інфраструктури;

− тестування продуктивності.

Після визначення завдань і функціоналу, що необхідно реалізувати, можна розпочати формування вимог до програмної системи.

2 ФОРМУВАННЯ ВИМОГ ДО ПРОГРАМНОЇ СИСТЕМИ

Програмна система, призначена для управління персоналом, завданнями та інвентарем для готельного бізнесу, потребує створення надійної серверної частини, орієнтованої на взаємодію з клієнтською частиною через API. Основними користувачами системи є працівники готелю, такі як адміністратор, менеджер, інвентарний менеджер, покоївка, технічний працівник, які повинні мати доступ до різноманітних функцій для ефективного адміністрування готелем. Це охоплює можливості додавання, редагування, видалення різноманітних даних (напр., завдань) та їх елементів, а також управління ролями працівників.

Для реалізації серверної частини системи обрано технологію ASP.NET Core Web API на платформі .NET 9, оскільки ці інструменти вирізняються високою продуктивністю, гнучкістю та широкими можливостями для створення веб-додатків і API. Вони дозволяють швидко розробляти та розгортати рішення, забезпечуючи при цьому ефективність і масштабованість.

Функціонал системи має бути розподілений за ролями та включати набір операцій адміністрування персоналом та відслідковуванням їх продуктивності, а також можливість купівлі квитків для звичайних користувачів. Потреби системи наведено нижче:

− додавання, редагування, видалення інформації про працівників;

− додавання, редагування, видалення інформації про завдання;

− додавання, редагування, видалення інформації про інвентар та історію взаємодії з ним;

− сповіщення працівників про нове призначене їм завдання;

− перегляд графіку продуктивності працівників;

− можливість заздалегідь визначати завдання та призначати їх працівникам;

− реєстрація нових співробітників адміністратором через електронну пошту співробітника;

− змінення стану завдання як виконувачем завдання, так і перевіряючим завдання (Not Accepted → In Progress → Completed → Approved/Rejected)

Інтеграція платіжних систем та контракти з поставниками меблів, предметів гігієни тощо не будуть враховані для цієї роботи.

Вищеперераховані модулі можуть розглядатись у наступних версіях програми. Наприклад, інтеграція з платіжними системами ПриватБанку або одразу ж Visa/MasterCard.

Для підтримки та розгортання системи використовується платформа Render, яка дозволяє безкоштовно розгортати багато сервісів, у переліку яких є API. Платформа надає базові метрики для відслідковування навантаження та ефективності системи. Також присутній перегляд файлів логування, які надає програма під час своєї роботи. Вищеперераховані факти в купу з можливістю безкоштовного використання роблять цю платформу досить конкурентоспроможною, порівнюючи, наприклад, з Google Cloud чи Azure.

Щоб забезпечити швидкій «он-бордінг» нових розробників до розробки серверної частини, першочергово потрібно створити технічну документацію. Для цього буде використаний Swagger [11] (рис 3.1), який надає красивий, зручний та зрозумілий інтерфейс та дозволяє описувати запити до API, структурувати їх, авторизуватись один раз і на всі запити тощо.

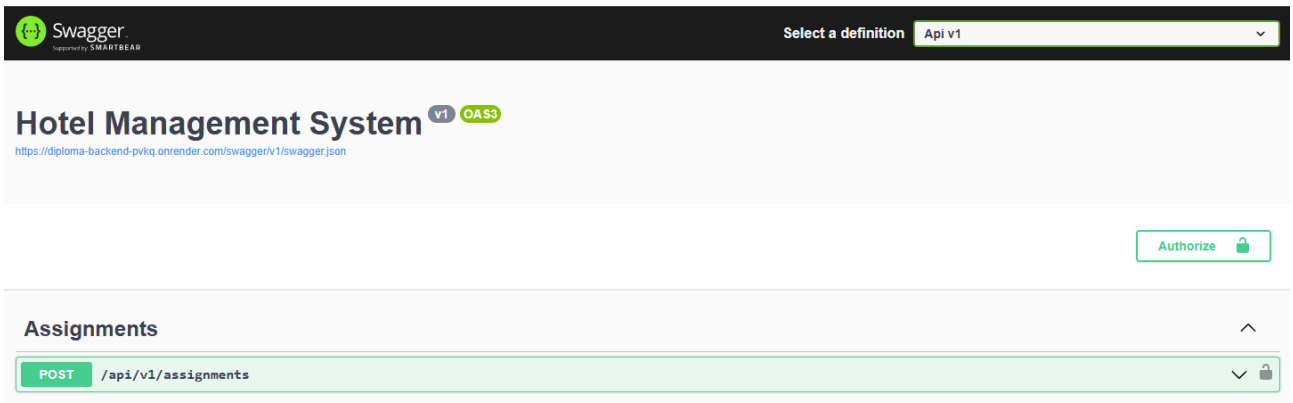


Рисунок 2.1 – Приклад інтерфейсу Swagger (рисунок виконаний самостійно)

Спираючись на сучасні тренди, надійність, швидкість, підтримку новітніх технологій та підтримку розробниками, була обрана база даних PostgreSQL. База даних має бути структурованою та оптимізованою для ефективного зберігання й

обробки інформації (напр., з використанням індексів). Резервне копіювання може бути розглянуто у майбутніх версіях застосунку, що підвищить стійкість до збоїв і мінімізує ризик втрати даних у разі непередбачених ситуацій.

Для сповіщення працівників про надходження нового завдання використовується механізм веб-сокетів та SignalR. SignalR - це безкоштовна програмна бібліотека з відкритим вихідним кодом для Microsoft ASP.NET, що дає змогу серверному коду надсилати асинхронні сповіщення клієнтським веб-застосункам. Бібліотека включає серверні та клієнтські компоненти JavaScript.

Однією з ключових вимог є безпека даних, оскільки система обробляє конфіденційні дані, такі як інформація про співробітників (напр., ПІБ, електронна пошта). Для задовільненя цієї потреби було обрано вбудовану (з версії .NET 7) бібліотеку у ASP.NET Web API, що надає можливості для зручного, швидкого, безпечного управління користувачами та механізми аутентифікації та авторизації. Також присутня підтримка підтвердження за номером телефону, за електронною поштою. Що дуже важливо, також наявна підтримка блокування акаунтів після N спроб увійти та «Security Stamp». Для безпечної передачі даних між клієнтом та сервером використовується протокол HTTPS, а для впровадження механізму сесій користувача та безпеки використовується протокол JWT.

На основі проведеного аналізу та сформульованих вимог можна зробити висновок про необхідність впровадження великої кількості технологій щоб розроблювана система вийшла конкурентоспроможною.

3 АРХІТЕКТУРА ТА ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 UML проектування ПЗ

У розроблюваній системи будуть наявні 5 ролей: менеджер (адміністратор), інвентарний менеджер, звичайні працівники (покоївка, технічний співробітник).

Можливості інвентарного менеджеру наведені на рисунку 4.1. Управління інвентарем:

− створення, видалення, редагування предметів в інвентарі;

− замовлення предметів;

− моніторинг використання інвентарю;

− поповнення запасів інвентарю (отримання предметів, кількість яких менше мінімально дозволеної і замовлення їх);

− генерація звітів: найуживаніший предмет та історія змінення кількості предмету на складі;

− перегляд власного профілю та його редагування.

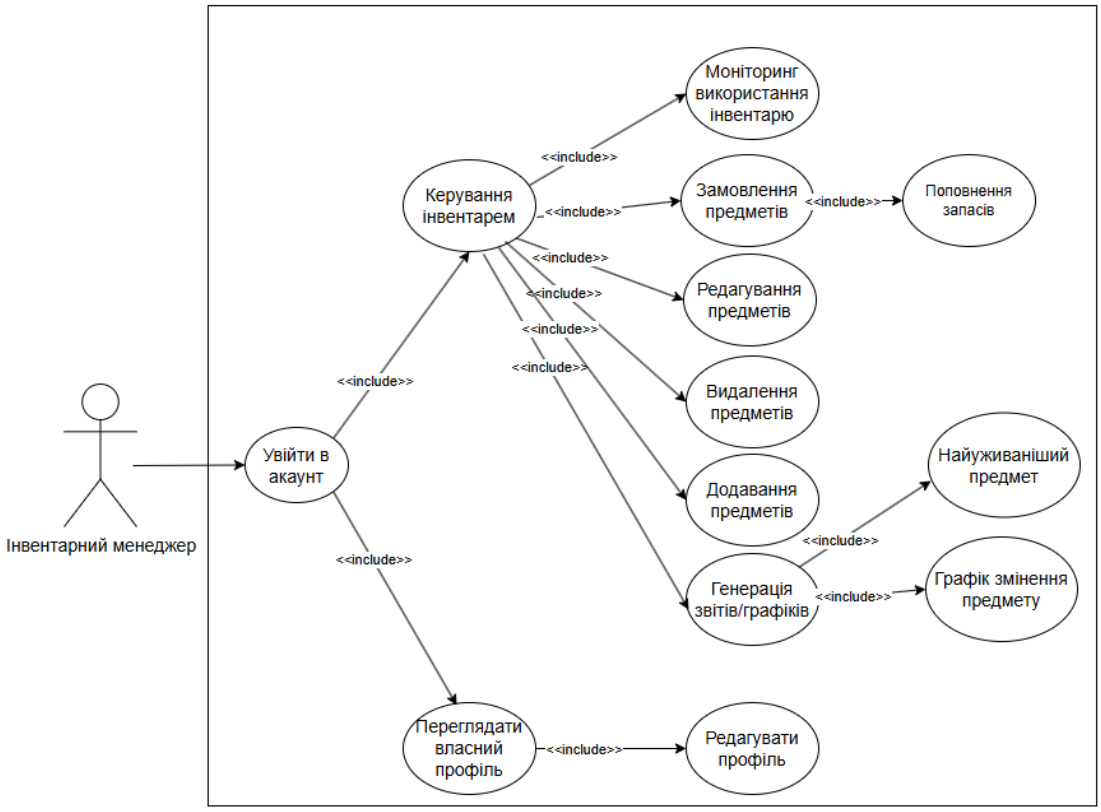


Рисунок 3.1 – Use Case – Інвентарний менеджер (рисунок виконаний самостійно)

Можливості та права менеджеру наведені на рисунку 4.2.

