МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА

МИСЬКО БОГДАН ВОЛОДИМИРОВИЧ

Допускається до захисту:
в.о. завідувача кафедри
інформаційних технологій
канд. техн. наук, доцент
_____ О. В. Зелінська
«____»_____ 20__ р.

РОЗРОБКА СЕРВЕРНОЇ ЧАСТИНИ CRM-СИСТЕМИ ДЛЯ ШКОЛИ ІНОЗЕМНИХ МОВ

Спеціальність 122 Комп’ютерні науки

Кваліфікаційна (бакалаврська) робота

Керівник:
І. В. Фриз, старший викладач кафедри
інформаційних технологій,
к. ф.-м. н.
_____

Оцінка: _____ / _____ / _____
(бали/за шкалою ЄКТС/за національною шкалою)
Голова ЕК: _____

Вінниця – 2025

АНОТАЦІЯ

Мисько Б.В. Розробка серверної частини CRM-системи для школи іноземних мов. Спеціальність 122 «Комп'ютерні науки», освітня програма «Комп'ютерні науки». Донецький національний університет імені Василя Стуса, Вінниця 2025.

У кваліфікаційній (бакалаврській) роботі досліджено функціональні можливості та структуру CRM-систем, проаналізовано сучасні технології розробки серверної частини та обґрунтовано вибір архітектурного підходу. Результатом роботи є розроблена серверна частина CRM-системи для школи іноземних мов, реалізована мовою C# з використанням ASP.NET Web API та бази даних PostgreSQL. Система забезпечує надійне збереження і обробку даних, підтримує RESTful API для керування навчальним процесом, а також містить механізми JWT-аутентифікації, хешування паролів (алгоритм BCrypt) та логування (Serilog).

Ключові слова: CRM-система, серверна частина, REST API, аутентифікація, авторизація, PostgreSQL, ASP.NET Core.

57 ст. 4 рис., 2 табл., 4 дод., 40 джерел.

ABSTRACT

Mysko B.V. **Development of the Server Part of the CRM System for a School of Foreign Languages**. Specialty 122 «Computer Science», educational program «Computer Science». Vasyl Stus Donetsk National University, Vinnytsia, 2025.

In the qualification (bachelor's) work, the functional capabilities and structure of CRM systems were explored, modern back-end development technologies were analyzed, and a suitable architectural approach was justified. As a result, the server part of a CRM system for a language school was developed using C# and ASP.NET Web API with a PostgreSQL database. The system provides reliable data storage and processing, implements a RESTful API for managing the educational process, and

includes mechanisms for JWT-based user authentication, password hashing (BCrypt), and logging via Serilog.

Keywords: CRM system, server side, REST API, authentication, authorization, PostgreSQL, ASP.NET Core.

ЗМІСТ

ВСТУП 5

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ РОЗРОБКИ BACK-END ЧАСТИНИ CRM-СИСТЕМИ 8

1.1 Функціональні можливості та структура CRM-системи 8

1.2 Аналіз сучасних технологій для розробки back-end частини 9

1.3 Архітектурні патерни та підходи до побудови серверної частини CRM-системи 14

РОЗДІЛ 2. ПРОЄКТУВАННЯ СЕРВЕРНОЇ ЧАСТИНИ CRM-СИСТЕМИ 20

2.1 Визначення функціональних вимог бази даних та її проєктування 20

2.2 Розробка архітектури програмної реалізації back-end частини 26

2.3 Реалізація алгоритмів автентифікації, авторизації та безпеки 30

РОЗДІЛ 3. ТЕСТУВАННЯ, ОПТИМІЗАЦІЯ ТА ВПРОВАДЖЕННЯ BACK-END ЧАСТИНИ CRM 34

3.1 Розробка API для її інтеграції з front-end частиною 34

3.2 Тестування та оптимізація back-end частини 40

3.3 Аналіз результатів розробки та перспективи розвитку 45

ВИСНОВКИ 51

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 53

ДОДАТКИ 58

ВСТУП

У сучасних умовах стрімкого розвитку цифрових технологій бізнесові процеси дедалі більше потребують автоматизації, систематизації та аналітичної підтримки. Це особливо актуально для освітньої сфери, зокрема власне для приватних шкіл зі спрямуванням на вивчення іноземних мов, які стикаються з необхідністю ефективно управляти взаємодією з клієнтами, організовувати внутрішні процеси, вести облік занять, контролювати оплату, відстежувати успішність студентів тощо. Саме для цього доцільним є впровадження CRM-систем – програмного забезпечення, яке надає можливості управління взаєминами з клієнтами та оптимізації внутрішніх бізнес-процесів.

Незважаючи на наявність великої кількості універсальних CRM-систем на ринку, такі рішення часто є надлишково функціональними або ж недостатньо адаптованими до специфіки навчального процесу у школах іноземних мов. Це створює потребу у розробці кастомізованих систем, що враховують специфічні бізнес-процеси та вимоги конкретних закладів освіти. У зв’язку з цим виникає актуальна проблема створення серверної частини CRM-системи, що забезпечить надійну обробку даних, захист персональної інформації користувачів, гнучку інтеграцію з клієнтською частиною та зручний інтерфейс для адміністраторів і викладачів.

Існує чимало досліджень з питань проєктування та реалізації CRM-систем, в яких було розглянуто принципи управління клієнтськими базами, процеси цифрової трансформації бізнесу, моделі побудови інформаційних систем, архітектуру розподілених систем тощо. Проте більшість із цих досліджень мають загальний характер або орієнтовані на великі підприємства. Потреба ж у розробці адаптованого серверного функціонала для невеликих освітніх організацій досі лишається відкритим і практично важливим завданням. Тому розробка серверної частини CRM-системи для школи іноземних мов, з урахуванням її функціональних, безпекових та інтеграційних вимог, є актуальним напрямом, що

поєднує теоретичні напрацювання в галузі інформаційних технологій та їх прикладну реалізацію у сфері освіти.

Метою бакалаврської роботи є розробка функціональної, надійної та масштабованої серверної частини CRM-системи, орієнтованої на потреби школи іноземних мов.

Завданнями дослідження:

- визначити основні функціональні можливості та вимоги до CRM-системи для освітнього закладу;
- проаналізувати сучасні технології для розробки серверної частини десктопних застосунків, провести їх критичний аналіз;
- дослідити архітектурні підходи до побудови back-end частини;
- сформувати вимоги до функціоналу системи та спроєктувати структуру бази даних;
- реалізувати алгоритми автентифікації, авторизації та захисту персональних даних;
- створити REST API для взаємодії з клієнтською частиною системи;
- протестувати та оптимізувати back-end частину CRM-системи відповідно до реальних сценаріїв її використання;
- проаналізувати отримані результати та визначити перспективи подальшого розвитку.

Об’єктом дослідження є процес автоматизації внутрішньої діяльності школи іноземних мов за допомогою CRM-системи.

Предметом дослідження є серверна частина CRM-системи: її функціональне наповнення, архітектура, реалізація API, безпека та інтеграція з іншими компонентами програмного забезпечення.

Теоретичне значення роботи полягає у систематизації знань про архітектурні моделі, патерни проєктування та сучасні інструменти для створення back-end частини CRM-систем, а також у виявленні переваг і недоліків наявних підходів до організації серверної логіки в освітньому програмному забезпеченні.

Практичне значення дослідження полягає у реалізації серверної частини CRM-системи, яка може бути використана у діяльності шкіл іноземних мов або адаптована до інших освітніх проєктів. Запропоноване рішення базується на сучасних технологіях та забезпечує можливості для масштабування, розширення функціональності та подальшого розвитку.

Апробація результатів дослідження: результати дослідження доповідалися на VI Всеукраїнській науково-практичній конференції здобувачів вищої освіти та молодих вчених «Прикладні інформаційні технології», 22 травня 2025 р., Донецький національний університет імені Василя Стуса, м. Вінниця, тема доповіді: «Інтеграція JWT та алгоритмів гешування для автентифікації та авторизації в CRM-системах».

Кваліфікаційна робота складається з **трьох основних розділів**, кожен з яких логічно розкриває етапи дослідження та реалізації серверної частини CRM-системи: розділ 1 розкриває теоретичні засади побудови CRM-систем, функціональні можливості та аналіз сучасних технологій розробки серверної частини; розділ 2 містить безпосередньо проєктування серверної частини, бази даних та реалізацію систем автентифікації та авторизації; розділ 3 присвячений інтеграції розробленого API з клієнтською частиною, тестуванню та аналізу ефективності роботи системи.

РОЗДІЛ 1
ТЕОРЕТИЧНІ ОСНОВИ РОЗРОБКИ BACK-END ЧАСТИНИ CRM-СИСТЕМИ

1.1 Функціональні можливості та структура CRM-системи

CRM-система (від англ. *Customer Relationship Management*) – це програмне забезпечення для управління взаємовідносинами з клієнтами та оптимізації бізнес-процесів компанії. Впровадження CRM дозволяє централізовано зберігати та обробляти інформацію про клієнтів, автоматизувати рутинні операції і покращувати взаємодію на всіх етапах роботи з клієнтом [1]. У контексті освітніх послуг, зокрема приватних шкіл іноземних мов, CRM-система допомагає організувати навчальний процес, вести облік студентів і викладачів, контролювати розклад занять і платежі, а також підтримувати комунікацію з учнями та їх батьками. Традиційні методи (наприклад, журнали або електронні таблиці) не забезпечують достатньої прозорості й ефективності, тому використання CRM-платформи сприяє підвищенню якості управління школою [2].

Основними функціональними можливостями CRM-системи для мовної школи є:

1. Управління даними студентів: зберігання детальної інформації про студентів (ПІБ, дата народження, контактні дані), а також про їхніх батьків або контактних осіб. Ведення історії навчання кожного студента, включно з відвідуванням занять та успішністю.

2. Управління групами та розкладом занять: формування навчальних груп за мовами і рівнями, призначення викладачів до груп, планування розкладу занять (дні тижня, час проведення). Відстеження присутності студентів на заняттях.

3. Управління персоналом і викладачами: збереження даних про співробітників (адміністраторів, викладачів), їхні ролі і контактну інформацію. Призначення викладачів на навчальні групи, контроль навантаження.

4. Укладання договорів та облік оплат: реєстрація договорів з клієнтами (студентами або їхніми батьками), ведення інформації про дату укладення договору, суму оплати, стан платежів. Генерація рахунків та квитанцій, відстеження своєчасності оплат.

5. Комунікація та повідомлення: можливість фіксувати та автоматизувати комунікацію зі студентами: розсилання повідомлень про зміни в розкладі, нагадування про оплату, повідомлення про нові курси тощо. Інтеграція з електронною поштою або месенджерами для відправки групових повідомлень.

6. Аналітика та звітність: формування звітів для адміністраторів школи: успішність студентів, відвідуваність занять, фінансова звітність (отримані платежі, заборгованості), ефективність роботи викладачів. Це допомагає оцінювати якість освітнього процесу і приймати обґрунтовані рішення щодо розвитку школи [3].

Відповідно до цих вимог необхідно спроєктувати структуру системи, тобто визначити основні сутності (камери зберігання даних) та зв’язки між ними. Структура бази даних CRM-системи має підтримувати всі ключові бізнес-об’єкти школи: студентів, викладачів (співробітників), навчальні групи, розклад, договори тощо. Це буде розроблено в наступному розділі.

1.2 Аналіз сучасних технологій для розробки back-end частини

Для розробки серверної (back-end) частини сучасних застосунків існує широкий спектр технологій – мов програмування, фреймворків та систем управління базами даних. Вибір стеку технологій впливає на продуктивність, масштабованість, безпеку та швидкість розробки системи, тому важливо проаналізувати найбільш популярні та придатні варіанти перед прийняттям архітектурних рішень. У цьому підрозділі буде розглянуто кілька провідних

технологій, які використовуються для побудови десктопних сервісів та CRM-систем, і обґрунтуємо вибір засобів для реалізації серверної частини CRM школи.

На сьогодні серед найбільш поширених мов програмування для серверної розробки існують – C# (.NET), Java, JavaScript (Node.js), Python, PHP, а також інші (Ruby, Go тощо). Кожна з них має розвинуті фреймворки, що прискорюють створення API та обробку запитів. Розгляньмо коротко особливості деяких із цих технологій:

- C# / ASP.NET Core. Платформа .NET Core (нині .NET) – це крос-платформений фреймворк від Microsoft для розробки веб, мобільних і десктопних застосунків [4]. В основі – мова C#, яка є строго типізованою, об’єктно-орієнтованою і компілюється у проміжний код (IL) з подальшою JIT-компіляцією. Фреймворк ASP.NET Core дозволяє створювати RESTful сервіси (Web API) та вебсайти (MVC, Razor Pages) [5]. Сильними сторонами ASP.NET Core є висока продуктивність завдяки багатопотоковості та ефективному керуванню пам’яттю, а також підтримка інструментів промислового рівня (потужні IDE, засоби відлагодження, профілювання тощо) [6]. За даними досліджень, ASP.NET Core перевершує Node.js при виконанні ресурсомістких завдань, що потребують інтенсивного використання CPU [7]. Ця технологія добре підходить для масштабних корпоративних застосунків, де важливі надійність, швидкодія і підтримка складної бізнес-логіки.
- Java / Spring. Java традиційно використовується для enterprise-систем [8]. Фреймворк Spring та його модуль Spring Boot забезпечують зручне створення сервісів на Java, що реалізовує принципи інверсії керування, впровадження залежностей, інтеграції з базами даних (через Hibernate) тощо. Java є компільованою мовою, що працює у віртуальній машині JVM, що забезпечує портованість між платформами [9-10]. Продуктивність Java-сервісів також на високому рівні, а розгалужена екосистема бібліотек і велика спільнота розробників роблять Spring одним із найбільш потужних рішень для розробки back-end [11]. Недоліком може бути відносно більший поріг входження та складність налаштування для новачків у порівнянні з деякими іншими мовами.