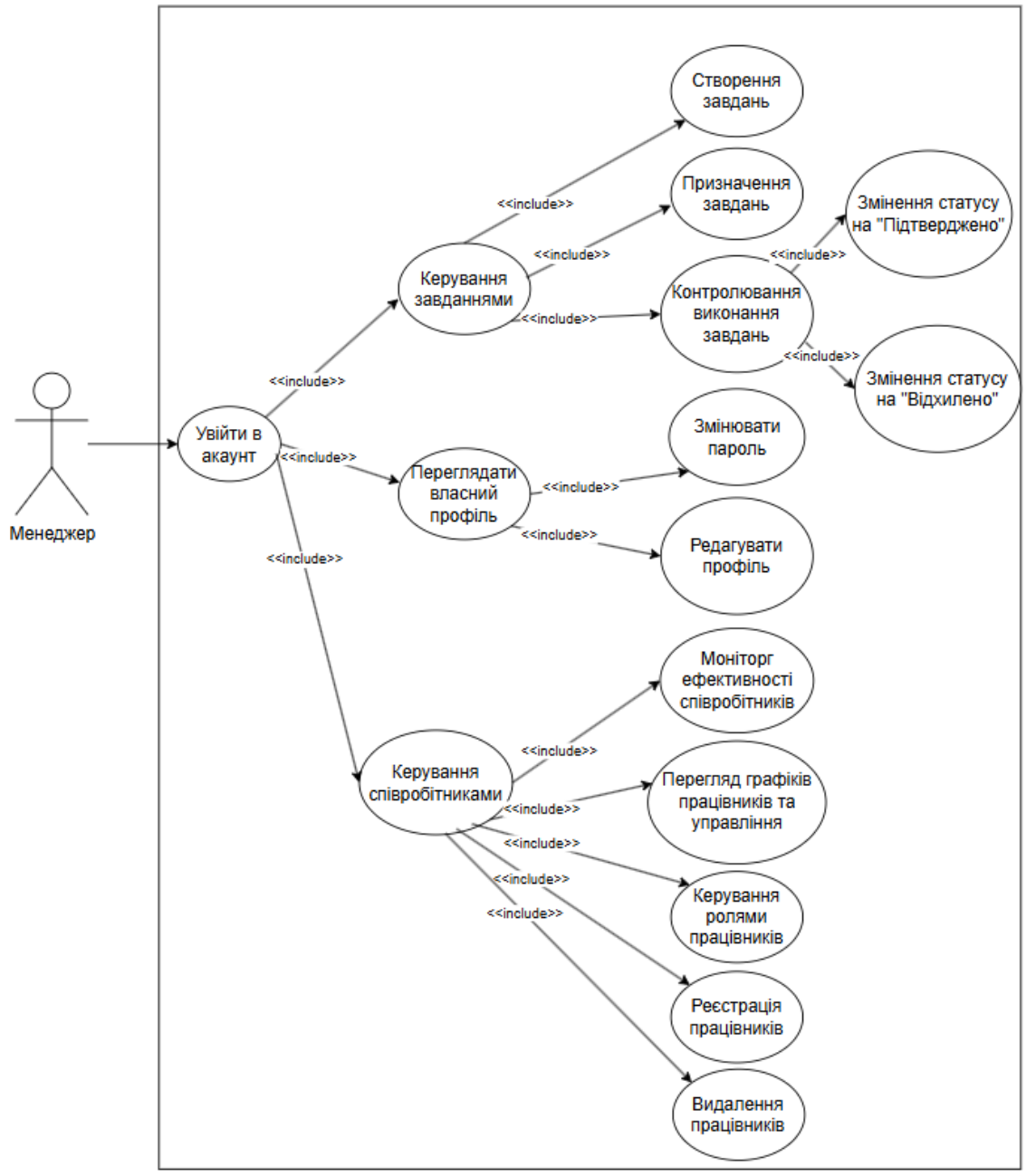


Рисунок 3.2 – Use Case діаграма – Менеджер (рисунок виконаний самостійно)

Загальні взаємодії з працівниками та завданнями:

− створення, видалення, редагування завдань для працівників;

− призначення завдань для працівників;

− контролювання виконання завдань працівниками, змінення їх стану;

− змінення паролю, редагування імені, прізвища тощо;

− моніторинг ефективності співробітників за допомогою графіку;

− перегляд та управління графіками працівників;

− створення, видалення акаунтів співробітників;

− керування доступом та ролями.

Можливості співробітників (технічного працівника та покоївки) наведені на рисунку 4.3. Взаємодія з інвентарем та завданнями:

− змінення статусів завдань (Not Accepted → In Progress → Completed);

− моніторинг власних завдань;

− отримання сповіщень про нові завдання;

− взяти та повернути предмет на склад;

− змінення паролю;

− змінення інформації;

− отримання інформації про себе.

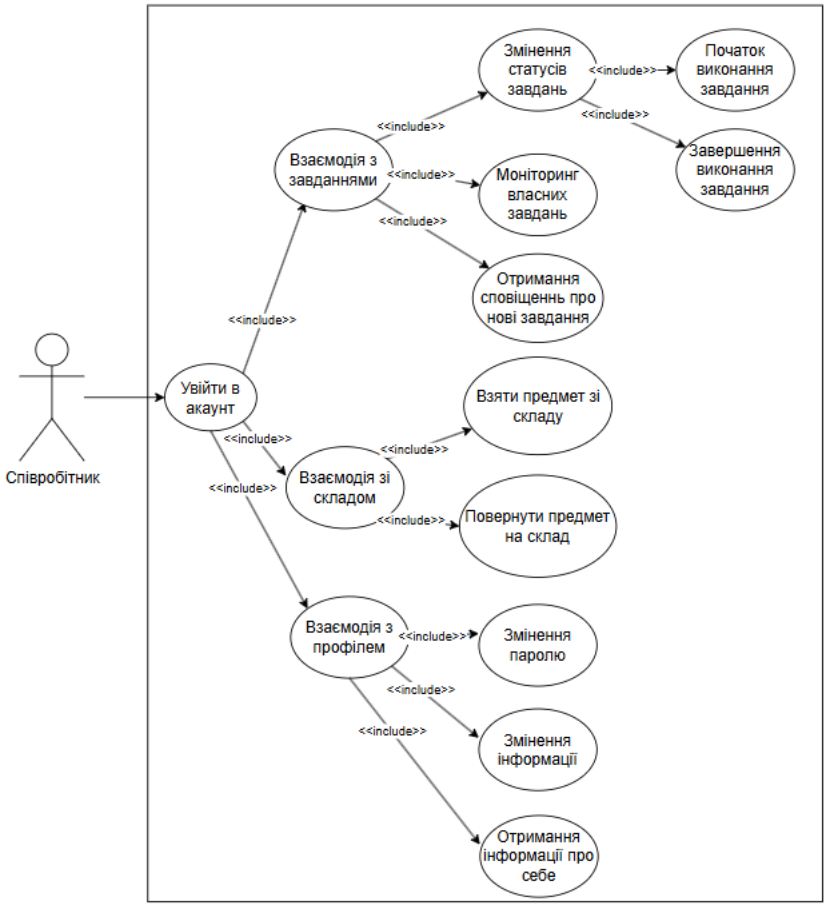


Рисунок 3.3 – Use Case – Співробітник (рисунок виконаний самостійно)

Таким чином, були визначені ролі, які будуть наявні у розроблюваній системі.

3.2 Проєктування архітектури ПЗ

Для розробки серверної частини програмної системи управління персоналом та інвентарем для готельного бізнесу було обрано архітектуру Onion (Onion Architecture), яка є сучасним підходом до проєктування програмних систем, що орієнтується на високий рівень ізоляції бізнес-логіки від зовнішніх залежностей.

Вибір саме цієї архітектури зумовлений її здатністю забезпечити максимальну гнучкість, підтримуваність та розширюваність проєкту. Основною перевагою Onion Architecture є суворе дотримання принципу інверсії залежностей: внутрішні шари не мають залежностей від зовнішніх, тоді як зовнішні шари можуть залежати від внутрішніх. Це дозволяє змінювати інфраструктурні або презентаційні компоненти без ризику порушити цілісність бізнес-логіки.

Ця модель складається з кількох концентричних шарів. Domain - центральний шар, який містить сутності та бізнес-правила, є незалежним від технологій і зовнішніх змін. Application визначає сервіси й інтерфейси, які описують сценарії використання системи. Infrastructure реалізує зовнішні сервіси, наприклад, інтеграцію з поштою чи файлами. Persistence забезпечує доступ до бази даних і реалізує відповідні інтерфейси з Application. Нарешті, Presentation відповідає за взаємодію з користувачем, використовуючи сервіси Application для виконання бізнес-операцій.

Такий підхід дозволяє розробникам зосередитись на бізнес-логіці незалежно від технічних деталей реалізації. Крім того, кожен шар може бути протестований окремо, що значно підвищує якість та надійність програмного забезпечення. Завдяки чіткому розмежуванню обов’язків і залежностей, система стає легшою для супроводу, розширення та захищеною від змін у зовнішньому середовищі. Архітектуру реалізовано відповідно до схеми, наведеної на рисунку 4.4.

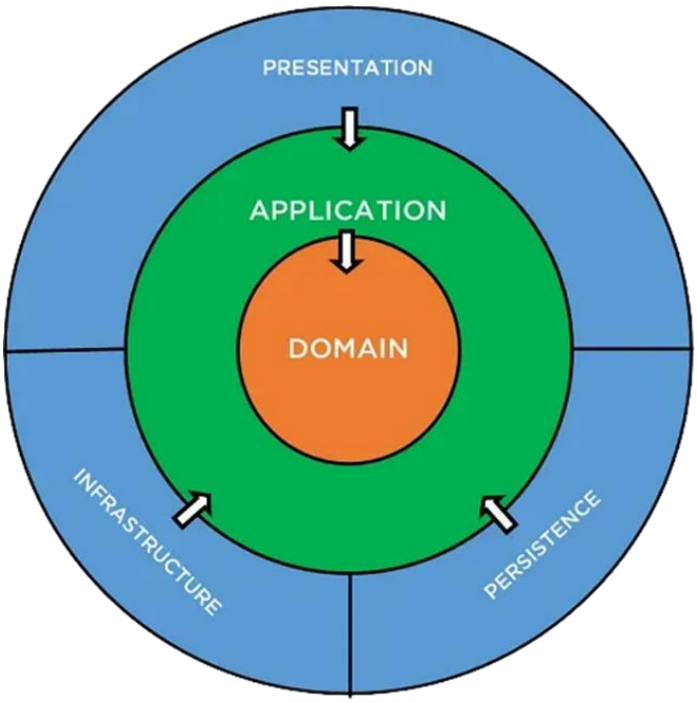


Рисунок 3.4 – Архітектура системи (рисунок виконаний самостійно)

У розроблюваному проєкті ключову роль відіграє шар взаємодії з базою даних — PERSISTENCE, який відповідає за збереження та отримання інформації. Він відокремлює логіку роботи з даними від інших частин системи, зокрема від бізнес-логіки, завдяки чому забезпечується чітка модульність та гнучкість при розширенні функціональності.