- JavaScript / Node.js. Платформа Node.js дозволяє виконувати JavaScript на стороні сервера, що було революційним кроком для розробки. Node.js побудований на базі швидкого рушія V8 та застосовує неблокуючу, подіє-орієнтовану модель вводу/виводу [35]. Основний фреймворк для побудови API на Node.js – Express, легкий і мінімалістичний. Перевагою Node.js, звісно, є JavaScript, яка є єдиною мовою і на front-end, і на back-end, а також висока продуктивність при обробці великої кількості одночасних I/O-запитів завдяки асинхронній моделі [36]. Як відзначають фахівці, Node.js особливо ефективний для легкої, швидкої розробки прототипів та мікросервісів, що забезпечує швидкий запуск та відгук системи. Втім, для задач, що потребують інтенсивних обчислень, продуктивність Node.js може поступатися рішенням на базі .NET через однопотокову природу Node і накладні витрати інтерпретації JavaScript-коду [37]. Node.js широко використовується такими компаніями, як Netflix, Twitter, LinkedIn, що підтверджує його придатність для високо-навантажених сервісів.
- Python / Django, Flask. Python – інтерпретована мова, відома простотою синтаксису та швидкістю розробки [8]. Фреймворк Django надає рішення «все-в-одному» для розробки, включно з ORM, панеллю адміністратора та ін. Flask – мікрофреймворк, що дає більшу гнучкість, але вимагає ручного підключення компонентів [33]. Сервери на Python дещо поступаються в швидкодії C# і Java, особливо під значним навантаженням, проте відзначаються швидким циклом розробки і великим набором бібліотек, зокрема для здійснення обчислень, що може бути корисно, якщо CRM потребуватиме аналітичних модулів [38]. Python часто обирають для невеликих і середніх проєктів, де швидкість розробки важливіша за пікову продуктивність [8].
- PHP / Laravel. PHP – одна з найстаріших мов розробки, яка досі залишається популярною. Сучасний фреймворк Laravel забезпечує структурований підхід до розробки на PHP, включаючи ORM (Eloquent), механізми міграцій БД, черги завдань тощо. PHP традиційно використовувався для генерації сторінок, але з Laravel/PHP можна будувати і RESTful API [12].

Продуктивність PHP суттєво покращилася з виходом PHP 7+, проте в сценаріях з великою кількістю одночасних з’єднань йому можуть знадобитися додаткові рішення (напр. шардінг, балансування навантаження) [34]. Перевага PHP – низький поріг входження і велика кількість готових рішень, однак для масштабних систем вибір все частіше схиляється на користь більш сучасних платформ, як-от .NET чи Node.

Важливим компонентом серверної частини є саме зберігання даних, тож розгляньмо системи керування базами даних (СКБД). Розробник back-end має вибір між реляційними БД (SQL) та нереляційними БД (NoSQL). Класичні СКБД – MySQL/MariaDB, PostgreSQL, Microsoft SQL Server, Oracle та інші, забезпечують збереження даних у вигляді таблиць зі строгими схемами і підтримують мову структурованих запитів SQL [13-14, 31]. Вони гарантують виконання транзакцій за принципами ACID, що важливо для фінансових даних і цілісності інформації. У випадку CRM-системи навчального закладу більшість даних мають чітку структуру (студенти, групи, платежі), тому реляційна база даних є природним вибором. Для проєкту написання серверної частини CRM-системи доцільно використати Microsoft SQL Server (з огляду на зручну інтеграцію з Entity Framework в .NET) або відкриту альтернативу PostgreSQL [15]. Обидві СКБД підтримують складні зв’язки, транзакції та зберігання процедур, що забезпечує надійність даних.

NoSQL-рішення (такі як MongoDB, Firebase, CouchDB тощо) зберігають дані у більш гнучких формах, а саме: документи, ключ-значення, графи або колонкові сховища [15]. Вони часто використовуються, коли структура даних динамічно змінюється або потрібна горизонтальна масштабованість на багатьох вузлах без складних JOIN-операцій [16]. У контексті CRM для школи іноземних мов немає потреби в NoSQL, оскільки інформація добре моделюється реляційно. Однак окремі модулі (наприклад, логування дій користувачів або зберігання великих масивів напівструктурованих даних) можуть потенційно бути ефективніше реалізовані з використанням NoSQL. У більшості випадків реляційна БД забезпечує баланс між простотою запитів та цілісністю даних.

Сучасні back-end фреймворки часто включають ORM-бібліотеки (Object-Relational Mapping), що полегшують роботу з базою даних, представляючи записи у вигляді об’єктів мови програмування. Наприклад, у .NET використовується Entity Framework Core, у Java – Hibernate, у Python (Django) – вбудований ORM Django, у PHP (Laravel) – Eloquent [18]. ORM дозволяє розробнику маніпулювати даними через об’єктні моделі (класи), а бібліотека під капотом будує необхідні SQL-запити. Це підвищує продуктивність розробки і знижує кількість помилок, пов’язаних з написанням SQL вручну. Водночас, важливо розуміти особливості ORM щоб уникнути неефективних запитів (N+1 problem тощо) під час роботи з великою кількістю зв’язаних даних.

У таблиці А.1, яка представлена в додатку А, наведено порівняльні дані щодо основних технологій розробки, які розглядалися як варіанти для реалізації серверної частини CRM-системи (ASP.NET Core та Node.js для JavaScript-платформи).

Проаналізувавши таблицю А.1, можна зробити висновок, що обидва підходи мають переваги. Node.js забезпечує простоту і швидкість розробки, а також чудово виконує одночасне обслуговування великої кількості клієнтських з’єднань. Натомість ASP.NET Core на C# демонструє вищу продуктивність на важких обчисленнях та надає більш строгий каркас для побудови масштабованої архітектури великого застосунку [7].

Отже, з огляду на потреби розробки серверної частини десктопного застосунку CRM-системи, де важливі цілісність даних, складні транзакційні операції (договір та оплати) і чітка структура коду, прийнято рішення реалізовувати серверну частину на платформі ASP.NET Core з використанням мови C# та ORM Entity Framework Core. Ця технологія поєднує високу продуктивність з багатим набором інструментів для прискорення розробки і забезпечує надійну інтеграцію з реляційною базою даних. Як СКБД буде використана реляційна система PostgreSQL відповідно до спроєктованої схеми даних [17]. Застосування саме такого стеку (C# + .NET + SQL) відповідає

вимогам до функціоналу та масштабу системи і закладає основу для реалізації всіх необхідних можливостей CRM.

1.3 Архітектурні патерни та підходи до побудови серверної частини CRM-системи

Під час проєктування серверної частини необхідно визначити архітектурний підхід, тобто як структуровано компоненти системи, як вони взаємодіють і розгортаються. Правильний вибір архітектури забезпечить гнучкість розвитку CRM, її масштабованість та підтримуваність коду. У цьому підрозділі буде розглянуто ключові архітектурні патерни, актуальні для систем, а саме: монолітна архітектура та мікросервіси, багатошарова архітектура, принципи REST для API, а також підходи до автентифікації та розгортання застосунку на сервері.

Для побудови серверної частини існують два основні стилі – це моноліт (єдиний цілісний застосунок) і мікросервіси (набір незалежних сервісів) [19]. Монолітний застосунок є цілісною програмою, що містить увесь необхідний функціонал, тоді як мікросервісна архітектура розбиває систему на набір дрібніших сервісів, кожен з яких відповідає за свою ділянку функціоналу і може розгортатися окремо. Монолітна архітектура простіша в розробці та розгортанні на ранніх етапах, оскільки всі компоненти розташовані в одній кодовій базі та запускаються як один процес [20]. Це підходить для відносно невеликих або середніх систем, де команда розробки невелика і модулі тісно пов’язані. З іншого боку, у міру розростання моноліту можуть виникнути проблеми з підтримкою, оскільки зміна в одній частині потребує повторної збірки і розгортання всього застосунку, у результаті важче масштабувати окремі підсистеми.

Мікросервіси дозволяють розбити систему на автономні модулі, які взаємодіють через чітко визначені API (наприклад, HTTP/REST або повідомлення). Кожен сервіс можна масштабувати незалежно (наприклад, додати потужності тільки для модуля, що відповідає за розклад, якщо саме він є

«вузьким місцем» продуктивності). Цей підхід підвищує гнучкість розробки – різні команди можуть працювати над різними сервісами, можна використовувати різні технології для кожного (Polyglot) [21]. Класичним прикладом успішного переходу від моноліту до мікросервісів є кейс Netflix, який зі зростанням навантаження розділив величезний моноліт на тисячі дрібних сервісів, що оновлюються й масштабуються незалежно. Однак, мікросервіси додають складності, бо потрібно впроваджувати механізми оркестрації, балансування навантаження, забезпечення узгодженості даних між сервісами, а також DevOps-практики для керування великою кількістю розгортань [22]. Для невеликої організації це може бути занадто витратно.

Незалежно від того, моноліт чи мікросервіси, внутрішня структура серверного застосунку, як правило, організована в шари. Класична трирівнева (N-Layer) архітектура включає: рівень представлення (UI), рівень бізнес-логіки (BLL) та рівень доступу до даних (DAL).

Побудова CRM-системи за принципом шарів означає, що логіка роботи з даними (створення картки про студента, запис у групу, нарахування оплати) відокремлена від деталей зберігання даних і від способу взаємодії з користувачем. Зокрема, контролери Web API звертаються не безпосередньо до бази даних, а до сервісів бізнес-логіки. Ті, у свою чергу, оперують об’єктами доменної моделі та можуть викликати рівень доступу до даних (репозиторії або ORM) для зчитування чи збереження інформації. Така структура запобігає порушенню принципу єдиного обов’язку, де кожен шар відповідає за свою функцію і може модифікуватися незалежно. Наприклад, можна замінити СКБД (перейти з SQL Server на іншу) шляхом зміни лише DAL-рівня, не торкнувшись бізнес-логіки або контролерів. Окрім того, її легше тестувати модульно, оскільки бізнес-логіку можна перевіряти за допомогою підміни шару даних на фейковий, і навпаки.

На рис. 1.1 зображено спрощену багатошарову архітектуру застосунку. Клієнти надсилають запити до API-шару (контролери), який взаємодіє з внутрішніми сервісами та через них – з базою даних. Між клієнтом і сервером

може знаходитись проксі або розподілювач навантаження, що підвищує масштабованість і безпеку системи.

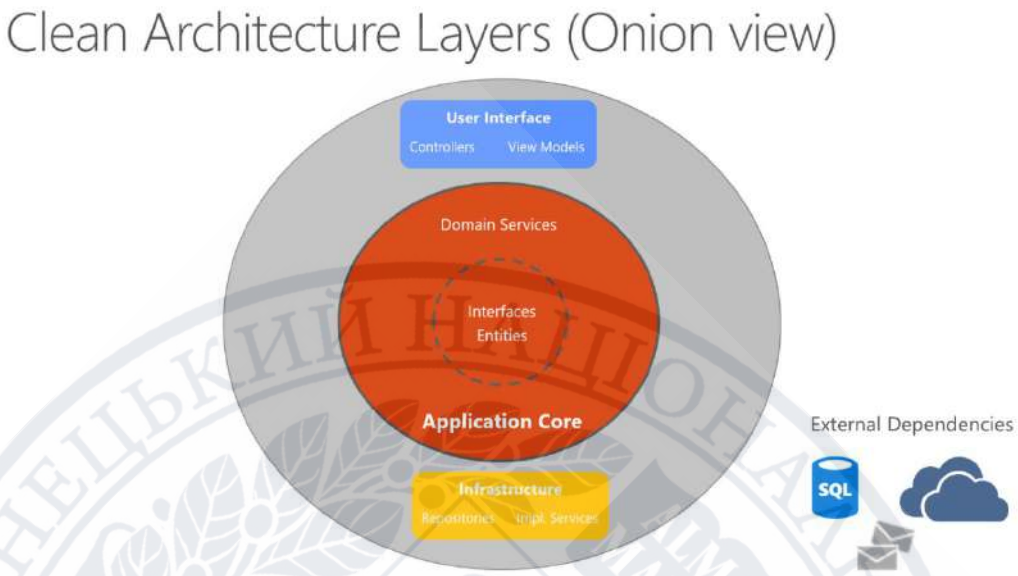

Рисунок 1.1 – Багатошарова архітектура

Як показано на рис. 1.1, контролери виконують роль «вхідних воріт» до серверної частини. Вони отримують HTTP-запит, валідують його, звертаються до відповідного сервісного методу бізнес-логіки, а отримавши результат – формують відповідь (наприклад, у форматі JSON) назад клієнту. Рівень бізнес-логіки інкапсулює правила: тут реалізується перевірка умов (чи є вільні місця в групі перед додаванням студента), розрахунки (наприклад, обчислення загальної суми оплати за певний період) та інші операції. Цей шар працює з абстрактними інтерфейсами доступу до даних (наприклад, через шаблон *Repository*), без прямої залежності від конкретної СКБД. Рівень даних відповідає за збереження та читання, а саме: реалізації репозиторіїв, виклики до ORM (EF Core) або SQL-запитів, оскільки відома структура таблиць і забезпечене виконання транзакцій.

Завершальним аспектом архітектури серверної частини є підхід до розгортання. Традиційно застосунок розгортають на виділеному сервері або в хмарі, де середовище налаштовується вручну (встановлення потрібної версії .NET, СКБД, сервера тощо). Сучасні DevOps-практики віддають перевагу контейнеризації – упаковці застосунку та всіх його залежностей у стандартний контейнер (Docker), який може бути запущений на будь-якому сервері під

управлінням Docker-рушія [23]. Контейнер – це ізольоване середовище, що містить все необхідне для роботи програми (бібліотеки, конфігурацію тощо), завдяки чому досягається повторюваність та портативність, де в будь-якому оточенні контейнер буде поводитись однаково. Docker-контейнер незатратний і запускається швидше, ніж повноцінна віртуальна машина, оскільки використовує ядро ОС хоста спільно з іншими контейнерами [24].

Docker значно полегшує розгортання мікросервісної архітектури, але й для монолітного застосунку дає переваги. Можна створити два контейнери: один із CRM API і другий з СКБД [23]. Запустивши їх спільно, наприклад, за допомогою Docker Compose, отримаємо повністю функціональну систему. При цьому робоче середовище і середовище розробки будуть максимально ідентичними – розробник може локально запускати ті самі контейнері, що й на сервері. Docker також спрощує масштабування, тому що за необхідності можна запустити декілька екземплярів контейнера з CRM API за системою балансування навантаження [25]. Контейнери легко розгортаються у хмарних середовищах (AWS, Azure, GCP) або оркеструються за допомогою Kubernetes. У підсумку оновлення застосунку зводиться до заміни контейнерного образу, що забезпечує швидкість і надійність процесу.

На рис. 1.2 схематично показано архітектуру розгортання системи з використанням Docker. Клієнтські застосунки взаємодіють по HTTP з контейнером CRM API, який у свою чергу з’єднується з контейнером бази даних. Обидва контейнери ізольовані в межах одного хосту Docker. Також з рис. 1.2 видно, що CRM API Container містить все, що потрібно для запуску ASP.NET Core застосунку (в тому числі сам код CRM, .NET Runtime, сторонні бібліотеки), а Database Container – СКБД з даними. Контейнери взаємодіють між собою через внутрішню мережу Docker (наприклад, CRM API звертається до хоста db – контейнера з БД). З боку хост-системи (чи хмарної платформи) достатньо запускати контейнери – немає потреби окремо встановлювати .NET або БД на сервер. Така ізоляція підвищує стабільність, оскільки оновлення одного компонента (наприклад, нова версія CRM) не вплине на інші служби на сервері.

Крім того, у випадку збою контейнер можна швидко перезапустити або замінити, що забезпечить мінімальний простій.

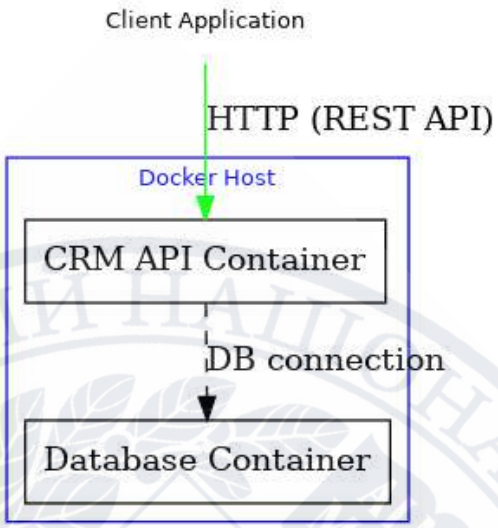

Рисунок 1.2 – Схема розгортання серверної частини CRM у Docker

З точки зору зазначеної розробки, буде підготовано Dockerfile для застосунку, який визначає, як побудувати образ. Для бази даних можна використати готовий образ SQL Server Express з ініціалізацією схемою даних. Під час розгортання через Docker Compose файли конфігурації задають параметри з'єднання між контейнерами, змінні оточення тощо [24]. Все це стане частиною інфраструктурного коду і забезпечить надійне розгортання без ручного втручання. Використання контейнерів у проєкті дасть змогу розгорнути серверну частину CRM-системи школи іноземних мов у хмарі або на локальному сервері з мінімальними зусиллями, спростить підтримку (оновлення версій через перевипуск контейнера) і забезпечить масштабованість у разі зростання навантаження.

Отже, у першому розділі проведено огляд та аналіз теоретичних засад, необхідних для розробки серверної частини CRM-системи для школи іноземних мов. Розглянуто призначення і функціональні можливості CRM-систем, визначено вимоги до функцій саме в освітньому середовищі (управління даними студентів, групами, розкладом, оплатами тощо) і на цій основі спроєктовано структуру бази даних, що включає всі сутності предметної області. Проаналізовано сучасні технології back-end розробки – мови програмування,

фреймворки, СКБД, та обґрунтовано вибір стеку ASP.NET Core + C# + SQL як оптимального для цього проєкту з точки зору продуктивності та надійності. Окрему увагу приділено архітектурним патернам: для серверної частини обрано багатошарову монолітну архітектуру, що спрощує розробку та підтримку CRM і забезпечує розподіл відповідальності між компонентами. Застосування REST-принципів при побудові API дозволить отримати уніфікований та зрозумілий інтерфейс взаємодії клієнтів із сервером, а впровадження JWT-аутентифікації – захистити дані від несанкціонованого доступу. Нарешті, використання контейнеризації (Docker) розглянуто як сучасний підхід до розгортання, який забезпечить портативність і масштабованість системи. Сформовані в розділі теоретичні положення та прийняті архітектурно-технологічні рішення створюють основу для подальшої реалізації серверної частини CRM-системи у практичних розділах роботи.

РОЗДІЛ 2
ПРОЄКТУВАННЯ СЕРВЕРНОЇ ЧАСТИНИ CRM-СИСТЕМИ

2.1 Визначення функціональних вимог бази даних та її проєктування

Проєктування серверної частини CRM-системи для школи іноземних мов є важливим етапом у процесі створення ефективного, надійного та масштабованого програмного забезпечення. Центральне місце в архітектурі серверної частини посідає база даних, яка слугує основним середовищем для зберігання, обробки та управління всією інформацією, пов’язаною з навчальним процесом, користувачами системи, адмініструванням, а також внутрішніми та зовнішніми комунікаціями. Саме база даних є підґрунтям для всіх ключових функцій системи, таких як: реєстрація студентів, формування навчальних груп, управління розкладом, опрацювання особистих даних, збереження фінансової та договірної інформації, а також взаємодія між працівниками та користувачами.

Розробка моделі бази даних у межах архітектури серверної частини здійснюється на основі підходу Code First, що передбачає попереднє визначення структури даних у вигляді класів (сутностей) мовою програмування C#, з подальшим автоматичним створенням структури бази даних засобами ORM (Object-Relational Mapping), зокрема Entity Framework [26]. Такий підхід дає змогу забезпечити узгодженість між бізнес-логікою застосунку та фізичним рівнем зберігання даних, а також дозволяє швидко оновлювати або реорганізовувати структуру бази даних без втручання в SQL-код.

Визначення функціональних вимог до бази даних починається з усвідомлення цільового призначення системи. CRM-система, яка розробляється для школи іноземних мов, повинна забезпечити повний цикл управління навчальним процесом, кадровим складом, студентами, комунікаціями, а також супровідними адміністративними процесами, такими як фінансова звітність і укладання договорів. Кожен з цих функціональних напрямів вимагає наявності

відповідних сутностей у базі даних, які зберігають структуровану інформацію, забезпечують взаємозв'язки між об'єктами та підтримують складні операції вибірки й оновлення даних. Тож, на основі аналізу предметної області було виявлено основні сутності та їх зв'язки, що реалізовані у вигляді класів за допомогою підходу Code First у середовищі .NET [26]. До основних сутностей належать:

- RoleEntity (роль користувача),
- UserEntity (обліковий запис користувача),
- MessageEntity (повідомлення),
- ContactEntity (контактні дані),
- ContractEntity (договори),
- EmployeeEntity (співробітник),
- GroupEntity (група занять),
- GroupLessonDayEntity (графік занять для групи),
- LanguageEntity (мови навчання),
- LessonDayEntity (дні занять),
- PersonEntity (загальні дані особи),
- StudentEntity (студент)
- StudentGroupEntity (зв'язок студент–група).

Першою і однією з основних вимог є необхідність зберігання повної та актуальної інформації про всіх користувачів системи. Це включає адміністраторів, викладачів, студентів, а також інших осіб, які мають доступ до функціоналу CRM. Усі користувачі повинні мати можливість проходити аутентифікацію та авторизацію відповідно до своєї ролі в системі. Реалізація цього механізму потребує наявності сутностей, які забезпечують збереження ієрархічної структури ролей (RoleEntity) та детальної інформації про кожного користувача (UserEntity). На рис 2.1 у моделі UserEntity визначено збереження таких атрибутів, як повне ім'я, дата народження, логін, хеш пароля, електронна

пошта, а також додаткові поля, які використовуються для адміністративних цілей – зокрема, дати створення та оновлення облікового запису.

Рисунок 2.1 – ER-діаграма бази даних CRM-системи

Іншою важливою вимогою є можливість відображення ієрархії ролей користувачів у системі. Як показано на ER-діаграмі на рис. 2.1 це забезпечується за рахунок встановлення зв’язку один-до-багатьох між таблицею ролей (RoleEntity) та таблицею користувачів (UserEntity). Завдяки цьому можна забезпечити централізоване керування доступом і реалізувати розмежування функціональності: адміністратор отримує повний доступ до всіх функцій системи, викладач – до інструментів, пов’язаних із розкладом, групами та студентами, студент – до особистого кабінету, повідомлень і договірної інформації. Код для забезпечення зв’язку між таблицями ролей та користувачів має такий вигляд:

```
public class RoleEntity
{
    public int Id { get; set; }
    public string Name { get; set; } = string.Empty;
```

```
    public ICollection<UserEntity> Users { get; set; } = new
List<UserEntity>();
}
public class UserEntity
{
    public Guid Id { get; set; }
    public string FullName { get; set; } = string.Empty;
    public DateTime BirthDate { get; set; }
    public string Email { get; set; } = string.Empty;
    public string Username { get; set; } = string.Empty;
    public string PasswordHash { get; set; } = string.Empty;
    public int RoleId { get; set; }
    public DateTime CreatedAt { get; set; } = DateTime.UtcNow;
    public DateTime UpdatedAt { get; set; } = DateTime.UtcNow;
    public string? ImageUrl { get; set; }

    public RoleEntity Role { get; set; } = null!;
}
```

Зв’язок з таблицею ролей дає змогу ефективно реалізувати політику контролю доступу, розмежувавши дозволи та дії для кожної категорії користувачів.

Ключовою складовою проєктування бази даних є збереження детальної інформації про учасників навчального процесу. Йдеться про абстрактне уявлення про людину – незалежно від її ролі (студент, викладач, батько/мати тощо). Для цього в системі використовується сутність PersonEntity, і як показано на рис. 2.1 включає поля для збереження повного імені та дати народження. Ця таблиця є основою для інших пов’язаних таблиць, таких як StudentEntity та EmployeeEntity, які реалізовують прив’язку до загальних особових даних через зовнішній ключ PersonId. Такий підхід дозволяє уникнути дублювання інформації та полегшує керування спільними даними. Код для сутності PersonEntity має вигляд:

```
public class PersonEntity
{
    public Guid Id { get; set; }
    public string FullName { get; set; } = string.Empty;
    public DateTime BirthDate { get; set; }

    public ICollection<ContactEntity> Contacts { get; set; } = new
List<ContactEntity>();
}
```

Наступною ключовою вимогою системи є збереження контактної інформації кожної особи, що реалізується за допомогою сутності ContactEntity. Ця таблиця містить відомості про електронну пошту та номер телефону, як показано на рис. 2.1, й пов’язується з PersonEntity за допомогою зовнішнього ключа. Завдяки такій структурі можна забезпечити гнучке розширення контактів (наприклад, кілька електронних адрес або номерів телефонів для однієї особи), а також надалі реалізувати функціонал фільтрації, пошуку й масового розсилання повідомлень.

У системі обліку студентів важливою є можливість зв’язку між студентом і його батьками. На рис. 2.1 видно, що це реалізовано в StudentEntity через необов’язковий зв’язок ParentId, який також вказує на іншу особу з таблиці PersonEntity. Отже, можна формувати пов’язані структури, що відображають родинні зв’язки, що особливо важливо для комунікацій із батьками неповнолітніх студентів, інформування їх про успішність або фінансові питання.

Фінансовий облік у навчальному закладі є невід’ємною частиною управлінського процесу. Для реалізації цього аспекту в системі передбачена таблиця ContractEntity, яка містить відомості про номер договору, дату підписання та суму оплати, а також має зовнішній ключ на таблицю студентів, як показано на рис. 2.1. Це дозволяє вести облік укладених договорів між студентами та закладом, контролювати платежі, генерувати звітність і здійснювати зв’язок з іншими модулями CRM.