Для реалізації цього шару застосовується шаблон проєктування Repository, що дозволяє створити узагальнений інтерфейс для взаємодії з джерелами даних, ізолюючи зовнішні системи зберігання від внутрішніх процесів обробки. Репозиторії слугують проміжною ланкою між доменними сутностями (DOMAIN) та механізмами фізичного зберігання, надаючи уніфікований набір методів для базових CRUD-операцій (створення, читання, оновлення, видалення). Такий підхід дозволяє будувати систему, що легко тестується та підтримується.

В основі взаємодії з базою даних використовується ORM-фреймворк Entity Framework Core (EF Core), який дає змогу здійснювати роботу з реляційними базами даних на основі об’єктно-орієнтованого підходу. Це дозволяє автоматизувати багато рутинних операцій, зокрема побудову запитів SQL,

підтримку реляцій між таблицями, керування транзакціями тощо. Застосування EF Core суттєво знижує складність реалізації та пришвидшує розробку.

На рівні APPLICATION реалізовано бізнес-логіку системи, що відповідає за опрацювання користувацьких запитів і реалізацію сценаріїв використання, які поєднують доменні правила (DOMAIN) з технічними аспектами. Тут відбувається валідація вхідних даних, координація взаємодії між об'єктами та запуск відповідних дій. Для впорядкування комунікації між об’єктами застосовано патерн Mediator, що дозволяє централізувати керування запитами та командами, уникнувши надмірної залежності між компонентами.

До прикладу, за допомогою MediatR (реалізації цього патерну), усі команди та запити маршрутизуються через єдиний посередник, що сприяє чистоті архітектури та дотриманню принципу розділення відповідальностей. Шар DOMAIN у структурі проєкту виконує функцію основи – тут знаходяться головні бізнес-об’єкти та правила, які описують поведінку системи незалежно від технічної реалізації. Цей рівень не має залежностей від інфраструктурних чи прикладних компонентів, що дозволяє забезпечити максимальну стабільність і повторне використання моделей.

На PRESENTATION-шарі реалізується взаємодія з користувачем через HTTP API, побудований на основі REST-принципів. Контролери виступають точкою входу в систему, приймаючи HTTP-запити, які потім делегуються в Application-шар. REST-архітектура забезпечує стандартизований підхід до керування ресурсами за допомогою унікальних URI та методів HTTP (GET, POST, PUT, DELETE), що робить інтерфейс інтуїтивно зрозумілим і добре масштабованим.

Контролери при цьому виконують виключно роль посередника між зовнішніми клієнтами та внутрішньою логікою, не містячи бізнес-обробки. Це дозволяє підтримувати код чистим і легким для змін.

INFRASTRUCTURE-шар відповідає за підтримку технічної бази проєкту – сюди входять налаштування зовнішніх бібліотек, інтеграції з сторонніми сервісами та інші низькорівневі залежності. Наприклад, тут реалізуються класи для

надсилання електронної пошти, логування, роботи з файловою системою тощо. Це дозволяє залишити інші шари "чистими" від деталей реалізації.

Розгортання системи виконується на хмарній платформі Render, яка забезпечує автоматизацію розгортання, моніторинг та масштабування без зайвих витрат часу. Render підтримує CI/CD, дозволяючи безперервно оновлювати додаток та зберігати стабільність роботи. Завдяки цій платформі забезпечується висока доступність, зручність розгортання та економічна ефективність у порівнянні з традиційною інфраструктурою.

У перспективі можливе впровадження Docker або перехід до мікросервісної архітектури, що сприятиме реалізації стратегії безперервного розвитку та модернізації системи. Застосування контейнеризації та мікросервісного підходу дозволить значно спростити оновлення програмного забезпечення, забезпечити ефективне масштабування окремих модулів та оптимізувати процеси CI/CD (неперервної інтеграції та розгортання).

Діаграма розгортання наведена на рисунку 4.5.

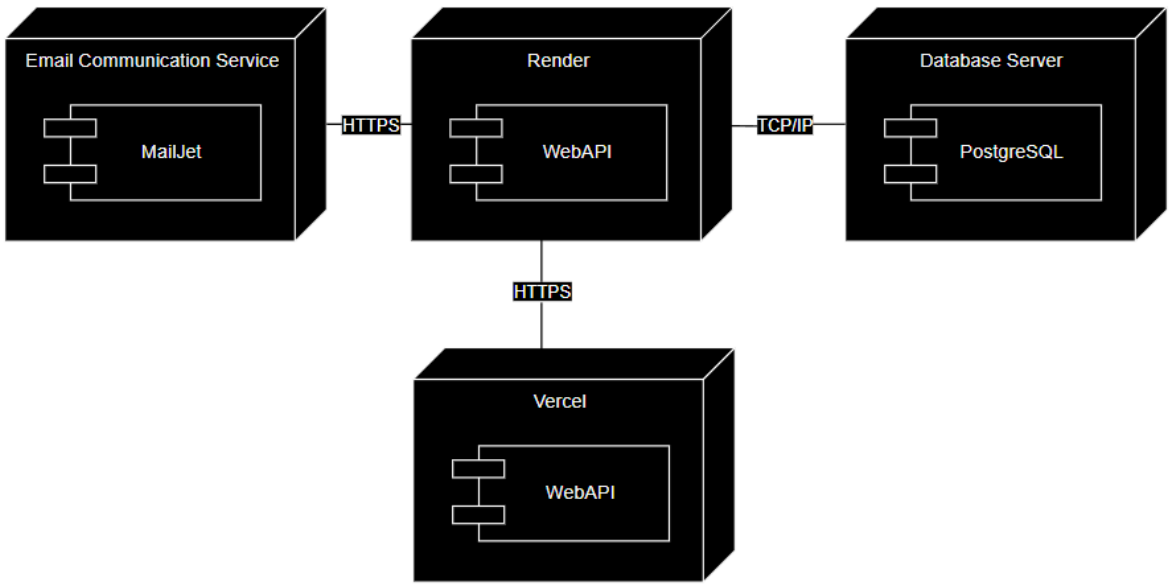


Рисунок 3.5 – Діаграма розгортання (рисунок виконаний самостійно)

Таким чином, були визначені спосіб розгортання та архітектура ПЗ.

3.3 Проєктування структури зберігання даних

Під час створення архітектури для збереження даних у програмній системі важливою складовою є ретельний вибір механізму управління базами даних. Для забезпечення стабільної, захищеної та ефективної роботи з даними було прийнято рішення використовувати PostgreSQL, яка є потужною об'єктно-реляційною СУБД. Її вибір зумовлений високою швидкодією, здатністю працювати з комплексними запитами, підтримкою масштабованості та розширеними можливостями безпеки.

Система PostgreSQL вже багато років використовується в різних сферах і довела свою ефективність у критично важливих середовищах. Завдяки своїй об'єктно-реляційній природі вона чудово підходить для реалізації систем, де особливу роль відіграє безпечна обробка транзакцій, наприклад у фінансових застосунках. Це робить PostgreSQL доречною для використання в готельних, де кожна транзакція має бути точно зафіксована без збоїв, як і у будь-яких інших системах. Завдяки дотриманню принципів ACID (атомарність, узгодженість, ізольованість, стійкість), ця СУБД гарантує високий рівень цілісності й захищеності даних.

Ще однією суттєвою перевагою PostgreSQL є її розвинена система безпеки, яка особливо важлива для установ, що працюють із конфіденційною інформацією — такою, як особисті дані, відомості графік роботи. Інструменти шифрування та механізми захищених з’єднань дозволяють ефективно протидіяти несанкціонованому доступу.

З точки зору масштабованості PostgreSQL здатна підтримувати як вертикальне, так і горизонтальне розширення, що дає змогу гнучко адаптуватися до зростання обсягів даних і навантаження на систему. Це критично важливо для готелів, які прагнуть збільшувати кількість філіалів без втрати стабільності роботи системи.

Для розміщення бази даних обрано хмарну платформу Neon, яка спеціалізується на хостингу PostgreSQL і дозволяє ефективно організувати розгортання, масштабування та адміністрування бази даних. Neon пропонує гнучке

керування ресурсами, підтримку автоматичного масштабування, моментальні знімки бази та швидке відновлення, що суттєво підвищує загальну надійність інфраструктури. Вбудована інтеграція з CI/CD-процесами та сучасними DevOps-інструментами робить цю платформу зручною для швидкого впровадження змін у систему.

На відміну від великих хмарних провайдерів, Neon пропонує оптимізовані умови саме для роботи з PostgreSQL, що дозволяє знизити витрати на хостинг без втрати якості обслуговування. Також платформа надає засоби моніторингу, керування правами доступу та підвищений рівень безпеки, що критично важливо для організацій, які працюють з конфіденційною або чутливою інформацією.

Додатково, можливість реалізації реплікації даних у Neon сприяє підвищенню надійності системи: дублювання інформації на резервні вузли дозволяє мінімізувати ризики втрати даних та покращує обробку запитів завдяки розподілу навантаження. Логічна модель сутностей системи наведена на рисунку 3.6.

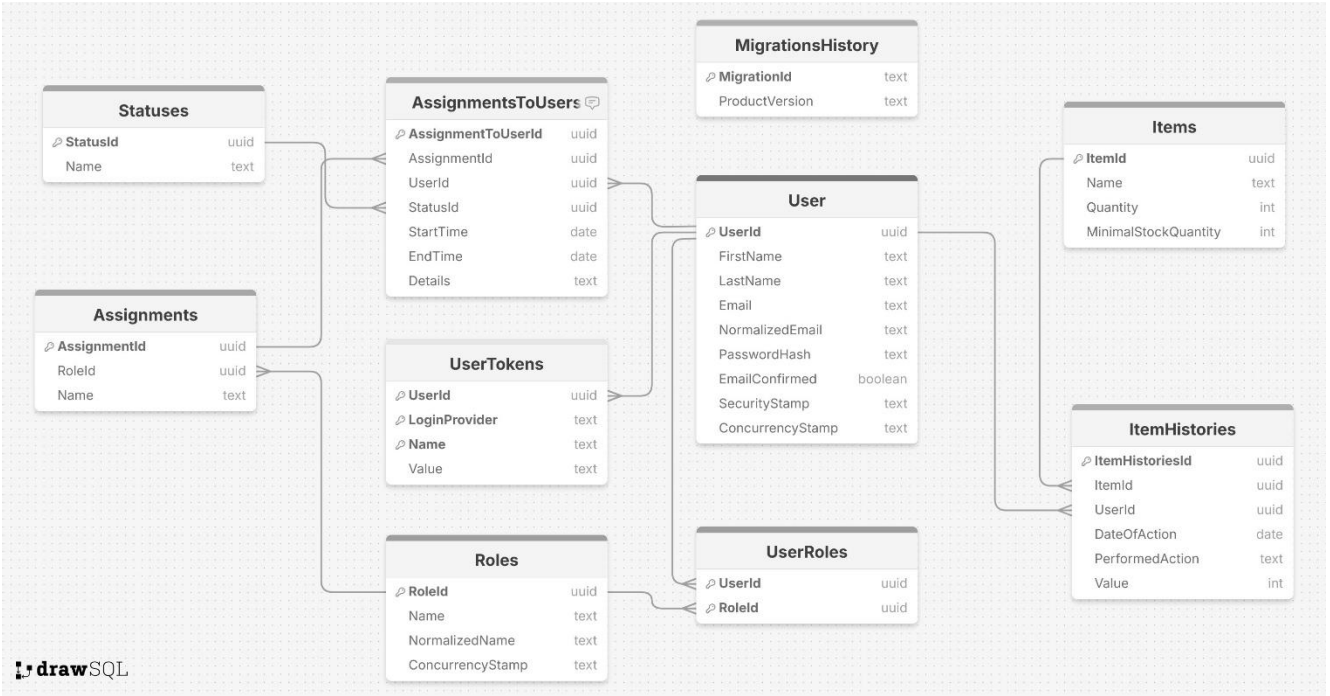


Рисунок 3.6 – Логічна модель БД (рисунок виконаний самостійно)

Таблиця «User» - одна з центральних сутностей у БД. Їм є будь-який користувач системи (працівник готелю), тобто, незареєстрованих у системі користувачей не може бути та вони ніяк далі не могли би взаємодіяти з БД.

У таблиці «Assignments» зберігаються заздалегідь визначені базові завдання (напр., «Прибрати кімнату») для того щоб не писати їх вручну кожен раз при створюванні завдання менеджером. До того ж, одне завдання може бути виконано лише однією роллю (табл. «Roles». Наприклад, прибрати кімнату може тільки покоївка).

«AssignmentsToUsers», тобто, завдання працівників, можуть мати статуси (Not Accepted → In Progress → Completed → Approved/Rejected). Статус завдань може пересуватись лише у напрямку зліва направо. Спочатку завдання має статус «Not Accepted». «In Progress» та «Completed» змінює працівник, а, чи прийняти, чи відхилити завдання, вирішує менеджер.

Можливість мати одному користувачу багато ролей (табл. «UserRoles») існує для підтримки майбутніх змін, які, можливо, будуть потребувати цієї можливості.

У базі даних наявні наступні зв’язки між сутностями:

− типи Assignments (Призначення) та AssignmentsToUsers (Призначення користувача) мають відношення «один-до-багатьох», так як одне завдання може бути призначено багатьом співробітникам;

− типи Statuses (Статуси) та AssignmentsToUsers (Призначення користувача) мають відношення «один-до-багатьох», так як один статус (напр., «Зроблено») може бути у багатьох завдань;

− типи Roles (Ролі) та Assignments (Призначення) мають відношення «один-до-багатьох», так як у однієї ролі може бути багато завдань;

− типи Roles (Ролі) та UserRoles (Ролі користувачів) мають відношення «один-до-багатьох», так як одна роль може бути у багатьох користувачів;

− типи Users (Користувачі) та AssignmentsToUsers (Призначення користувача) мають відношення «один-до-багатьох», так як у одного користувача може бути багато завдань (призначень);

− типи Users (Користувачі) та UserTokens (Токени користувача) мають відношення «один-до-багатьох», так як у одного користувача може бути багато токенів (напр., токен для підтвердження пошти);

− типи Users (Користувачі) та UserRoles (Ролі користувачів) мають відношення «один-до-багатьох», так як одна роль може бути у багатьох користувачів;

− типи Users (Користувачі) та ItemHistories (Історія взаємодії з предметами) мають відношення «один-до-багатьох», так як один користувач може взаємодіяти з багатьма предметами та, відповідно, залишати багато записів у таблиці;

− типи Items (Предмети) та ItemHistories (Історія взаємодії з предметами) мають відношення «один-до-багатьох», так як про один предмет може бути багато записів про взаємодію з ним.

4 ОПИС ПРИЙНЯТИХ РІШЕНЬ

4.1 Робота з базами даних та даними

Зберігання даних серверної частини системи побудоване на основі сучасних технологій: СУБД PostgreSQL, Entity Framework Core (EF Core) для роботи з даними. У цьому розділі детально розглядаються аспекти роботи з базою даних, зокрема проектування її структури, організація операцій з даними, забезпечення продуктивності, безпеки та управління міграціями. Особливу увагу приділено архітектурним рішенням, які забезпечують ефективну інтеграцію бази даних із серверною логікою.

Для реалізації бази даних обрано PostgreSQL – реляційну СУБД із відкритим кодом, яка відзначається високою надійністю та продуктивністю. Вибір PostgreSQL обґрунтований кількома факторами: підтримкою складних запитів, можливістю масштабування, наявністю механізмів забезпечення цілісності даних та широким поширенням у сучасних веб-додатках. Ця СУБД забезпечує стабільну роботу навіть при значних навантаженнях, що є важливим для готельного бізнесу, де обсяги даних можуть швидко зростати.

Взаємодія з базою даних здійснюється через Entity Framework Core – ORM-фреймворк, який дозволяє працювати з даними на рівні об'єктів, автоматично перетворюючи LINQ-запити у SQL-команди. EF Core спрощує розробку, забезпечує автоматичне управління транзакціями та підтримує міграції, що полегшує оновлення структури бази даних. На рисунку 5.1 наведений приклад LINQ-запиту з використанням паттерну репозиторіїв, який допомагає реалізовувати принципі SOLID, а саме допомагає не повторювати один і той самій код виконання одних і тих самих запитів по декілька разів.

```
public int Count(Expression<Func<T, bool>>? predicate = null)
{
    IQueryable<T> set = Context.Set<T>().AsNoTracking();
    if (predicate is not null)
        set = set.Where(predicate);

    return set.Count();
}
```

Рисунок 4.1 – LINQ-запити (рисунок виконаний самостійно)

Під час написання API, наприклад, може знадобитись у десятках місць у коді десятки разів отримати сутність з БД. Код для цього усі рази один і той самий. А Generic-репозиторій дозволяє винести однаковий для всіх сутностей код (всі сутності мають мати змогу бути доданими, видаленими, редагованими або прочитаними) ще вище, позбавляючи розробника від написання так званого boilerplate коду. Усі інші, індивідуальні репозиторії для кожної сутності, успадковують generic репозиторій і, у свою чергу, можуть визначати особисті методи за потребами конкретної сутності. Приклад «універсального» репозиторію наведений на рисунку 5.2. Як видно, він може робити усі базові функції, які є загальними для кожної таблиці в базі даних. До того ж, методи читання з БД приймають аргументи, які дозволяють витягувати з БД дані «пачками», так звана «пагінація», сортувати видачу або писати умови за якими шукати дані.

Структура бази даних розроблена з урахуванням потреб готельного бізнесу, зокрема необхідності обліку персоналу, інвентарю, розподілу завдань та історії операцій. На основі наданої ER-діаграми у пункті 4.3 можна виділити основні сутності та зв'язки між ними.