Організація навчального процесу потребує створення навчальних груп. Кожна група повинна мати унікальне ім’я, бути прив’язаною до певної мови вивчення та мати графік занять. На рис. 2.1 показано, що для цього в базі даних передбачено таблицю GroupEntity, що містить основну інформацію про групу, а також встановлює зв’язок із таблицею мов (LanguageEntity) та таблицею днів тижня (LessonDayEntity) через проміжну таблицю GroupLessonDayEntity. Остання реалізує зв’язок типу «багато-до-багатьох», що дає змогу закріплювати за кожною групою кілька днів тижня, коли проводяться заняття. Така модель повністю відповідає реаліям навчального процесу в мовних школах, де графіки

занять є гнучкими та часто змінюються. Далі наведено код для таблиць GroupEntity і GroupLessonDayEntity:

```
public class GroupEntity
{
    public int Id { get; set; }
    public string GroupName { get; set; }
    public int LanguageId { get; set; }
    public LanguageEntity Language { get; set; }
    public ICollection<GroupLessonDayEntity> GroupLessonDays { get;
set; }
}

public class GroupLessonDayEntity
{
    public int Id { get; set; }
    public int GroupId { get; set; }
    public int LessonDayId { get; set; }
    public GroupEntity Group { get; set; }
    public LessonDayEntity LessonDay { get; set; }
}
```

Окрім створення груп, система повинна мати можливість реєстрації студентів у відповідних групах. Цей функціонал реалізується за допомогою таблиці StudentGroupEntity, яка встановлює зв’язок між студентом і групою. Окрім зовнішніх ключів, у таблиці зберігається інформація про рівень підготовки студента та номер пари, на яку він зарахований. Така структура дозволяє реалізувати повноцінний механізм ведення розкладу, оцінювання ефективності навчання та планування наповнюваності груп. Код для таблиці StudentGroupEntity:

```
public class StudentGroupEntity
{
    public int Id { get; set; }
    public int StudentId { get; set; }
    public int GroupId { get; set; }
    public StudentEntity Student { get; set; }
    public GroupEntity Group { get; set; }
}
```

Ще однією важливою функціональною вимогою є реалізація внутрішньої системи повідомлень, яка дозволяє надсилати інформаційні чи адміністративні повідомлення студентам або співробітникам. Для цього передбачена таблиця MessageEntity, яка містить тему повідомлення, його текст, дату відправлення, адресу одержувача та зовнішній ключ, що вказує на конкретного користувача. Цей модуль є важливим інструментом комунікації між адміністрацією школи та учасниками освітнього процесу. Для таблиці MessageEntity представлено код:

```
public class MessageEntity
{
    public int Id { get; set; }
    public string Subject { get; set; }
    public string MessageText { get; set; }
    public DateTime SendDate { get; set; }
    public int UserId { get; set; }
    public UserEntity User { get; set; }
}
```

Загалом, структура бази даних, спроєктована з урахуванням зазначених функціональних вимог, дозволяє створити ефективну й масштабовану CRM-систему, яка задовольняє потреби мовного навчального закладу. Вона забезпечує повний цикл роботи з особовими даними, навчальним процесом, фінансовими документами та засобами комунікації, а також є гнучкою для подальшого розширення функціоналу відповідно до зростаючих потреб школи.

2.2 Розробка архітектури програмної реалізації back-end частини

У процесі розробки серверної частини CRM-системи для школи іноземних мов важливим етапом є визначення архітектури для серверної частини. Оскільки система орієнтована на досить обмежену кількість користувачів і обсяги даних, а також має чітко окреслену предметну область, найбільш доцільною є реалізація монолітної архітектури, про яку вже йшлося в першому розділі. Такий підхід дозволить досягти ефективності в розробці, тестуванні та розгортанні, оскільки

зменшується складність інтеграції різних сервісів, а всі компоненти системи будуть зосереджені в одному застосунку [20].

Отже, розробка серверної частини системи для школи іноземних мов базується на монолітній архітектурі, що передбачає реалізацію всіх функціональних блоків в межах одного застосунку. Монолітна архітектура має перевагу в простоті реалізації та розгортання, оскільки передбачає мінімальні зусилля на координацію між різними сервісами та може бути легко розгорнута на сервері або в контейнері. У майбутньому, за необхідності, деякі частини системи можуть бути винесені в окремі мікросервіси (наприклад, аналітика, або модуль розсилки повідомлень), проте на початкових етапах розробки мікросервісна архітектура є надмірно складною.

Вибрана монолітна архітектура, про яку зазначається в підпункті 1.3, базується на платформі ASP.NET Core і використовує RESTful API, що дозволяє легко інтегрувати та обслуговувати компоненти системи [27]. У середині архітектури передбачається чітке розділення між ними:

1. Repository – робота з базою даних (CRUD-операції).

2. Services – інкапсуляція бізнес-логіки та взаємодія з репозиторіями, що працюють із базою даних.

3. Controllers – обробка HTTP-запитів, які взаємодіють із моделями та здійснюють CRUD- операції (створення, отримання, оновлення, видалення) [32].

ASP.NET Core є основною технологією для розробки серверної частини CRM-системи. Це дозволяє зберігати високий рівень інтеграції між компонентами та забезпечує простоту в підтримці і масштабуванні системи [28]. Кожен контролер API реалізує методи для роботи з відповідними сутностями, такими як студенти або групи. Для CRM-системи реалізація багатошарової архітектури заключатиметься в створенні контролерів API для основних сутностей (наприклад, *StudentController*, *GroupController*, *ContractController*), в яких будуть методи для CRUD-операцій (створити, отримати, оновити, видалити) [7-8]. За кожним таким методом стоїть логіка, реалізована в сервісах, що інкапсулюють роботу з декількома сутностями і бізнес-правилами. Сервіси

звертаються до репозиторіїв або безпосередньо до контексту бази даних (EF DbContext) для виконання потрібних запитів. Така архітектура відповідає принципам SOLID і спрощує підтримку системи, бо зміни в бізнес-правилах не впливатимуть на API інтерфейс, і навпаки.

Далі буде наведено невеликий приклад контролера в ASP.NET Web API, що демонструє метод контролера для отримання списку студентів, який викликає сервіс і повертає результат клієнту:

```
[ApiController]
[Route(«api/[controller]»)]
public class StudentController : ControllerBase
{
    private readonly IStudentService _studentService;
    public StudentController(IStudentService studentService)
    {
        _studentService = studentService;
    }

    // GET: api/student
    [HttpGet]
    public async Task<IActionResult> GetAllStudents()
    {
        var students = await _studentService.GetStudentsAsync();
        return Ok(students);
    }
}
```

У цьому фрагменті контролер StudentController через механізм впровадження залежностей (DI) отримує сервіс IStudentService. Метод GetAllStudents() обробляє HTTP GET запит на шлях api/student, викликає сервісний метод GetStudentsAsync() для отримання списку студентів і повертає результат із статусом 200 OK. Контролер не містить бізнес-логіки чи деталей доступу до даних – усе це інкапсульовано в сервісі та рівні збереження, що знаходяться «нижче» по шарі.

Серверна частина CRM буде реалізована як RESTful API, що визначає стиль взаємодії між клієнтом і сервером. REST (Representational State Transfer) – це архітектурний підхід до побудови розподілених систем, який використовує стандартні методи протоколу HTTP для управління ресурсами [28]. RESTful API

оперує поняттям ресурсів, тобто – це студенти, групи, договори тощо, для яких визначаються URL-ідентифікатори і над якими виконуються дії за допомогою методів GET, POST, PUT, DELETE та ін. Дотримання принципів REST означає, що кожен URL відповідає певному об’єкту або колекції об’єктів, а виклики з різними HTTP-методами мають стандартизоване значення.

Важливою властивістю REST є відсутність стану (stateless), бо сервер не зберігає інформацію про стан клієнта між запитами. Кожен запит містить всю необхідну інформацію (наприклад, ідентифікатор ресурсу, токен автентифікації), і сервер обробляє його незалежно. Це спрощує масштабування, оскільки будь-який запит може бути спрямований на будь-який екземпляр серверу, і підвищує надійність – збій одного запиту не впливає на інші. Дотримання REST-принципів робить API передбачуваним та уніфікованим, що полегшує його використання клієнтами і інтеграцію з іншими системами.

Розроблений API відповідатиме стилю REST, що буде використовувати читабельні маршрути (наприклад, /api/students для роботи зі студентами, /api/groups – з групами і т.д.), повертати дані у форматі JSON (за замовчуванням) та застосовувати стандартні коди відповіді HTTP (200 OK, 201 Created, 400 Bad Request, 401 Unauthorized, 404 Not Found тощо). Нижче наведено приклад типового запиту до REST API системи та відповідь сервера:

```
GET /api/students/3 HTTP/1.1
Host: crm-school.com
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...

HTTP/1.1 200 OK
Content-Type: application/json

{
    «id»: «3»,
    «fullName»: «Іваненко Марія»,
    «birthDate»: «2010-05-12»,
    «grade»: 9,
    «parent»: {
        «fullName»: «Іваненко Олена»,
        «phone»: «+3801234567»
    },
    «groups»: [
```

```
        { «groupName»: «English A1 Mon/Wed», «language»: «English»
}
    ]
}
```

У цьому прикладі клієнт надсилає GET-запит на ресурс студента з ID 3. Заголовок Authorization: Bearer <JWT> містить JWT-токен, який використовується для автентифікації. У відповіді сервер повертає JSON-об’єкт з усіма запитуваними даними студента та вкладеними підресурсами (інформація про батьків і про навчальні групи). Код 200 OK свідчить про успішну обробку. Такий формат відповіді є самозадокументованим і зрозумілим клієнту, а використання стандартних методів і кодів спрощує розробку клієнтської частини.

Отже, монолітна архітектура на основі ASP.NET Core є ефективним рішенням для реалізації серверної частини CRM-системи школи іноземних мов. Використання RESTful API та принципів SOLID дозволяє створити чітко структуровану, ефективну та безпечну серверну частину, яка здатна легко масштабуватися у майбутньому за потреби.

2.3 Реалізація алгоритмів автентифікації, авторизації та безпеки

У сучасних застосунках питання забезпечення безпеки є одними з найважливіших аспектів. Впровадження ефективних механізмів автентифікації та авторизації не лише забезпечує захист персональних даних користувачів, а й дозволяє обмежувати доступ до чутливої інформації. В рамках цієї роботи розглядаються алгоритми автентифікації, авторизації та безпеки, які були реалізовані для серверної частини CRM-системи для школи іноземних мов.

Автентифікація – це процес перевірки достовірності користувача на основі його даних для доступу до системи. У нашій CRM-системі для забезпечення високого рівня безпеки використовуються сучасні методи автентифікації, зокрема хешування паролів за допомогою бібліотеки BCrypt [29]. Цей підхід дозволяє зберігати паролі в захищеному вигляді і не зберігати їх у відкритому вигляді в базі даних.

Для хешування паролю реалізовано клас «PasswordHasher», який відповідає за генерацію хешу паролю та перевірку введеного користувачем паролю з хешованим значенням у базі даних. Використання бібліотеки BCrypt дозволяє отримувати стійкий до атак типу «brute-force» хеш завдяки алгоритмам сольового хешування з додатковими налаштуваннями складності обчислень [40]. Відповідний код, який відповідає за створення та перевірка захешованого паролю має вигляд:

```
namespace CRMSystem.WebAPI.Auth
{
    public class PasswordHasher : IPasswordHasher
    {
        public string Generate(string password) =>
            BCrypt.Net.BCrypt.EnhancedHashPassword(password);

        public bool Verify(string password, string hashedPassword)
=>
            BCrypt.Net.BCrypt.EnhancedVerify(password,
hashedPassword);
    }
}
```

Цей алгоритм забезпечує максимальний рівень захисту при зберіганні паролів користувачів, оскільки кожен хеш створюється з унікальною сіллю, що ускладнює спроби вгадати пароль навіть за наявності великої кількості захешованих паролів.

Авторизація є процесом надання користувачу певних прав доступу до ресурсів або функціоналу системи після того, як він успішно пройшов автентифікацію. У системі, яка буде розроблена, авторизація базується на ролях користувачів, що дозволяє ефективно контролювати доступ до різних частин CRM-системи. У цьому контексті реалізовані три основні політики доступу:

- для адміністраторів (AdminOnly),
- для користувачів (UserOnly)
- для обох ролей (UserOrAdmin).

Код із цими видами користувачів має вигляд:

```
namespace CRMSystem.WebAPI.Auth
{
    public static class AuthorizationPolicies
    {
        public const string AdminOnly = «AdminOnly»;
        public const string UserOnly = «UserOnly»;
        public const string UserOrAdmin = «UserOrAdmin»;
    }
}
```

Кожен користувач під час входу в систему отримує роль, яка визначає його права доступу. За допомогою зазначених політик забезпечується контроль доступу, який обмежує можливості користувачів залежно від їхніх ролей. Наприклад, адміністратор має доступ до всіх функцій системи, тоді як звичайний користувач може лише виконувати обмежені операції.

Json Web Token (JWT) є стандартним методом для авторизації у застосунках. Він дозволяє створювати компактні і безпечні токени, які містять необхідну інформацію для перевірки достовірності користувача та його прав. У системі, що розробляється, для генерації токенів використовується бібліотека «System.IdentityModel.Tokens.Jwt» [29]. JWT генерується після успішної автентифікації користувача і містить в собі основні дані про користувача, що дозволяє на сервері підтвердити його автентичність під час наступних запитів. Токен має обмежений термін дії, після якого його необхідно поновити, що додає додатковий рівень безпеки [30]. Програмний код, який реалізує формування JSON Web Token на основі даних користувача, необхідний для здійснення авторизації у системі представлений у додатку Б.

Важливою частиною реалізації безпеки є впровадження політик доступу до різних частин системи. Завдяки цьому за допомогою механізму авторизації в ASP.NET Core, забезпечується те, щоб користувачі з певними правами могли виконувати тільки дозволені їм дії, а також запобігається доступ до функцій, що не призначені для певних категорій користувачів.

Реалізація алгоритмів автентифікації, авторизації та безпеки є критично важливою для забезпечення захисту даних користувачів та інтеграції в CRM-

систему. Впроваджені механізми, зокрема хешування паролів, генерація та перевірка JWT, а також політики доступу, забезпечують високий рівень безпеки та надійності системи, що є необхідним для її коректної та безпечної роботи під час використання [30].

РОЗДІЛ 3
ТЕСТУВАННЯ, ОПТИМІЗАЦІЯ ТА ВПРОВАДЖЕННЯ BACK-END ЧАСТИНИ CRM

3.1 Розробка API для її інтеграції з front-end частиною

Серверна частина CRM реалізована у вигляді API на ASP.NET Core з багаторівневою архітектурою. Для цього застосовано шаблон «контролер-сервіс-репозиторій», що забезпечує розподіл відповідальності між рівнями:

- репозиторії відповідають за доступ до бази даних,
- сервіси містять бізнес-логіку,
- контролери обробляють HTTP-запити і формують відповіді.

Такий підхід відповідає принципам чистої архітектури і підвищує гнучкість та можливість тестування застосунку. Контролери звертаються до відповідних сервісів, які в свою чергу працюють із репозиторіями та моделями даних. Дані зберігаються в реляційній базі PostgreSQL з використанням Entity Framework Core для абстракції доступу. У проєкті використано шаблон Репозиторію для всіх основних сутностей (студенти, працівники, групи, тощо), що дозволяє ізолювати логіку роботи з даними та зробити код більш підтримуваним. Усі залежності між компонентами впроваджуються через механізм вбудованої інверсії керування (dependency injection), що спрощує заміну реалізацій.

У складі Web API реалізовано низку контролерів, кожен з яких відповідає за певний набір REST-ресурсів. Основні з них, які задіяні у взаємодії з клієнтською частиною:

- AuthController – контролер автентифікації, забезпечує реєстрацію та вхід/вихід користувачів;
- StudentRegistrationController – контролер управління даними студентів (створення, отримання, редагування, видалення студентів, а також імпорт/експорт списку студентів з файлів);

- EmployeeRegistrationController – контролер управління даними працівників (створення, отримання, редагування, видалення працівників, імпорт/експорт списку);

Крім того, в API наявні й інші контролери для підтримки додаткових можливостей: FinanceReportController для формування фінансових звітів; EmailController для відправки електронних повідомлень та перегляду історії розсилок; UserController для перегляду і оновлення профілю поточного користувача; UploadController для завантаження фотографій користувачів.

Кожен контролер має визначений маршрут, відносно якого будується шлях до його ендпоінтів. Наприклад, AuthController має маршрут базового рівня api/auth, StudentRegistrationController – api/student-registration, EmployeeRegistrationController – api/employee-registration тощо. Всі контролери помічені атрибутом [ApiController], що забезпечує автоматичну валідацію вхідних даних і зручне формування помилок відсутніх параметрів.

Контролер AuthController забезпечує три основні ендпоінти для керування автентифікацією користувачів:

- POST /api/auth/sign-up – реєстрація нового користувача. Приймає JSON-об’єкт типу SignUpRequestDto з полями *FullName* (ПІБ), *BirthDate* (дата народження), *Email*, *Username* та *Password* (пароль) і *ConfirmPassword*. Контролер перевіряє коректність даних (через валідатор) і делегує реєстрацію сервісу користувачів. У разі успіху повертає статус 200 OK. Якщо email або ім’я користувача вже зайняті, або дані некоректні – повертає 400 Bad Request з повідомленням про помилку;
- POST /api/auth/sign-in – вхід (авторизація) користувача. Приймає SignInRequestDto з полями *Username* та *Password*. Перевіряє валідність даних і викликає метод сервісу авторизації, який звіряє пароль та генерує JWT-токен. Цей JWT-токен контролер повертає клієнту у вигляді HttpOnly cookie з ім’ям jwt. У відповіді встановлюється cookie (SameSite=Strict, HttpOnly, Secure), а результатом є 200 OK. Некоректні дані доступу призводять до 400 Bad Request з текстом помилки.

- POST /api/auth/sign-out – вихід із системи. Цей ендпоінт видаляє cookie jwt з токеном у відповіді (очищує його) та повертає 200 OK. Використовується для завершення сеансу авторизованого користувача. Доступ до цього методу дозволений тільки автентифікованим користувачам, про що свідчить атрибут [Authorize(Policy = UserOrAdmin)] у контролері.

Контролер StudentRegistrationController надає CRUD-функціонал для сутності Student та можливості імпорту/експорту:

- POST /api/student-registration – створення нового студента. Приймає дані студента у форматі JSON (RegisterStudentDto), що містять особисту інформацію студента та пов’язані дані навчання. Після валідації контролер викликає StudentRegistrationService для створення відповідних записів у базі. У разі успішного створення повертається 200 OK з даними нового студента. Вимагає роль Admin для додавання студентів;
- GET /api/student-registration/{id} – отримання даних студента за його ID. Повертає 200 OK з JSON-об’єктом RegisterStudentDto з інформацією про студента. Якщо студента не знайдено – повертається 404 Not Found. Доступ дозволений як адміністратору, так і звичайному користувачу (політика UserOrAdmin), тобто авторизований персонал школи може переглядати інформацію студентів.
- GET /api/student-registration – отримання списку всіх студентів. Може приймати необов’язкові параметри фільтрації. Повертає масив студентів у форматі JSON. Також доступний для ролей User і Admin.
- PUT /api/student-registration/{id} – оновлення інформації студента за ID. Приймає JSON з оновленими даними такого ж формату, як при створенні. Перевіряє валідність і звертається до сервісу для внесення змін. Якщо успішно – 200 OK з оновленими даними, якщо студента не знайдено – 404 Not Found. Вимагається роль Admin.
- DELETE /api/student-registration/{id} – видалення студента за ID. Без тіла запиту, видаляє відповідний запис та пов’язані з ним залежні дані. Повертає 204 No Content у разі успіху. Вимагається роль Admin.

- POST /api/student-registration/import-file – імпорт списку студентів з файлу. Дозволяє адміністратору завантажити файл формату CSV, Excel, JSON з переліком великої кількості студентів для реєстрації. Файл передається як частина форми multipart/form-data – поле з файлом. Контролер через сервіс StudentFileService автоматично визначає формат і читає дані. На виході повертає 200 OK з масивом імпортованих даних студентів або повідомляє про помилку 400, якщо файл порожній чи дані некоректні. Цей метод суттєво спрощує перенесення великої кількості студентів у систему із зовнішніх джерел.
- POST /api/student-registration/export-file – експорт списку студентів у файл. Дозволяє адміністратору отримати актуальний список студентів у вигляді файлу формату формату CSV, Excel, JSON. У запиті вказується бажане ім’я файлу через параметр fileName. У відповіді повертається сам файл з відповідним Content-Type і заголовком Content-Disposition для завантаження. Цей функціонал спирається на сервіс StudentFileService, який збирає дані всіх студентів та генерує файл потрібного формату. Вимагається роль Admin.

Контролер EmployeeRegistrationController працює аналогічно до StudentRegistrationController, але для сутності Employee (працівник школи). Він містить ті ж самі ендпоінти: створення працівника (POST /api/employee-registration), отримання одного працівника (GET /api/employee-registration/{id}) або списку (GET /api/employee-registration), оновлення (PUT /api/employee-registration/{id}), видалення (DELETE /api/employee-registration/{id}), імпорт працівників з файлу (POST /api/employee-registration/import-file) та експорт списку працівників (POST /api/employee-registration/export-file). Правила доступу такі ж: тільки адміністратор може створювати/редагувати/видаляти та імпортувати/експортувати, а переглядати дані дозволено й звичайним авторизованим користувачам. Дані працівників включають ПІБ, дату народження, контактні дані, посаду та зарплату. При імпорті з файлу також підтримуються формати CSV/Excel/JSON; усі записи перевіряються валідатором перед додаванням.

У проєкті реалізовано JWT-аутентифікацію для захисту API. Після успішного входу користувач отримує JSON Web Token, який містить його ідентифікатор та роль у вигляді запитів (claims). Цей JWT-токен зберігається на клієнті в HttpOnly cookie, щоб запобігти доступу до нього через JavaScript та атакам XSS, і має атрибут SameSite=Strict, аби токен не відправлявся на сторонні домени, з метою захисту від CSRFhttps://community.auth0.com/t/storing-access-tokens-in-httponly-secure-samesite-strict-cookie/132422. На стороні сервера налаштовано Middleware аутентифікації JWT Bearer, де кожен запит до захищених ендпоінтів проходить перевірку токена. У конфігурації аутентифікації задаються параметри валідації токену, де здійснюється перевірка підпису на основі секретного ключа з налаштувань, час життя токена тощо. Також встановлено подію OnMessageReceived, що дозволяє витягувати токен з cookie jwt автоматично для зручності.

Для авторизації доступу використано рольову модель з політиками. У системі визначено дві ролі користувачів – Admin (адміністратор) і User (звичайний користувач). Ролі зберігаються в базі даних (таблиця Roles) і пов’язані з обліковим записом користувача (поле RoleId в UserEntity). При реєстрації нового користувача йому автоматично присвоюється роль User, окрім випадку, коли в системі ще немає жодного користувача – тоді першому зареєстрованому задається роль Admin. Цим забезпечується наявність хоча б одного адміністратора в системі без ручного втручання.

Політики авторизації налаштовані так, щоб обмежити доступ до вразливих ендпоінтів. Зокрема, визначено три політики:

- AdminOnly – доступ лише для користувачів з роллю Admin.
- UserOnly – доступ лише для ролі User.
- UserOrAdmin – доступ для авторизованих користувачів будь-якої з двох ролей.

Ці політики застосовані через атрибути [Authorize] у контролерах. Наприклад, методи створення, редагування та видалення студентів/працівників помічені як [Authorize(Policy = «AdminOnly»)], тому лише адміністратор може їх

виконувати. Натомість отримання списків (GetAll) чи окремих записів (GetById) студентів/працівників має [Authorize(Policy = «UserOrAdmin»)], що дозволяє і звичайним користувачам переглядати ці дані без права модифікації. Контролер AuthController дозволяє неавторизованим виконувати sign-up та sign-in, тоді як sign-out має [Authorize(UserOrAdmin)]. Таке налаштування гарантує захищеність API, де без JWT-токена користувач отримає статус 401 Unauthorized при спробі доступу до більшості ресурсів.

У додатку В наведено таблицю В.1 з основними доступними ендпоінтами серверного API та їх призначенням. Як видно з таблиці, більшість операцій додавання/редагування/видалення даних доступні лише адміністраторам, у той час як перегляд даних дозволений також і звичайним користувачам. Це відповідає бізнес-логіці CRM, коли викладачі можуть переглядати списки студентів та іншу інформацію, але лише адміністратор має право вносити зміни або масово імпортувати/експортувати дані.

Серверна частина працює з низкою сутностей, що відображають предметну область школи. У базі даних визначені таблиці Students, Employees, Persons, Contacts, Contracts, Groups, Languages, LessonDays, StudentGroups та ін. Використовується підхід Code First, де кожна сутність описана класом C# з відповідними властивостями і зв’язками напряму або через проміжні таблиці. Наприклад, студент має посилання на PersonEntity, де зберігається його ПІБ і дата народження, може мати посилання на батьків (Parent, також PersonEntity), а також пов’язаний з навчальними групами через таблицю-зв’язку StudentGroupEntity. Контракт (ContractEntity) містить дату підписання та суму оплати за навчання, пов’язаний зі студентом. Група (GroupEntity) містить назву та мову навчання, пов’язана з днями занять (LessonDayEntity, наприклад, «Понеділок-середа») через GroupLessonDayEntity, а також з студентами через StudentGroupEntity. Така структура дозволяє гнучко моделювати навчальний процес (групи, розклад занять, запис студентів на групи, відстеження оплат через контракти тощо).

Для ілюстрації загальної логіки роботи системи на рис. 3.1 показано спрощену блок-схему користувацького інтерфейсу.

Рисунок 3.1 – Схема роботи користувача з системою

Після авторизації на головному вікні доступні вкладені розділи «Адміністрування студентів», «Адміністрування працівників», «Перегляд розкладу» (групи і заняття), «Повідомлення» (електронні розсилки), «Календар подій», «Звітність школи» (фінансові звіти), «Таск-менеджер», «Зарплатний лист» та «Про систему». Вкладка профілю користувача містить функції роботи з власним обліковим записом (перегляд профілю, зміна пароля, завантаження фото) і вихід із системи.

З рисунку 3.1 можна побачити, що після запуску програми користувач проходить авторизацію/реєстрацію. Далі – головне вікно, де користувач має доступ до різних вкладок системи. Програма завершує роботу після натискання кнопки «Вихід із системи», яка міститься у профілі користувача.