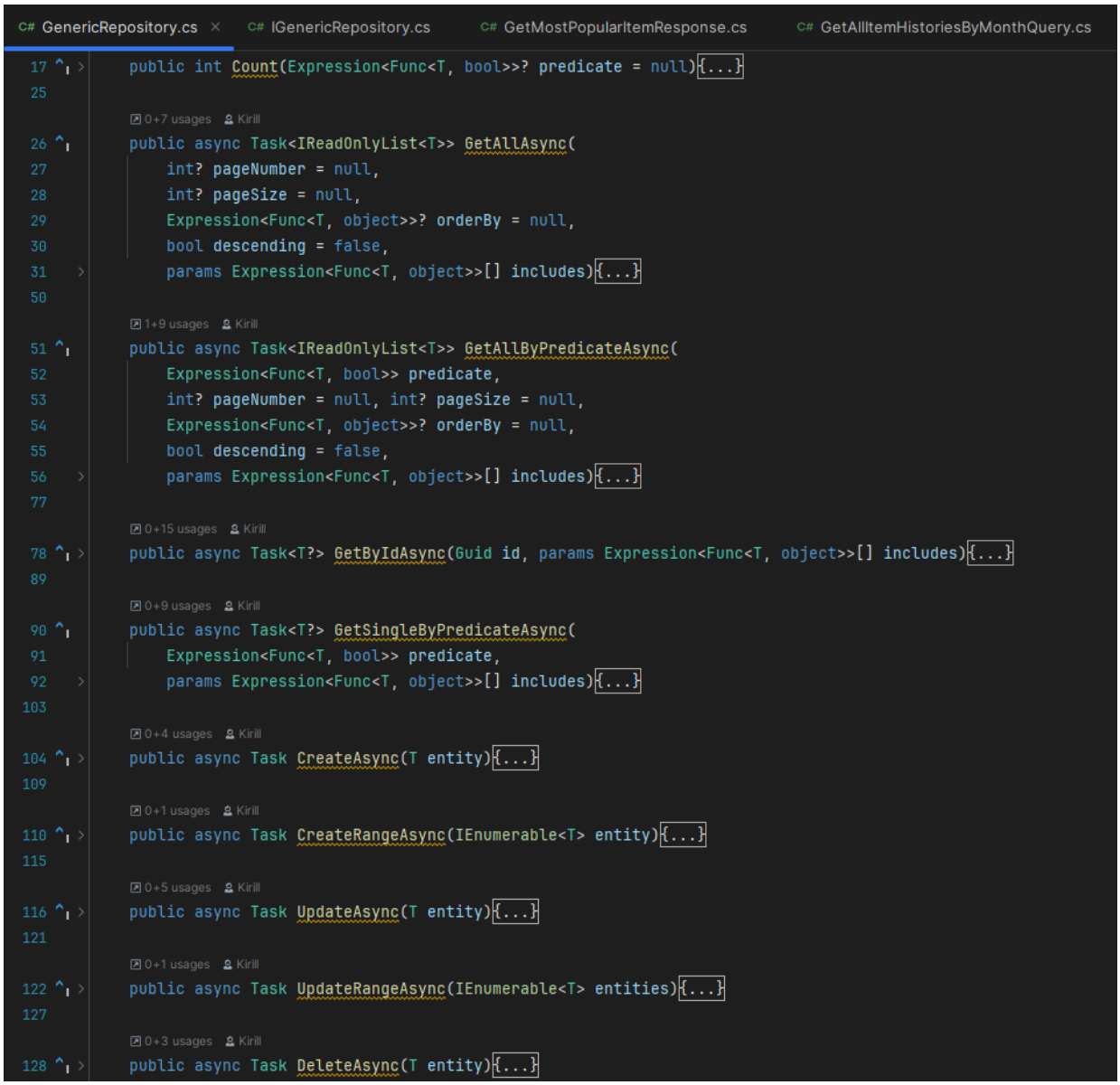


Рисунок 4.2 – Generic репозиторій (рисунок виконаний самостійно)

Основні таблиці бази даних включають:

Users (користувачі), що містить інформацію про персонал готелю, зокрема ідентифікатор (UserId), ім'я (FirstName, LastName), електронну пошту (NormalizedEmail), пароль (PasswordHash) та інші атрибути.

Items (інвентар) зберігає дані про предмети, такі як ідентифікатор (ItemId), назва (Name) та мінімальна кількість на складі (MinimalStockQuantity).

Assignments (завдання) описує завдання, що призначаються персоналу, з атрибутами, такими як ідентифікатор (AssignmentId), назва (Name) та зв'язок із роллю (RoleId).

AssignmentsToUsers (призначення завдань користувачам) – проміжна таблиця для зв'язку "багато-до-багатьох" між завданнями та користувачами, містить ідентифікатори завдання (AssignmentId) і користувача (UserId), а також часові рамки (StartTime, EndTime).

ItemHistories (історія використання інвентарю) фіксує операції з предметами, включаючи ідентифікатор (ItemHistoriesId), зв'язок із предметом (ItemId), користувачем (UserId), дату дії (DateAction) та тип операції (PerformedAction, Value).

Roles (ролі) визначає ролі користувача з атрибутами, такими як ідентифікатор (RoleId) і назва (Name).

UserRoles (зв'язок користувачів із ролями) – проміжна таблиця для зв'язку "багато-до-багатьох" між користувачами та ролями.

Statuses (статуси) містить перелік можливих статусів для завдань.

UserTokens (токени користувачів) зберігає токени для аутентифікації.

MigrationHistory (історія міграцій) – системна таблиця для відстеження змін у структурі бази даних. При зміні сутностей у коді, ці зміни відображаються також на моделі в БД.

Зв'язки між сутностями реалізовано з урахуванням реляційної моделі. Наприклад, зв'язок між таблицями Users і Roles організовано через проміжну таблицю UserRoles, що дозволяє одному користувачеві мати кілька ролей, а одній ролі відповідати кільком користувачам (зв'язок "багато-до-багатьох"). Аналогічно, зв'язок між Users і Assignments реалізовано через таблицю AssignmentsToUsers. Таблиця ItemHistories пов'язана з Users і Items через зовнішні ключі UserId та ItemId, що дозволяє відстежувати, хто і коли виконував операції з інвентарем.

База даних спроектована з урахуванням принципів нормалізації, що дозволяє уникнути аномалій при операціях із даними та забезпечити ефективне зберігання. Проведений аналіз показує, що база даних відповідає вимогам третьої нормальної форми (3NF).

Перша нормальна форма (1NF) забезпечена, оскільки усі таблиці мають первинні ключі (наприклад, UserId, ItemId), а значення атрибутів є атомарними (немає складених значень або списків у полях).

Друга нормальна форма (2NF) виконується, адже усі неключові атрибути залежать від повного первинного ключа. Наприклад, у таблиці AssignmentsToUsers атрибути StartTime і EndTime залежать від комбінації AssignmentId і UserId.

Третя нормальна форма (3NF) досягнута, оскільки неключові атрибути не залежать від інших неключових атрибутів. Наприклад, у таблиці Users поле NormalizedEmail залежить виключно від UserId, а не від інших полів, таких як PasswordHash.

Нормалізація дозволяє зменшити дублювання даних і забезпечити їхню цілісність. Наприклад, використання проміжних таблиць для зв'язків "багато-до-багатьох" усуває надмірність, а наявність зовнішніх ключів гарантує узгодженість між пов'язаними записами.

Операції з даними в системі включають створення, читання, оновлення та видалення записів (CRUD), а також виконання запитів для звітності. Наприклад, додавання нового завдання в таблицю Assignments автоматично пов'язується з користувачами через таблицю AssignmentsToUsers, а списання інвентарю фіксується в таблиці ItemHistories. Для забезпечення швидкості виконання запитів використано індекси. Зокрема, індекси створено для полів, які часто використовуються в умовах пошуку або об'єднання таблиць, таких як UserId у таблиці ItemHistories чи ItemId у таблиці Items.

Entity Framework Core забезпечує зручну роботу з даними завдяки LINQ-запитам, які автоматично перетворюються на SQL. Наприклад, для отримання всіх завдань конкретного користувача можна використати запит на C# (рис. 5.3).

```
var assignments :IReadOnlyList<AssignmentToUser> = await _assignmentToUserRepository // IAssignmentToUserRepository
    .GetAllByPredicateAsync(predicate: p :AssignmentToUser => p.User.Email == request.Email,
    includes: [incl :AssignmentToUser => incl.Assignment.Role, incl :AssignmentToUser => incl.AssignmentToUserStatus]);
```

Рисунок 4.3 – Отримання завдань користувача (рисунок виконаний самостійно)

Для забезпечення цілісності даних використано транзакції, які гарантують атомарність операцій. Наприклад, при списанні інвентарю транзакція включає оновлення кількості предметів у таблиці Items і створення запису в таблиці ItemHistories. Якщо одна з операцій не виконується, транзакція скасовується, що запобігає неузгодженості даних.

Управління контекстами EF Core реалізовано через спеціальний сервіс, який створює новий екземпляр DbContext для кожного запиту. Це дозволяє уникнути конфліктів при паралельній обробці запитів, оскільки EF Core не підтримує одночасне використання одного контексту в кількох потоках. Для цього використано підхід, подібний до паттерну Factory.

Сервіс інтегровано в систему через механізм ін'єкції залежностей ASP.NET Core. Для цього створено метод розширення, який реєструє сервіс як Scoped, тобто новий екземпляр контексту створюється для кожного HTTP-запиту і знищується після його завершення. Це забезпечує ізоляцію запитів і підвищує стабільність системи. Код наведений на рисунку 5.4.

```
builder.Services.AddDbContext<DataContext>( optionsAction: options =>
{
    options.UseNpgsql(configuration.GetConnectionString( name: "BackendDbConnectionString"));
    options.EnableSensitiveDataLogging();
});
```

Рисунок 4.4 – Ін’єкція бази даних (рисунок виконаний самостійно)

Дані для підключення до БД (Connection String) зберігаються у файлі appsettings.json (рис. 5.5), котрий зчитується фреймворком при запуску і дозволяє діставати з нього інформацію через об’єкт Configuration.

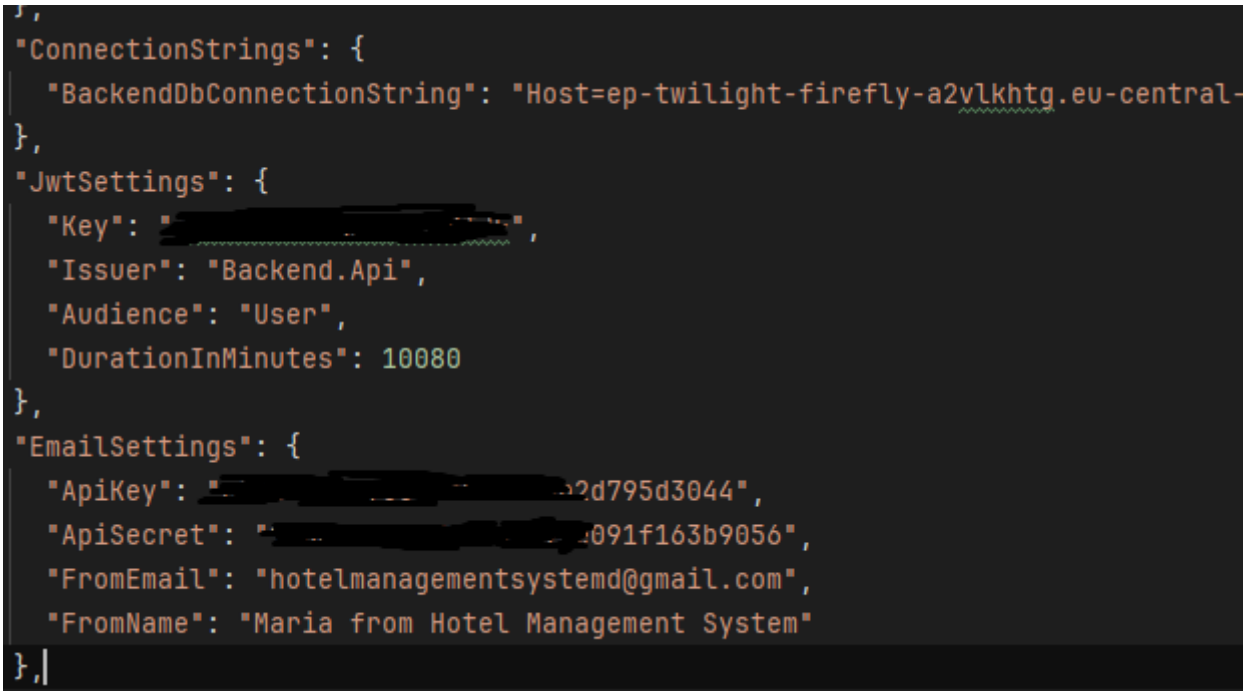


Рисунок 4.5 – Файл appsettings.json (рисунок виконаний самостійно)