3.2 Тестування та оптимізація back-end частини

Для забезпечення якості та стабільності роботи серверної частини було застосовано декілька рівнів тестування. Основний акцент зроблено на юніт-тестуванні контролерів і сервісів. Завдяки тому, що логіка побудована з

використанням інтерфейсів і dependency injection, контролери можна тестувати ізольовано, що дає змогу використовувати мок-об’єкти замість реальних залежностей. У проєкті використано фреймворк xUnit для написання тестів і бібліотеку Moq для створення моків (підроблених об’єктів) сервісів та інших залежностей. Для прикладу розроблено набір тестів у класі AuthControllerTests (юніт-тести для AuthController), програмну реалізацію якого представлено в додатку Г. Нижче наведено опис, як реалізовано тестування різних сценаріїв на основі цього контролера:

- тест успішної реєстрації (SignUp_ReturnsOk_WhenValid) перевіряє, що при наданні коректних даних реєстрації користувача метод SignUp контролера повертає результат 200 OK. У цьому тесті за допомогою Moq створюється мок-сервіс UserService та мок-валідатор IValidatorFactory. Валідатор налаштовано повертати true (валідні дані), а виклик UserService.SignUp(...) не видає винятків. Після виклику контролера перевіряється, що результат є OkResult, а метод сервісу SignUp був викликаний рівно один раз з очікуваними параметрами;
- тест неуспішної реєстрації з некоректними даними (SignUp_ReturnsBadRequest_WhenInvalid) імітує ситуацію, коли вхідні дані заповнені некоректно або не пройшли валідацію. Мок-валідатор налаштований повертати false і повідомлення про помилку. Очікувана поведінка: метод контролера має повернути 400 Bad Request і передати клієнту саме це повідомлення про помилку. Тест перевіряє, чи повернутий об’єкт є BadRequestObjectResult і чи містить в собі потрібний текст помилки;
- тест обробки винятку сервісу при реєстрації (SignUp_ReturnsBadRequest_WhenServiceThrows) симулює сценарій, коли під час реєстрації сервіс користувачів видає виняток, наприклад при спробі створити користувача з уже наявним email. Moq налаштовує userService.SignUp викидати InvalidOperationException з повідомленням «User already exists». Контролер повинен перехопити цей виняток і перетворити його на BadRequest з тим самим повідомленням. Тест перевіряє, що результат – BadRequestObjectResult з текстом «User already exists»;

- тест успішного входу (SignIn_ReturnsOk_AndSetsJwtCookie) перевіряє, чи при коректних логіні/паролі контролер AuthController.SignIn повертає Ok і встановлює JWT у cookie. У тесті мок-валідатор схвалює дані, а userService.SignIn налаштований повертати певний рядок-токен, наприклад, «mocked-jwt». Після виклику методу контролера очікується OkResult, а також перевіряється, що в об’єкті HttpContext.Response з’явився Cookie з назвою «jwt» і значенням, яке містить «mocked-jwt»;
- тест неуспішного входу з порожніми полями (SignIn_ReturnsBadRequest_WhenInvalid) аналогічно до випадку реєстрації, перевіряє відмову в авторизації, якщо логін/пароль не передано. Валідатор повертає помилку «Invalid credentials», контролер має повернути BadRequest з цим текстом;
- тест виходу (SignOut_DeletesJwtCookie_AndReturnsOk) перевіряє, чи під час виклику виходу контролер видаляє JWT-cookie і повертає Ok. Перед викликом методу в контекст контролера встановлюється підроблений користувач (ClaimsPrincipal) з Claim «username», щоби контролер міг залогувати ім’я. Після SignOut перевіряється, що результат – Ok, а в колекції Cookies відповіді відсутній токен.

Саме ці тести охоплюють базові сценарії для AuthController, інші контролери можна тестувати подібним чином. Наприклад, для StudentRegistrationController варто перевірити, чи:

- при коректних даних студента Create повертає Ok із створеним студентом;
- при некоректних – повертає BadRequest з повідомленням валідатора;
- метод Delete повертає NoContent і викликає сервіс видалення;
- метод ImportFile повертає Ok із даними або BadRequest для порожнього файлу тощо.

Окрім автоматизованих тестів, ручне тестування проводилося з використанням інструментів Postman та Swagger UI. Під час розробки проєкт було сконфігуровано для автоматичного генерування Swagger-документації через

сервіси AddEndpointsApiExplorer() і AddSwaggerGen() у Program.cs. У режимі розробки Swagger UI доступний за адресою */swagger*, що дозволяє розробнику інтерактивно виконувати запити до всіх ендпоінтів, переглядати структуру DTO та перевіряти відповіді сервера. Swagger спростив перевірку правильності маршрутизації, моделей запиту/відповіді та статус-кодів, а також слугував живою документацією API для front-end розробників.

У процесі розробки враховані аспекти продуктивності та масштабованості серверної частини. Усі операції доступу до даних реалізовані асинхронно з використанням async/await при зверненні до бази через EF Core, при виконанні файлових I/O тощо. Це забезпечує обслуговування запитів, яке не блокується, і кращу масштабованість при збільшенні числа одночасних користувачів. Також застосовано принцип «не більше необхідного» при отриманні даних – репозиторії надають методи GetAll та GetById, але для вибірки великих об’ємів даних можуть використовуватися фільтри, щоб зменшити трафік.

У процесі генерації фінансового звіту (FinanceReportService.GenerateReportAsync) здійснюються агрегатні операції (сума платежів, сума зарплат) над списками студентів, контрактів і працівників. Теоретично, це можна оптимізувати за допомогою використання агрегатних запитів безпосередньо на рівні бази даних (SQL SUM ... GROUP BY). У поточній реалізації дані спочатку завантажуються в пам’ять, а потім зведення рахується через LINQ. Для невеликої кількості записів це не створює проблем, але при масштабуванні варто буде оптимізувати цей фрагмент, щоб навантаження виконувалося СУБД.

У back-end частині впроваджено докладне логування подій із використанням бібліотеки Serilog. Конфігурація Serilog виконується на старті в Program.cs методом AddSerilogLogging(), який читає налаштування з файлу *appsettings.json* та додає логер у DI-контейнер. Логи записуються як в консоль для відладки під час розробки, так і в файл логів, який зберігає історію роботи застосунку, що важливо на продуктивному сервері. У коді контролерів і сервісів додано повідомлення різних рівнів: інформаційні про успішні операції,

попередження про некоректні вхідні дані або відсутність записів, помилки про виняткові ситуації. Логування мінімально впливає на продуктивність, проте дає значні переваги в підтримці системи.

Усі критично важливі точки вводу захищені механізмом серверної валідації. Замість стандартних атрибутів DataAnnotations, у проєкті реалізовано власний ValidatorFactory та набір валідаторів для різних типів запитів DTO. Це дозволяє виконувати більш гнучкі перевірки, у тому числі між полями. Наприклад, StudentValidator перевіряє: якщо встановлено прапорець *IsParentRegister*, то обов'язково має бути заповнено *ParentFullName*; вимога обов'язкового та правильного заповнення ПІБ, дати народження, межі допустимості класу, наявність телефону, email, вибраної мови, рівня, групи, днів занять, а також коректність дати укладення договору. EmployeeValidator перевіряє ПІБ, дату народження, телефон, email, посаду та невід'ємність зарплати. UserValidator охоплює логіку перевірки при реєстрації користувача: мінімальна довжина імені та логіна, валідність email через регулярний вираз/формат MailAddress, вимоги до пароля – мінімальна довжина, наявність цифр/спеціальних символів, збіг поля ConfirmPassword), під час входу, що логін і пароль не порожні, під час оновлення профілю та зміні пароля, що старий пароль правильний, а новий достатньо складний і збігається з підтвердженням. Усі ці перевірки виконуються на стороні сервера до здійснення основної логіки, і якщо якісь із умов не виконані – контролер одразу повертає 400 Bad Request з повідомленням про те, яке поле некоректне. Такий підхід гарантує цілісність даних у базі та безпеку.

Щодо безпеки і паролів користувачів, то вони ніколи не зберігаються в базі у відкритому вигляді. Під час реєстрації або зміні пароля використовується сервіс PasswordHasher, який застосовує алгоритм BCrypt для хешування пароля. Бібліотека BCrypt.Net генерує крипостійкий хеш і сіль для кожного пароля [39]. У базі даних в полі PasswordHash зберігається саме хеш. Під час входу введений користувачем пароль також хешується і порівнюється з хешем у БД

(PasswordHasher.Verify). BCrypt є стійким до brute-force і має високий час обробки, що додає рівень захисту від підбору паролів [40].

На додаток до авторизації ролей, можна відзначити захист від CSRF завдяки використанню JWT у cookie SameSite=Strict, а також загалом HTTPS для всіх з'єднань. Вразливості типу SQL-ін'єкцій в основному нейтралізовані використанням Entity Framework, де ORM будує запити за шаблонами параметрів. Також введено обмеження на типи файлів при завантаженні фото: UploadController приймає лише зображення JPEG, PNG, GIF певного розміру.

3.3 Аналіз результатів розробки та перспективи розвитку

У ході розробки серверної частини CRM-системи для мовної школи вдалося реалізувати всі основні заплановані функції серверної логіки. Тож далі описано детальніше про всі функціональні можливості.

1. Реєстрація, вхід та вихід користувачів. Система підтримує створення нових облікових записів, що дозволяє новим користувачам отримати доступ до системи. Впроваджено механізм входу (login) із застосуванням JWT-токенів, що забезпечує безперервну авторизацію користувача при наступних запитах. Також реалізовано можливість безпечного виходу (logout), що знищує сесію токена. При першій реєстрації автоматично призначається роль адміністратора, що спрощує початкову ініціалізацію системи. Надалі адміністратор може створювати інші облікові записи персоналу або вони можуть реєструватися самостійно, отримавши роль User.

2. Адміністрування студентів та працівників (CRUD-операції). Адміністратор має повний інтерфейс для керування інформацією про студентів і персоналу школи та може додавати дані про них. Є можливість переглядати ці списки, фільтрувати їх за певними критеріями, відкривати деталі конкретної особи. Також підтримується редагування записів – наприклад, оновлення контактної інформації студента або зміна посади працівника. Видалення записів видаляє особу з системи.

3. Імпорт та експорт даних (CSV/Excel/JSON). Для підвищення ефективності роботи адміністратора реалізовано масовий імпорт студентів і працівників з файлів. Якщо школа вже має базу в табличному форматі (Excel), вона може просто експортувати ці дані у форматах CSV, Excel, JSON і завантажити їх у CRM, замість того щоб вводити дані про кожного студента вручну. В іншому випадку можна будь-коли отримати актуальний список студентів чи викладачів, експортувавши його у файл, що полегшує підготовку звітів, списків груп, контактних даних для розсилок тощо. Ця функціональність суттєво економить час та знижує ризик помилок під час перенесення даних.

4. Фінансова звітність. Система надає можливість генерувати агреговані фінансові звіти по школі. На даному етапі реалізовано зведення: загальна кількість студентів, очікуваний дохід – сума всіх платежів по навчальних контрактах, які мали б бути сплачені), фактичний дохід – сума оплат від тих студентів, які сплатили за навчання, кількість працівників, загальна сума нарахованих зарплат і розрахунок чистого прибутку. Такий звіт доступний адміністратору одним запитом (GET /api/finance-report/summary). Він допомагає керівництву школи оцінити фінансовий стан у будь-який момент – наприклад, бачити, скільки коштів очікується від студентів, скільки вже отримано, які витрати на персонал, і підсумковий прибуток. Це важливий модуль CRM, адже фінансова прозорість дозволяє приймати зважені рішення.

5. Поштові розсилки та повідомлення. У системі реалізовано простий модуль розсилки електронних повідомлень. Авторизований користувач може надіслати листа одному чи кільком отримувачам через ендпоінт /api/email/send, задавши тему і текст повідомлення в об’єкті SendMessageDto. Використовується SMTP-клієнт, який налаштовується через конфігурацію SmtpSettings, та бібліотека MailKit для відправки email. Копія кожного надісланого повідомлення зберігається в базі, тому є можливість переглянути історію розсилок: /api/email/messages повертає список всіх повідомлень, а /api/email/messages/{receiverId} – відфільтрований список за отримувачем. Цей функціонал дозволяє швидко інформувати студентів про оплату або новини, а

також сповіщати працівників про збори, зміни тощо безпосередньо в CRM. У майбутньому його можна розширити до масових розсилок або шаблонізованих листів.

6. Профілі користувачів. Кожен користувач системи має свій профіль, який він може переглянути і відредагувати. Ендпоінт /api/user/me надає інформацію про поточного залогіненого користувача: ПІБ, дату народження, email, ім’я користувача та роль. Через PATCH-запити на той самий URL користувач може оновити свої особисті дані. Окремо реалізовано зміну пароля: PATCH /api/user/me/change-password очікує старий пароль та новий, який вводиться двічі для підтвердження, в UpdateUserPasswordDto. Back-end перевіряє правильність старого пароля і відповідність нового вимогам, після чого зберігає новий хеш пароля в базі. Як підсумок користувачі самостійно підтримують актуальність своїх даних і можуть забезпечити безпеку свого облікового запису.

7. Додавання фотографії до профілю. Додатковою функцією є можливість встановлювати для профілю користувача фотографію. Через ендпоінт /api/users/upload-photo користувач може завантажити зображення. Цей запит очікує multipart/form-data з полем File (типу IFormFile). Back-end перевіряє тип файлу (JPEG, PNG, GIF) і не приймає порожні файли. Файл зберігається на сервері у локальне сховище через LocalFileStorageService у спеціальну директорію, а шлях до нього зберігається у полі ImageUrl відповідного до користувача. Якщо той самий файл вже було завантажено раніше, сервіс може повернути створену URL замість створення дублікату. У відповіді після успішного завантаження повертається посилання URL на зображення. Надалі front-end може відображати цю світлину в профілі. Ця функція покращує зручність використання системи і додає візуальну ідентифікацію записів працівників чи учнів у інтерфейсі.

Завдяки переліченим можливостям, розроблене серверне рішення повністю покриває потреби мовної школи в управлінні своїми основними процесами: облік учнів і персоналу, комунікація з ними, фінансовий контроль,

базові адміністративні завдання. Архітектура системи модульна, тому за необхідності можна додавати нові підсистеми без суттєвої переробки створених.

Розроблений back-end для CRM-системи продемонстрував ефективність як з точки зору функціональності, так і якості програмного коду. По-перше, рішення відповідає бізнес-вимогам, де усі заявлені функції були успішно реалізовані і працюють відповідно до очікувань. Користувачі отримали інтуїтивно зрозумілий набір можливостей для виконання повсякденних задач, а керівництво – інструменти контролю. По-друге, архітектура серверної частини виявилася гнучкою і підтримуваною. Виділення шарів контролерів, сервісів і репозиторіїв спростило розробку і тестування, де кожен компонент має чітко визначену зону відповідальності. Наприклад, можна змінити спосіб зберігання файлів просто реалізувавши інший FileStorageService, не змінивши логіку UploadController. Така модульність також полегшує виявлення і виправлення помилок – логування на рівні кожного контролера/сервісу дає повну картину того, що відбувається в системі, і де стався збій.

Впровадження безпеки – JWT, валідація, хешування, зробило систему захищеною від основних загроз. Наприклад, перевірено, що неавторизовані запити не отримають доступу до даних, а спроби порушити цілісність блокуються валідаторами. Це свідчить про надійність рішення у використанні.

Продуктивність системи на тестових об’ємах даних є високою: запити виконуються швидко, імпорт декількох сотень записів з файлу займає долі секунд. Використання сучасного ASP.NET Core 8 і ефективних бібліотек EF Core, AutoMapper, MailKit позитивно впливає на швидкодію і оптимальність використання ресурсів. До того ж, горизонтальне масштабування можливе без змін коду, оскільки сервіс переважно не зберігає стан. Тож реалізоване серверне рішення є ефективним, безпечним і готовим до використання в реальних умовах. Воно створює міцний фундамент, на базі якого можна розширювати функціонал системи.

Під час аналізу і тестування були виявлені потенційні функції та удосконалення, які можна додати до системи в подальшій роботі. Наразі система

має дві ролі – Admin та User. У майбутньому можна додати більше градацій: наприклад, роль «Teacher» (викладач) – з доступом лише до своїх груп студентів та можливістю виставляти оцінки чи відмічати відвідування; роль «Student» – потенційно, щоб самі студенти могли заходити в систему, аби переглядати свій прогрес або розклад. Додавання ролей потребуватиме лише внесення їх у таблицю Roles, налаштування політик і розподілу прав у контролерах – архітектура вже підтримує авторизацію на основі політик.

Зокрема, можна реалізувати логіку оплати: інтегрувати платіжний шлюз для онлайн-оплати навчання, автоматично змінювати прапорець «IsPaid» у студента при надходженні коштів, генерувати квитанції. Також варто розширити модуль розкладу занять – додати ендпоінти для створення та редагування навчальних груп, призначення викладачів на групи, планування розкладу. Task Manager може бути реалізований для відстеження завдань співробітників, зберігати ці задачі в базі і відмічати їх виконання. Модуль зарплат теж доцільно додати: автоматичний розрахунок заробітної плати викладачів на основі проведених занять чи відпрацьованих годин, формування «зарплатних листів» по кінцю місяця.

CRM могла б обмінюватися даними з зовнішніми системами. Для прикладу, інтеграція з сервісом розсилок SMS або Viber, щоб окрім email надсилати повідомлення студентам про заняття чи борги по оплаті. Або інтеграція з бухгалтерською системою для передачі фінансових даних (доходи/витрати) напряму в облікову програму школи. Технічно це можна зробити через додаткові API-виклики зовнішніх сервісів з back-end або через експорт даних у встановленому форматі.

У випадку зростання кількості користувачів варто буде впровадити кешування для часто запитуваних даних з використанням Redis або внутрішнього MemoryCache, щоб зменшити звернення до бази. Можна реалізувати посторінкову вибірку для довгих списків студентів, якщо їх стане дуже багато. Крім того, варто подумати про рознесення сервісів: наприклад, модуль відправки email винести в окремий фоновий сервіс, який буде приймати завдання через

чергу, що забезпечить асинхронність розсилок і нульовий вплив на швидкість відповіді основного API.

У перспективі, якщо систему будуть використовувати треті сторони, можна опублікувати Swagger-документацію онлайн або підготувати керівництво розробника щодо використання API. Також важливо підтримувати актуальність безпеки – оновлювати алгоритми хешування, впроваджувати multi-factor authentication для входу адміністратора тощо.

У підсумку створена серверна частина охоплює ключові потреби CRM-системи школи іноземних мов і є готовим рішенням. Водночас, її архітектура і технологічний стек відкривають широкі можливості для подальшого розвитку. Реалізація нових ролей та модулів не потребуватиме кардинальної перебудови, що є важливою перевагою проєкту. Отже, розроблена серверна частина не лише вирішує поставлені завдання, але й закладає основу для масштабування системи разом із зростанням бізнесу школи.

ВИСНОВКИ

У процесі виконання бакалаврської роботи було розглянуто та реалізовано низку теоретичних і практичних завдань, які дозволили створити надійну, функціональну та адаптовану до потреб освітнього закладу серверну частину програмного продукту.

Проведене дослідження підтвердило високу актуальність теми, що зумовлена загальною цифровою трансформацією бізнесу й освіти, а також показало, що школи іноземних мов, як і інші приватні освітні заклади, зіштовхуються з низкою проблем організаційного характеру, а саме: необхідністю обліку та моніторингу відвідуваності занять, ведення фінансової звітності, комунікації з клієнтами, управління навчальними планами тощо. Автоматизація цих процесів можлива за рахунок впровадження CRM-систем, проте більшість наявних на ринку рішень орієнтовані на бізнес-середовище або мають зайву складність для потреб невеликих навчальних установ. Тож на відміну від універсальних CRM-систем, запропоноване рішення адаптивної системи враховує специфіку освітнього середовища та основні вимоги до функціоналу: створення груп і розкладів, база студентів, контроль оплат, моніторинг відвідуваності, звітність та взаємодія з користувачами системи – адміністраторами, викладачами та ін.

У ході проєктування архітектури серверної частини було обґрунтовано доцільність використання сучасних технологій та інструментів, таких як Entity Framework з підходом Code First для проєктування бази даних, REST API для взаємодії між клієнтською та серверною частинами, а також реалізація авторизації та автентифікації з використанням JWT та алгоритмів гешування. Застосування цих технологій дало змогу створити масштабовану архітектуру, орієнтовану на безпечну та ефективну обробку запитів.

У проєкті реалізовано логіку контролю доступу, що дозволяє обмежувати функціональні можливості користувачів відповідно до їх ролей. Окрему увагу надано захисту персональних даних, що є критично важливим у контексті роботи

із студентами. Було також реалізовано тестування API та оптимізацію запитів до бази даних, що дозволило досягти стабільної роботи системи при одночасному доступі кількох користувачів.

Результатом роботи є створена серверна частина CRM-системи, яка може бути застосована у діяльності реальних шкіл іноземних мов. Запропоноване рішення є гнучким, розширюваним і придатним до подальшого розвитку, зокрема додавання інших ролей користувачів, розширення API і бізнес-логіки, інтеграції з іншими сервісами, рефакторингу та оптимізації, документуванню та підтримці. У перспективі ця система може бути адаптована і для інших типів освітніх закладів – мовних курсів, центрів підготовки до складання іспитів тощо.

Отже, виконана робота поєднує теоретичне обґрунтування та практичну реалізацію актуального програмного рішення, що відповідає специфіці приватної освітньої установи. Здобуті результати мають значний практичний потенціал та можуть бути використані як основа для подальших наукових розробок або комерційного впровадження.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Жигалкевич Ж.М., Залуцький Р.О. Improving the quality of business processes of enterprises based on digitization. Journal of Strategic Economic Research. 2023. №2(13). P. 84–93. URL: https://doi.org/10.30857/2786-5398.2023.2.9

2. Янко А.С., Шахно В.О. Аспект інформаційної безпеки в сучасних CRM-системах в епоху діджиталізації економіки та бізнесу. Таврійський науковий вісник. Серія: Технічні науки. 2022. №4. С. 28-33. URL: https://doi.org/10.32851/tnv-tech.2022.4.4

3. Жосан Г. В., Кириченко Н. В. Управління цифровізацією бізнес-процесів діяльності підприємства. Economic Synergy. 2022. Вип. 4 (6). С. 82–91. URL: https://doi.org/10.53920/ES-2022-4-6

4. Golmohammadi A., Zhang M., Arcuri, A. .NET/C# instrumentation for search-based software testing. Software Qual J. 2023. Vol. 31. P. 1439–1465. URL: https://doi.org/10.1007/s11219-023-09645-1

5. Troelsen A., Japikse P. Exploring Entity Framework Core. In: Pro C# 10 with .NET 6. Apress, Berkeley, CA. 2022. URL: https://doi.org/10.1007/978-1-4842-7869-7_22

6. Голубничий Д. Ю., Колесник М. Ю. Використання платформи .NET для розробки застосунків. Поліграфічні, мультимедійні та web-технології: тези доп. IX Міжнар. наук.-техн. конф. Т. 1.(Харків, 14-18 травня 2024 р.). Харків: ТОВ «Друкарня Мадрид», 2024. С. 81-82. URL: https://openarchive.nure.ua/handle/document/26683

7. Dalbard A., Isacson J. Comparative study on performance between ASP.NET and Node.js Express for web-based calculation tools: thesis. 2021. URL: http://urn.kb.se/resolve?urn=urn:nbn:se:hj:diva-53664

8. Dachepally R. Implementing Cross-Platform APIs with Node.js, Python and Java. Journal of Mathematical & Computer Applications. 2023. Volume 2(3). P. 1–3. URL: https://doi.org/10.47363/jmca/2023(2)e162

9. Chakraborty Sudip and Aithal P. S. CRUD Operation on WordPress Database Using C# And REST API. International Journal of Applied Engineering and Management Letters (IJAEML). 2023, №7(4). P. 130-138. URL: http://dx.doi.org/10.2139/ssrn.4729159

10. Bonteanu A. M., Tudose C. Performance Analysis and Improvement for CRUD Operations in Relational Databases from Java Programs Using JPA, Hibernate, Spring Data JPA. Applied Sciences. 2024. Vol. 14, no. 7. URL: https://doi.org/10.3390/app14072743

11. Dhalla H. K. A Performance Comparison of RESTful Applications Implemented in Spring Boot Java and MS.NET Core. Journal of Physics: Conference Series. 2021. Vol. 1933, no. 1. URL: https://doi.org/10.1088/1742-6596/1933/1/012041

12. Richard G. A. Framework Comparison: .NET and Laravel: Undergraduate Honors Thesis, University of Nebraska-Lincoln. 2022.

13. Vadlamani V. Postgres Cluster and Database Backup. In: PostgreSQL Skills Development on Cloud. Apress, Berkeley, CA. 2024. URL: https://doi.org/10.1007/979-8-8688-0817-3_8

14. Backup and restore in Azure Database for PostgreSQL flexible server. URL: https://learn.microsoft.com/en-us/azure/postgresql/flexible-server/concepts-backup-restore (останнє звернення: 10.04.2025)

15. Yijie Weng and Jianhao Wu. Database management systems for artificial intelligence: Comparative analysis of postgre SQL and MongoDB. World Journal of Advanced Research and Reviews. 2025. Vol. 25(02). P. 2336-2342. URL: https://doi.org/10.30574/wjarr.2025.25.2.0586

16. Dhanagari, M. R. Improving Customer Experience with MongoDB: Personalization and Recommendations. American Journal of Technology. 2025. Vol. 4(1). P. 55–90. URL: https://doi.org/10.58425/ajt.v4i1.354

17. Kilavo H., Mrutu S. I., Dudu R. G. Securing Relational Databases against Security Vulnerabilities: A Case of Microsoft SQL Server and PostgreSQL. Journal of

Applied Security Research. 2021. P. 1–15. URL: https://doi.org/10.1080/19361610.2021.2006032

18. Wadhwa M. A comparative study between ADO.NET and Entity Framework. International Journal of Advanced Scientific and Technical Research. 2018. Vol. 6, no. 7. URL: https://doi.org/10.26808/rs.st.i7v6.17

19. Бабак О., Ісак Л. Архітектура комп'ютерних систем. Перспективи розвитку. Grail of Science, 2023, №26. С.258–260. URL: https://doi.org/10.36074/grail-of-science.14.04.2023.046

20. Blinowski G., Ojdowska A., Przybylek A. Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation. IEEE Access. 2022. Vol. 10. P. 20357–20374. URL: https://doi.org/10.1109/access.2022.3152803

21. Праворська Н., Грига С. Методи реалізації мікросервісних архітектур: переваги та недоліки, впровадження та тестування при розробці програмних застосунків. Herald of Khmelnytskyi National University. Technical sciences. 2024. Т. 335, № 3(1). С. 404–408. URL: https://doi.org/10.31891/2307-5732-2024-335-3-55

22. Gos K., Zabierowski W. The Comparison of Microservice and Monolithic Architecture. 2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design *(MEMSTECH)* (Lviv, Ukraine, 22–26 April 2020). Lviv, 2020. URL: https://doi.org/10.1109/memstech49584.2020.9109514

23. Choudhary A. Exploring Docker Alternatives. In: When Docker Meets Java. Apress, Berkeley, CA. 2025. URL: https://doi.org/10.1007/979-8-8688-1300-9_8

24. Design and Implementation of Flutter Based Multi-Platform Docker Controller App / A. Saxena et al. 2024 International Conference on Decision Aid Sciences and Applications (DASA), Manama, Bahrain, 11–12 December 2024. 2024, P. 1–6. URL: https://doi.org/10.1109/dasa63652.2024.10836516