Для забезпечення масштабованості системи передбачено можливість як вертикального, так і горизонтального масштабування. Вертикальне масштабування досягається шляхом покращення апаратного забезпечення сервера бази даних, наприклад, збільшення обсягу оперативної пам'яті чи потужності процесора. Горизонтальне масштабування реалізується через розподіл навантаження між кількома екземплярами бази даних. У PostgreSQL для цього можна використовувати реплікацію, наприклад, налаштувавши master-slave архітектуру, де основний сервер (master) обробляє операції запису, а репліки (slaves) – операції читання.

Продуктивність запитів підвищується завдяки індексам і оптимізації LINQ-запитів у EF Core. Наприклад, для звітів про використання інвентарю використано попереднє обчислення даних, які зберігаються в тимчасових таблицях. Це зменшує час виконання запитів і навантаження на базу даних.

Безпека даних є важливим аспектом системи. Для захисту від SQL-ін'єкцій використано параметризовані запити, які генеруються EF Core автоматично. Аутентифікація користувачів реалізується через токени, що зберігаються в таблиці

UserTokens, а авторизація базується на ролях із таблиці Roles. Наприклад, лише користувачі з роллю адміністратора можуть реєструвати нових працівників, тоді як покоївки мають доступ лише до своїх завдань. Чутливі дані, такі як паролі (PasswordHash), зберігаються в зашифрованому вигляді.

Для синхронізації структури бази даних із моделями даних використано міграції EF Core. Міграції дозволяють автоматично оновлювати схему бази даних при зміні сутностей, наприклад, додаванні нового поля до таблиці Items. Процес міграції включає створення міграційних файлів за допомогою команди Add-Migration і їхнє застосування через Update-Database.

Для автоматизації міграцій розроблено сервіс, який перевіряє наявність змін у структурі бази даних під час запуску програми та застосовує необхідні оновлення. Це дозволяє розробникам швидко адаптувати базу даних до змін у коді без ручного втручання.

Використання PostgreSQL, EF Core дозволило створити надійне та продуктивне рішення, яке відповідає сучасним вимогам. Нормалізована структура бази даних, оптимізація запитів через індекси, застосування паттернів проєктування та механізми безпеки забезпечують високу швидкість обробки даних і захист від уязвимостей. У майбутньому систему можна вдосконалити шляхом додавання підтримки кешування.

4.2 Авторизація та аутентифікація

У розробленій програмній системі для управління персоналом та завданнями та інвентарем у готельному бізнесі аутентифікація та авторизація відіграють ключову роль у забезпеченні безпеки та обмеженні доступу до функціоналу. Оскільки система працює через серверну частину, яка надає API для фронтенду, ці механізми реалізовані з урахуванням стандартів RESTful API та сучасних підходів до захисту даних. Аутентифікація гарантує, що користувач є тим, за кого себе видає, а авторизація визначає, які дії він має право виконувати.

Для аутентифікації використано JSON Web Tokens (JWT), які генеруються спеціальним класом JwtSecurityToken.

JWT-токени є стандартом для безпечної передачі даних між клієнтом і сервером у вигляді закодованого JSON-об'єкта. Вони складаються з трьох частин: заголовка, даних і підпису, які кодуються в Base64 і розділяються крапками. Заголовок містить інформацію про тип токена (JWT) і алгоритм шифрування (HMAC-SHA256). У даних зберігаються ідентифікатор користувача (UserId), його роль і термін дії токена. Підпис забезпечує перевірку цілісності токена, використовуючи секретний ключ, відомий лише серверу. Клас JwtSecurityToken відповідає за створення токена: він приймає дані користувача, формує JSON-об'єкт, підписує його та повертає готовий токен у вигляді рядка.

Процес аутентифікації починається з введення користувачем облікових даних – електронної пошти та пароля – через фронтенд. Ці дані передаються на сервер через захищене HTTPS-з'єднання на ендпоінт /api/v1/auth/register. Для аутентифікації використано ASP.NET Core Identity – бібліотеку, яка забезпечує вбудовані механізми управління користувачами. Сервер перевіряє, чи існує користувач із вказаною електронною поштою в таблиці Users, і порівнює введений пароль із збереженим хешем у полі PasswordHash. Якщо дані правильні, JwtGenerator генерує токен, який повертається клієнту у відповіді API. Токен зберігається на стороні клієнта і додається до всіх наступних запитів у заголовку Authorization: Bearer <токен>.

Сервер перевіряє токен при кожному запиті: розкодовує його, перевіряє підпис і термін дії, а також переконується, що токен не був анульований (наприклад, при виході користувача).

Важливою особливістю системи є вимога підтвердження електронної пошти для використання функціоналу. Під час реєстрації співробітника адміністратором адміністратор вводить електронну пошту співробітника, після чого на цю адресу надсилається лист із унікальним посиланням для підтвердження. У базі даних у таблиці Users поле EmailConfirmed має значення false, доки користувач не

підтвердить свою пошту. Без підтвердження користувач не може увійти в систему, що забезпечує додатковий рівень безпеки. Після переходу за посиланням у листі поле EmailConfirmed оновлюється на true, і користувач отримує доступ до системи.

Для шифрування паролів використано стандартні алгоритми, зокрема Argon2 – сучасний метод хешування, який переміг у конкурсі Password Hashing Competition у 2015 році. Argon2 має кілька переваг порівняно з іншими алгоритмами, такими як bcrypt або PBKDF2. Він дозволяє налаштувати три параметри: витрати пам'яті, кількість ітерацій і ступінь паралелізму, що дає змогу адаптувати алгоритм до апаратного забезпечення. Наприклад, збільшення витрат пам'яті ускладнює атаки типу brute force на апаратному рівні, оскільки зловмиснику потрібно більше ресурсів для кожної спроби. У системі Argon2 налаштовано на використання 64 МБ пам'яті, 4 ітерацій і 2 потоків, що забезпечує баланс між безпекою та продуктивністю. Пароль користувача хешується під час реєстрації або зміни пароля, а результат зберігається в полі PasswordHash. Це унеможливлює витік паролів у чистому вигляді навіть у разі компрометації бази даних.

Авторизація в системі базується на ролях, які визначені в таблиці Roles і пов'язані з користувачами через таблицю UserRoles. Система має ієрархію ролей: Адміністратор є найвищим рівнем і має доступ до всіх функцій; Менеджер і Інвентарний менеджер мають обмежений доступ, наприклад, до управління завданнями чи інвентарем відповідно; Покоївка і Технік мають найнижчий рівень доступу, обмежений їхніми завданнями. Зв'язок "багато-до-багатьох" між користувачами та ролями дозволяє гнучко призначати кілька ролей одному користувачеві, якщо це необхідно.

ASP.NET Core надає вбудовану авторизацію через атрибут Authorize, який застосовується до ендпоінтів API. Усі ендпоінти в системі розбиті по ролям, що забезпечує чітке розділення прав доступу. Наприклад, ендпоінт /api/v1/assignments з методом [POST] для призначення нового завдання доступний лише для ролі "Менеджер" і захищений атрибутом [Authorize(Roles = "Manager")].

Перевірка ролей відбувається автоматично: сервер витягує роль із JWT-токена і порівнює її з вимогами ендпоінта. Якщо користувач не має потрібної ролі, повертається помилка 403 Forbidden. Для складніших сценаріїв, таких як змінення статусів завдань, використовується додаткова фільтрація на основі бізнес-логіки. Наприклад, співробітник може змінити статус завдання тільки на «В процесі», тобто, прийняти його на виконання і на «Виконано». А менеджер може змінити його на «Схвалено» або «Відхилено».

Для підвищення безпеки реалізовано захист від атак типу brute force. Якщо користувач 5 разів поспіль вводить неправильний пароль, його обліковий запис тимчасово блокується на 15 хвилин. Цей механізм вбудований у ASP.NET Core Identity і налаштований через політику блокування. Інформація про невдалі спроби входу записується в системний лог, що дозволяє адміністратору відстежувати підозрілу активність.

Усі запити до API передаються через HTTPS, що запобігає перехопленню токенів під час передачі. JWT-токени мають обмежений термін дії, що зменшує ризик їхнього використання в разі крадіжки. При виході користувача з системи токен видаляється з таблиці UserTokens, що анулює його дію, навіть якщо термін дії ще не закінчився. Відсутність refresh-токенів спрощує систему, але вимагає від користувача повторного входу після закінчення терміну дії токена, що може бути незручним для довгих сесій. У майбутньому можна додати підтримку refresh-токенів для покращення користувацького досвіду.

Система також веде журнали аудиту для всіх дій, пов'язаних із аутентифікацією та авторизацією. Наприклад, при кожному вході фіксується час ідентифікатор користувача. При невдалій спробі входу записується причина (неправильний пароль або непідтверджена пошта).

Одним із викликів під час реалізації було забезпечення швидкості перевірки токенів при великій кількості запитів. Для цього використано кешування: після першої перевірки токена його статус зберігається в пам'яті сервера, що зменшує

кількість звернень до бази даних. Кеш оновлюється при видаленні токена, наприклад, при виході користувача з системи.

Для додаткового захисту в майбутньому можна реалізувати двофакторну аутентифікацію (2FA). Наприклад, після введення пароля користувач отримуватиме код підтвердження на пошту, який потрібно ввести для завершення входу. Це особливо корисно для користувачів із роллю адміністратора, які мають доступ до чутливих даних. Також можна додати обмеження доступу за IP-адресами, щоб дозволити вхід лише з певних мереж, наприклад, із внутрішньої мережі готелю.

Аутентифікація та авторизація в системі забезпечують надійний захист даних і чіткий розподіл прав доступу відповідно до ієрархії ролей. Використання JWT-токенів, ASP.NET Core Identity і шифрування паролів через Argon2 дозволяє досягти високого рівня безпеки. Вимога підтвердження електронної пошти додає додатковий бар'єр для зловмисників. У майбутньому систему можна вдосконалити шляхом додавання 2FA, підтримки refresh-токенів і розширення журналів аудиту, що зробить її ще більш захищеною та зручною для користувачів.

5 ТЕСТУВАННЯ РОЗРОБЛЕНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Тестування програмного забезпечення є критично важливим і невід'ємним етапом у процесі створення продукту. Під час розробки кожен учасник команди виконує перевірку функціональності продукту. Розробники, зокрема, зобов'язані оцінити граничні випадки перед передачею додатка на тестування спеціалістам з якості, які, у свою чергу, повинні перевірити всі можливі сценарії взаємодії з елементами функціоналу. Особи, відповідальні за демонстрацію продукту, також мають переконатися, що під час презентації не виникне несправностей.

У контексті розробки додатка, я, як розробник, відповідаю за перевірку коректної роботи створеного ендпоінту та відповідності його результатів очікуванням. Для цього я використовую два ключові інструменти, одним із яких є Swagger. Swagger — це потужна бібліотека, що підтримується більшістю сучасних мов програмування і призначена для документування API. Вона дозволяє чітко описати доступні адреси сервісу, параметри запитів, які вони приймають, та відповіді, які повертають. Крім того, Swagger надає зручний веб-інтерфейс, який забезпечує можливість виконання запитів до сервісу в процесі розробки. Цей інструмент широко застосовується у більшості сучасних проєктів, і його використання є стандартом для веб-розробників.

На прикладі інтерфейсу Swagger, зображеному на рис. 5.1, можна побачити декілька доступних ендпоінтів сервісу.

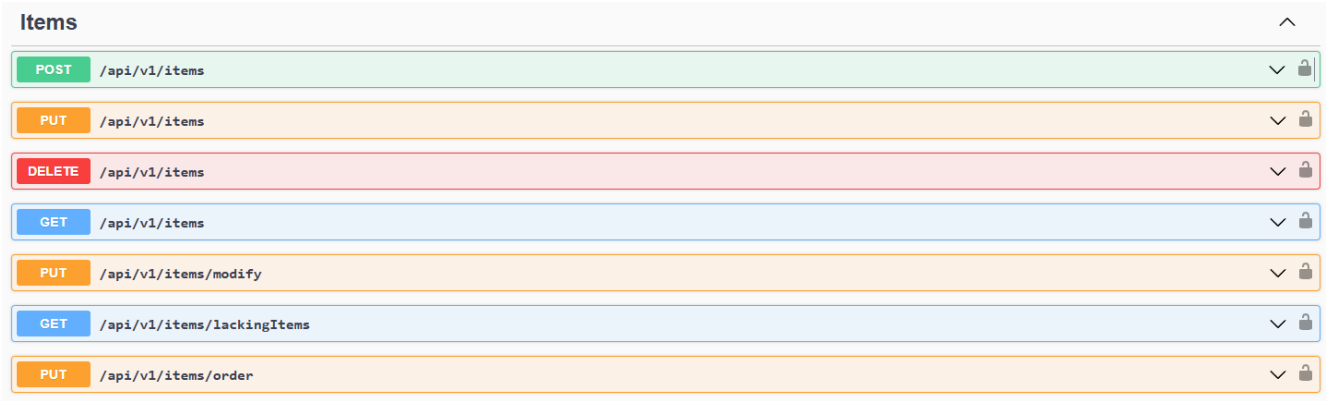


Рисунок 5.1 – Вигляд частини ендпоінтів (рисунок виконано самостійно)

Кожен ендпоінт супроводжується інформацією про метод запиту (наприклад, GET, POST), його URL-адресою та структурою очікуваного тіла запиту/відповіді (рис. 5.2). У верхній частині інтерфейсу доступна опція налаштування авторизації, що дозволяє додати токен, який автоматично включатиметься до заголовків усіх запитів. Крім того, Swagger забезпечує детальний опис ендпоінтів, що дає змогу автоматично генерувати клієнтський код для популярних мов програмування, спрощуючи інтеграцію з сервісом. Завдяки простоті інтеграції та зручності використання, Swagger є незамінним інструментом для швидкого та ефективного тестування розроблених ендпоінтів без потреби у додаткових програмах.

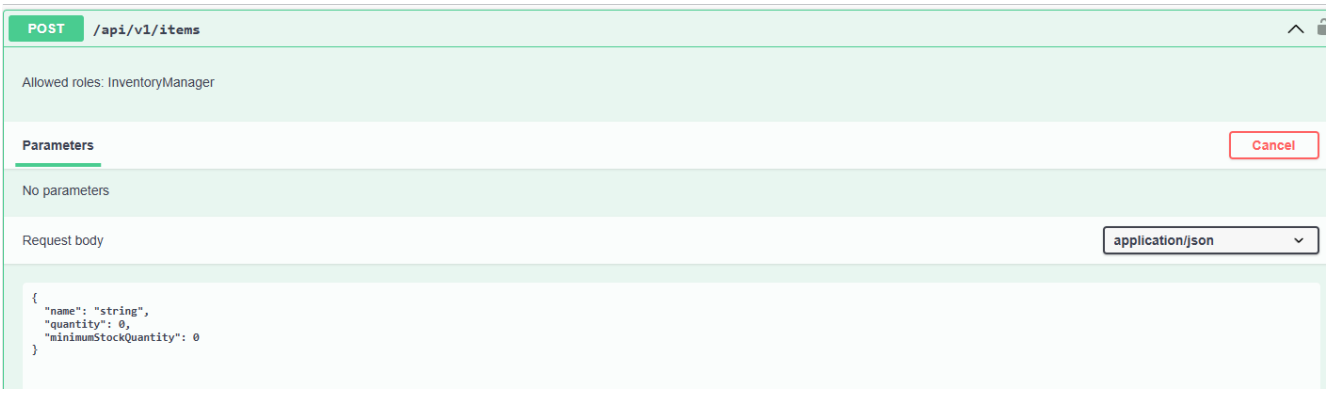


Рисунок 5.2 – Вигляд окремої кінцевої точки (рисунок виконано самостійно)

Далі буде наведено декілька тест-кейсів (табл. 5.1-5.3), які були виконані способом мануального тестування через Swagger.

Таблиця 5.1 – Тест-кейс 1 (таблиця виконана самостійно)

Ім'я	Готвянський К. П.
Дата	21.05.2025
Версія продукту	1.0
Назва тесту	Авторизація
Вхідні дані	Логін: manager@localhost.com Пароль: P@ssword1
Кроки	1. Перейти до інтерфейсу свагеру 2. Обрати ендпоінт «**POST/api/v1/auth/login»** 3. Ввести вхідні дані 4. Виконати запит

Кінець таблиці 5.1

Очікуваний результат	Успішно (JWT токен отримано)
Фактичний результат	Успішно (JWT токен отримано)
Відгук	Успішно пройдений тест

Таблиця 5.2 – Тест-кейс 2 (таблиця виконана самостійно)

Ім'я	Готвянський К. П.
Дата	21.05.2025
Версія продукту	1.0
Назва тесту	Отримати завдання користувачів (перша сторінка – 20 ел.)
Вхідні дані	Номер сторінки: 1 Кількість елементів - 20
Кроки	1. Перейти до інтерфейсу свагеру 2. Авторизуватись як менеджер 3. Обрати ендпоінт «**GET/api/v1/assignmentsToUsers»** 4. Ввести вхідні дані 5. Виконати запит
Очікуваний результат	Отримано завдання користувачів (макс. 20 ел.)
Фактичний результат	Отримано завдання користувачів (макс. 20 ел.)
Відгук	Успішно пройдений тест

Таблиця 5.3 – Тест-кейс 3 (таблиця виконана самостійно)

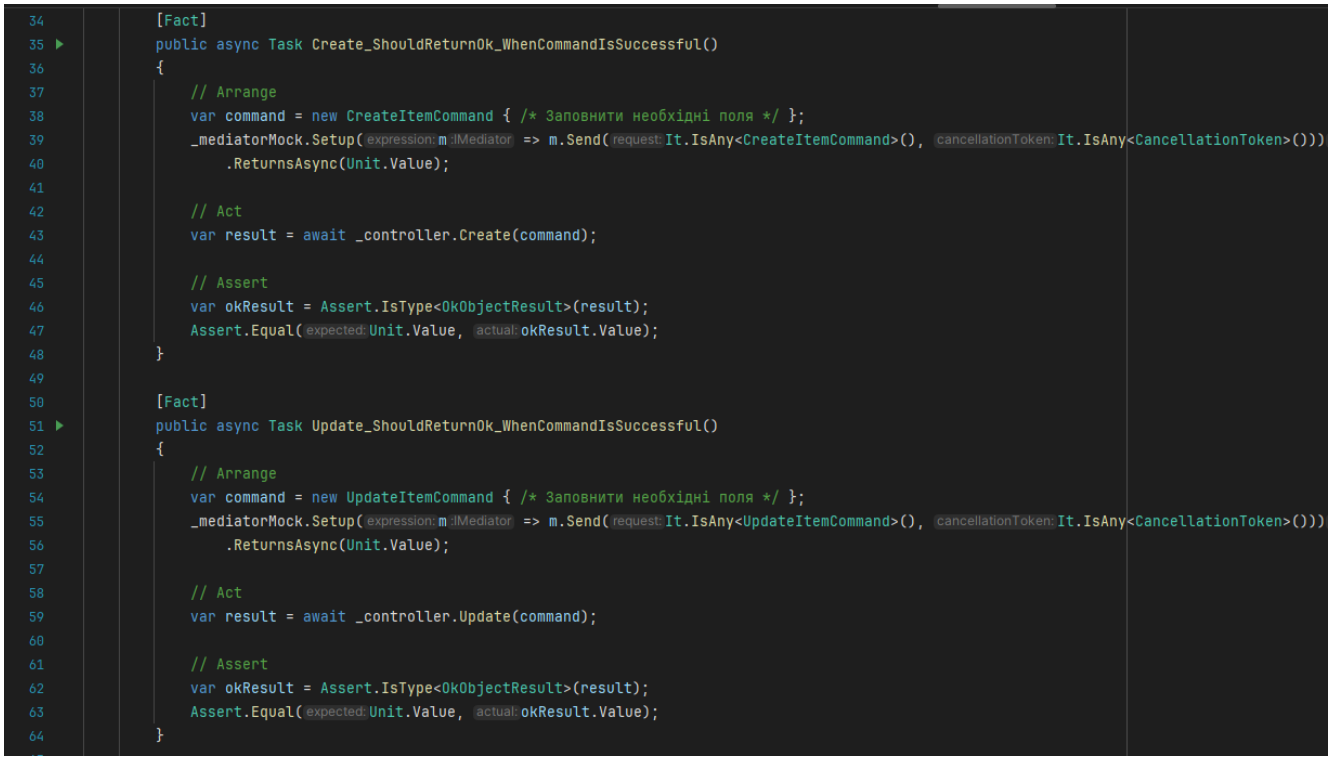
Ім'я	Готвянський К. П.
Дата	21.05.2025
Версія продукту	1.0
Назва тесту	Запит до захищеного ендпоінту з непідходящою роллю
Вхідні дані	Неважливі
Кроки	1. Перейти до інтерфейсу свагеру 2. Авторизуватись як хто завгодно, тільки не інвентарний менеджер 3. Обрати ендпоінт «**POST/api/v1/items»** та виконати запит
Очікуваний результат	Помилка
Фактичний результат	403 Forbidden – You can’t access this resource
Відгук	Успішно пройдений тест

Юніт-тестування є фундаментальним етапом у процесі розробки програмного забезпечення, спрямованим на перевірку коректності окремих компонентів системи в ізоляції. Цей процес є невід'ємною частиною роботи всіх учасників команди розробки. Розробники тестують свої модулі, щоб переконатися у їхній правильній поведінці в граничних випадках, тестувальники перевіряють, чи відповідає функціонал специфікаціям, а особи, відповідальні за демонстрацію продукту, забезпечують, щоб під час презентації не виникало збоїв. Юніт-тести дозволяють виявити помилки на ранніх етапах, що значно знижує витрати на їх виправлення.

У процесі розробки веб-додатків я, як розробник, використовую юніт-тести для перевірки контролерів API, зокрема для забезпечення коректної роботи ендпоінтів. Для цього застосовую два потужні інструменти: xUnit і Moq. xUnit — це сучасний фреймворк для тестування в екосистемі .NET, який підтримує створення чітких і структурованих тестів за допомогою анотацій, таких як [Fact] для одиничних тестів або [Theory] для параметризованих сценаріїв. Він забезпечує зручний механізм для виконання асинхронних тестів, що є критично важливим для тестування API. Moq, у свою чергу, є бібліотекою для створення моків, яка дозволяє імітувати поведінку залежностей, таких як об'єкти IMediator, що використовуються в контролерах, для ізоляції тестованого коду від зовнішніх компонентів.

На прикладі юніт-тестів для контролера ItemsController (рис. 5.3) можна побачити, як xUnit і Moq застосовуються для перевірки ендпоінтів. Кожен тест, створений за допомогою xUnit, перевіряє окремий метод контролера, наприклад, створення елемента через POST /api/v1/items або отримання списку елементів через GET /api/v1/items. Moq дозволяє налаштувати поведінку залежностей, таких як повернення заздалегідь визначених даних із IMediator, що імітує реальну взаємодію без підключення до бази даних чи інших сервісів. Тести побудовані за шаблоном AAA (Arrange, Act, Assert), де спочатку конфігуруються моки та вхідні дані, потім викликається метод контролера, а після цього перевіряється коректність результату, наприклад, повернення OkObjectResult із очікуваними даними.

xUnit і Moq є стандартом у .NET-розробці завдяки своїй гнучкості та простоті використання. xUnit забезпечує легке написання та запуск тестів, а також інтеграцію з CI/CD-системами для автоматизації тестування. Moq дозволяє ізолювати тести від зовнішніх залежностей, що робить їх швидкими та надійними. Ці інструменти значно спрощують перевірку функціональності ендпоінтів, дозволяючи розробникам швидко виявляти помилки та забезпечувати стабільність коду перед його розгортанням у продакшен.



```
34      [Fact]
35      public async Task Create_ShouldReturnOk_WhenCommandIsSuccessful()
36      {
37          // Arrange
38          var command = new CreateItemCommand { /* Заповнити необхідні поля */ };
39          _mediatorMock.Setup(expression: m => m.Send(request: It.IsAny<CreateItemCommand>(), cancellationToken: It.IsAny<CancellationToken>()))
40              .ReturnsAsync(Unit.Value);
41
42          // Act
43          var result = await _controller.Create(command);
44
45          // Assert
46          var okResult = Assert.IsType<OkObjectResult>(result);
47          Assert.Equal(expected: Unit.Value, actual: okResult.Value);
48      }
49
50      [Fact]
51      public async Task Update_ShouldReturnOk_WhenCommandIsSuccessful()
52      {
53          // Arrange
54          var command = new UpdateItemCommand { /* Заповнити необхідні поля */ };
55          _mediatorMock.Setup(expression: m => m.Send(request: It.IsAny<UpdateItemCommand>(), cancellationToken: It.IsAny<CancellationToken>()))
56              .ReturnsAsync(Unit.Value);
57
58          // Act
59          var result = await _controller.Update(command);
60
61          // Assert
62          var okResult = Assert.IsType<OkObjectResult>(result);
63          Assert.Equal(expected: Unit.Value, actual: okResult.Value);
64      }
```

Рисунок 5.3 – Юніт-тести для Items (рисунок виконано самостійно)

Таким чином, були визначені методи тестування та було проведено тестування серверної частини додатку.

6 ВПРОВАДЖЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

У процесі вивчення теми, пов’язаної зі створенням програмного забезпечення для керування персоналом і матеріальними ресурсами в готельній сфері, було підготовлено тези доповіді на XXIX Міжнародний молодіжний форум «РАДІОЕЛЕКТРОНІКА ТА МОЛОДЬ У XXI столітті» [12].

Тези отримали назву: «Програмне забезпечення для організації роботи персоналу та управління інвентарем у готельному бізнесі». Повний текст тез подано в додатку А до цього документа.

У тезах акцентовано увагу на автоматизації повсякденних операцій у готелях, зокрема на оптимізації роботи персоналу, веденні обліку майна й устаткування, а також підвищенні ефективності внутрішніх процесів. Було проаналізовано сучасні тенденції цифровізації готельної індустрії, виявлено ключові виклики та запропоновано можливі рішення. Окремо розглянуто аспекти забезпечення кібербезпеки, масштабування системи та її інтеграції із зовнішніми платформами.

ВИСНОВКИ

В ході виконання завдання було досягнуто значних успіхів у створенні серверної частини програмного забезпечення для управління персоналом та інвентарем для готельного бізнесу. На початковому етапі було здійснено огляд наявних рішень і підходів, які застосовуються готелями для вирішення цих проблем. Цей аналіз допоміг виявити основні проблеми та недоліки, зокрема труднощі в координації роботи співробітників, управлінні інвентарем, а також у обробці й збереженні даних про взаємодію з інвентарем. Виявлені обмеження лягли в основу для розробки детальних і чітких вимог до системи.

На наступному кроці було визначено вимоги до програмного продукту, які охоплювали функціональні та нефункціональні аспекти. Зокрема, було окреслено ключові компоненти системи, такі як управління завданнями, реєстрація нових працівників, замовлення предметів і захист даних, а також можлива інтеграція з іншими платформами. Ці вимоги стали основою для проєктування архітектури системи. Окрім того, було сформульовано специфікації для серверної частини, включно з розробкою безпечного API для обміну даними між сервером і клієнтами та забезпеченням високого рівня захисту інформації.

На етапі проєктування та створення архітектури було розроблено модульну структуру з використанням сучасних технологій, зокрема ASP.NET Core. Як базу даних обрано PostgreSQL, а для розгортання системи використано Render, що гарантує надійне зберігання даних і швидкий доступ до них. Завдяки вибору цих технологій система матиме високу продуктивність, масштабованість і безпеку.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. C# Guide - .NET managed language. URL: https://learn.microsoft.com/en-us/dotnet/csharp/ (дата звернення: 01.05.2025).

2. ASP.NET Core, an open-source web development framework | .NET. URL: https://dotnet.microsoft.com/en-us/apps/aspnet (date of access: 01.05.2025).

3. Erich Gamma. Design Patterns: Elements of Reusable Object-Oriented Software Elements of Reusable Object-Oriented Software. Prentice Hall, 1994.

4. Роберт Мартін. Чиста архітектура. 2-ге вид. Львів: Фабула, 2022

5. RoomRaccoon. URL: https://roomraccoon.com/ (дата звернення: 03.05.2025).

6. Hotel Management System- Cloud-based Hotel PMS | Hotelogix. URL: https://www.hotelogix.com/ (дата звернення: 03.05.2025).

7. Entity Framework documentation hub. URL: https://learn.microsoft.com/en-us/ef/ (дата звернення: 03.05.2025).

8. JetBrains. Rider: The Cross-Platform .NET IDE from JetBrains. URL: https://www.jetbrains.com/rider/ (дата звернення: 06.05.2025).

9. Docker: Accelerated Container Application Development. URL: https://www.docker.com/ (дата звернення: 06.05.2025).

10. Cloud Application Platform | Render. URL: https://render.com/ (date of access: 08.05.2025).

11. API Documentation & Design Tools for Teams | Swagger. URL: https://swagger.io/ (дата звернення: 10.05.2025).

12. Готвянський К.П., Примаченко М.Є. Програмна система для адміністрування персоналом та інвентарем готелю. 29-й Міжнародний молодіжний форум «РАДІОЕЛЕКТРОНІКА ТА МОЛОДЬ В XXI ст.», м. Харків. С. 255-257