25. Choudhary A. Learning Advanced Docker Concepts. In: When Docker Meets Java. Apress, Berkeley, CA. 2025, P. 65–85. URL: https://doi.org/10.1007/979-8-8688-1300-9_4

26. Naylor L. Using Entity Framework Code First with an Existing Database. In: ASP.NET MVC with Entity Framework and CSS . Apress, Berkeley, CA. 2016. URL: https://doi.org/10.1007/978-1-4842-2137-2_11

27. Arjun Singh Saud, & Tenish Shrestha. Template based Program synthesis with Declarative Programming for RESTful APIs. Recent Trends in Cloud Computing and Web Engineering, 2024, 7(1), 10–20. URL: https://doi.org/10.5281/zenodo.13883634

28. Aravinda A Kumar, Divya TL. Security measures implemented in RESTful API Development. Open Access Research Journal of Engineering and Technology. 2024. Vol. 7, no. 1. P. 105–112. URL: https://doi.org/10.53022/oarjet.2024.7.1.0042

29. Shivam Gupta M. Secure API Gateway with Rate Limiting and JWT Authentication. International Journal for Research in Applied Science and Engineering Technology. 2025. Vol. 13, no. 4. P. 3559–3562. URL: https://doi.org/10.22214/ijraset.2025.68953

30. Xu B. *et al.* JWTKey: Automatic Cryptographic Vulnerability Detection in JWT Applications. In: Tsudik, G., Conti, M., Liang, K., Smaragdakis, G. (eds) Computer Security – ESORICS 2023. ESORICS 2023. Lecture Notes in Computer Science, vol 14346. Springer, Cham. 2024. URL: https://doi.org/10.1007/978-3-031-51479-1_14

31. Hansen Kim. Practical Oracle SQL: Mastering the Full Power of Oracle Database. 2020. URL: http://dx.doi.org/10.1007/978-1-4842-5617-6

32. Hu N. The Limitations of Traditional PID Controllers and Modern Optimization Methods. Applied and Computational Engineering, 2025. URL: https://doi.org/10.54254/2755-2721/2025.22912

33. Islam Nawroze, Chowdhury Md Rubel, Islam Sefatul. A Comparative Analysis of PHP and Python Programming Languages for Optimal Software Development. figshare. Journal contribution. 2023. URL: https://doi.org/10.6084/m9.figshare.24885846.v1

34. Niarman, A., Iswandi, Candri, A. K. Comparative Analysis of PHP Frameworks for Development of Academic Information System Using Load and Stress Testing. International Journal Software Engineering and Computer Science *(IJSECS)*. 2023. Vol. *3*(3), 424–436. URL: https://doi.org/10.35870/ijsecs.v3i3.1850

35. Turcotte A., Arteca E., Mishra A. *et al.* Stubbifier: debloating dynamic server-side JavaScript applications. Empir Software Eng. 2022. Vol. 27, no. 161. URL: https://doi.org/10.1007/s10664-022-10195-6

36. Ksenia Peguero, Xiuzhen Cheng. CSRF protection in JavaScript frameworks and the security of JavaScript applications. High-Confidence Computing. 2021. Vol. 1, Issue 2. URL: https://doi.org/10.1016/j.hcc.2021.100035

37. M. H. M. Bhuiyan, A. S. Parthasarathy, N. Vasilakis, M. Pradel and C. -A. Staicu, "SecBench.js: An Executable Security Benchmark Suite for Server-Side JavaScript," 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), Melbourne, Australia, 2023, pp. 1059-1070. URL: http://dx.doi.org/10.1109/icse48619.2023.00096

38. Mishra Pawan, Singh ShubhamKumar, Mishra Sonu, Singh Siddharth. User Authentication System Using Python. International Journal of Innovative Research in Advanced Engineering. 2024. Vol. 11. P. 957-963. URL: https://doi.org/10.26562/ijirae.2024.v1112.11

39. Sriramya P., Karthika R.A. Providing password security by salted password hashing using Bcrypt algorithm. ARPN Journal of Engineering and Applied Sciences. 2015. Vol. 10(13). P. 5551–5556

40. Batubara Toras, Efendi Syahril, Nababan Erna. Analysis Performance BCRYPT Algorithm to Improve Password Security from Brute Force. Journal of Physics: Conference Series 1811 012129. 2021. URL: http://dx.doi.org/10.1088/1742-6596/1811/1/012129

ДОДАТКИ

ДОДАТОК А

Порівняння фреймворків ASP.NET Core та Node.js для розробки серверної частини

Таблиця А.1

Критерій	**ASP.NET Core (C#)**	**Node.js (JavaScript)**
Тип мови	Статично типізована, компільована (C#, .NET CLR)	Динамічна, інтерпретована (JavaScript, V8 Engine)
Продуктивність	Висока, багатопотокова; особливо ефективна для CPU-інтенсивних задач	Висока для I/O-інтенсивних задач; асинхронна, однопотокова модель
Масштабованість	Вертикальна масштабованість (збільшення ресурсів сервера, потоків)	Горизонтальна масштабованість (запуск декількох екземплярів сервісу)
Екосистема	Потужна екосистема Microsoft, NuGet-пакети, Entity Framework Core, інтегровані інструменти для розробки та тестування	Велика кількість npm-пакетів, мінімалістичні бібліотеки (Express.js), гнучкість у виборі рішень
Безпека	Вбудовані механізми аутентифікації та авторизації (Identity, JWT), висока захищеність від помилок	Потребує додаткових модулів (Passport.js, JWT), гнучкі рішення з ручним налаштуванням
Підтримка ORM	Entity Framework Core (повноцінний ORM з	Sequelize, TypeORM, Prisma (менш інтегровані з

	високим рівнем абстракції, LINQ-запити)	фреймворком, потребують додаткових налаштувань)
Крива навчання	Помірна: потребує знання ООП, строгих типів і концепцій .NET	Відносно низька: JavaScript широко відомий, швидке освоєння
Документація і підтримка	Високоякісна документація, підтримка Microsoft, велика професійна спільнота	Хороша документація, величезна open-source спільнота, багато прикладів
Інструменти розробки	Visual Studio, Visual Studio Code, Rider з інтеграцією профілювання, тестування, CI/CD	Переважно Visual Studio Code, WebStorm; інструменти розробки легкі та гнучкі
Сфери застосування	Великі корпоративні додатки, CRM, ERP, фінансові та банківські застосунки	Real-time додатки, вебчати, легкі сервіси, стартапи, швидка розробка MVP

ДОДАТОК Б

Фрагмент програмного коду, що генерує JSON Web Token з використанням даних користувача для подальшої авторизації

```
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using System.Text;
using CRMSystem.WebAPI.Models;
using Microsoft.Extensions.Options;
using Microsoft.IdentityModel.Tokens;

namespace CRMSystem.WebAPI.Auth
{
    public class JwtProvider(IOptions<JwtOptions> options)
        : IJwtProvider
    {
        private readonly JwtOptions _options = options.Value;

        public string GenerateToken(User user)
        {
            Claim[] claims =
            [
                new(«userId», user.Id.ToString()),
                new(«username», user.Username),
                new(«role», user.RoleId.ToString())
            ];

            var signingCredentials = new SigningCredentials(
                new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(_options.SecretKey)),
                SecurityAlgorithms.HmacSha256);

            var token = new JwtSecurityToken(
                claims: claims,
                signingCredentials: signingCredentials,
                issuer: _options.Issuer,
                audience: _options.Audience,
                expires:
DateTime.UtcNow.AddHours(_options.ExpireHours));

            var tokenString = new
JwtSecurityTokenHandler().WriteToken(token);

            return tokenString;
        }
    }
}


public static void AddApiAuthentication(this IServiceCollection
services, IConfiguration configuration)
        {
```

```
            var                    jwtOptions                    =
configuration.GetSection(nameof(JwtOptions)).Get<JwtOptions>();


services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)

.AddJwtBearer(JwtBearerDefaults.AuthenticationScheme, options =>
                {
                    options.TokenValidationParameters     =     new
TokenValidationParameters
                    {
                        ValidateIssuer = false,
                        ValidateAudience = false,
                        ValidateLifetime = true,
                        ValidateIssuerSigningKey = true,
                        IssuerSigningKey          =          new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(jwtOptions!.SecretKey)
)
                    };

                    options.Events = new JwtBearerEvents
                    {
                        OnMessageReceived = context =>
                        {
                            context.Token                        =
context.Request.Cookies[«jwt»];
                            return Task.CompletedTask;
                        }
                    };
                });

            services.AddAuthorization();
        }

        public     static     void     AddAuthorizationPolicy(this
IServiceCollection services)
        {
            services.AddAuthorizationBuilder()
                .AddPolicy(AuthorizationPolicies.AdminOnly,  policy
=>

policy.RequireRole(((int)Roles.Admin).ToString()))
                .AddPolicy(AuthorizationPolicies.UserOnly,   policy
=>

policy.RequireRole(((int)Roles.User).ToString()))
                .AddPolicy(AuthorizationPolicies.UserOrAdmin,
policy =>

policy.RequireRole(((int)Roles.User).ToString(),
((int)Roles.Admin).ToString()));
        }
```

```
        public static void AddCookiePolicy(this IApplicationBuilder
app)
        {
            app.UseCookiePolicy(new CookiePolicyOptions
            {
                MinimumSameSitePolicy = SameSiteMode.Strict,
                HttpOnly = HttpOnlyPolicy.Always,
                Secure = CookieSecurePolicy.Always
            });
        }
```

ДОДАТОК В

Основні ендпоінти API серверної частини та розподіл доступу за ролями

Таблиця В.1

HTTP	**URL (ендпоінт)**	**Призначення**	**Доступ (роль)**
POST	/api/auth/sign-up	Реєстрація нового користувача	Публічний (без токена)
POST	/api/auth/sign-in	Вхід користувача, видача JWT-cookie	Публічний
POST	/api/auth/sign-out	Вихід (видалення JWT-cookie)	**User** або **Admin**
POST	/api/student-registration	Створити запис студента (реєстрація студента)	**Admin**
GET	/api/student-registration	Отримати список всіх студентів (з фільтрацією)	**User** або **Admin**
GET	/api/student-registration/{id}	Отримати дані конкретного студента	**User** або **Admin**
PUT	/api/student-registration/{id}	Оновити дані студента	**Admin**
DELETE	/api/student-registration/{id}	Видалити студента	**Admin**
POST	/api/student-registration/import-file	Імпортувати студентів із CSV/Excel	**Admin**
POST	/api/student-registration/export-file	Експортувати всіх студентів у файл	**Admin**
POST	/api/employee-registration	Створити запис працівника (реєстрація персоналу)	**Admin**
GET	/api/employee-registration	Отримати список всіх працівників	**User** або **Admin**
GET	/api/employee-registration/{id}	Отримати дані конкретного працівника	**User** або **Admin**
PUT	/api/employee-registration/{id}	Оновити дані працівника	**Admin**
DELETE	/api/employee-registration/{id}	Видалити працівника	**Admin**

POST	/api/employee-registration/import-file	Імпортувати працівників із CSV/Excel	**Admin**
POST	/api/employee-registration/export-file	Експортувати всіх працівників у файл	**Admin**
GET	/api/finance-report/summary	Отримати фінансовий звіт (зведення)	**Admin**
POST	/api/email/send	Відправити електронне повідомлення (лист)	**User** або **Admin**
GET	/api/email/messages	Переглянути всі надіслані повідомлення	**User** або **Admin**
GET	/api/email/messages/{receiverId}	Переглянути повідомлення, надіслані конкретному отримувачу (за його ID)	**User** або **Admin**
GET	/api/user/me	Отримати профіль поточного користувача	**User** або **Admin**
PATCH	/api/user/me	Оновити дані профілю (ПІБ, email, дату нар.)	**User** або **Admin**
PATCH	/api/user/me/change-password	Змінити пароль користувача	**User** або **Admin**
POST	/api/users/upload-photo	Завантажити фото для профілю користувача	**User** або **Admin**

ДОДАТОК Г

Тестування різних сценаріїв на основі контролера AuthController у класі AuthControllerTests

```
using System.Security.Claims;
using CRMSystem.WebAPI.Controllers;
using CRMSystem.WebAPI.DTOs.Auth;
using CRMSystem.WebAPI.Interfaces;
using CRMSystem.WebAPI.Services;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using Moq;

namespace CRMSystem.Tests.Tests
{
    public class AuthControllerTests
    {
        private readonly Mock<UserService> _userServiceMock;
        private readonly Mock<IValidatorFactory>
_validatorFactoryMock;
        private readonly Mock<ILogger<AuthController>> _loggerMock;
        private readonly AuthController _controller;

        public AuthControllerTests()
        {
            _userServiceMock = new Mock<UserService>();
            _validatorFactoryMock = new Mock<IValidatorFactory>();
            _loggerMock = new Mock<ILogger<AuthController>>();

            _controller = new
AuthController(_userServiceMock.Object,
_validatorFactoryMock.Object, _loggerMock.Object)
            {
                ControllerContext = new ControllerContext
                {
                    HttpContext = new DefaultHttpContext()
                }
            };
        }

        [Fact]
        public async Task SignUp_ReturnsOk_WhenValid()
        {
            // Arrange
            var dto = new SignUpRequestDto(
                «John Doe»,
                DateTime.UtcNow.AddYears(-25),
                «test_user@mail.com»,
                «test_user»,
                «qwerty»,
```

```
                «qwerty»
            );

            _validatorFactoryMock.Setup(v => v.Validate(dto, out
It.Ref<string>.IsAny)).Returns(true);

            // Act
            var result = await _controller.SignUp(dto);

            // Assert
            Assert.IsType<OkResult>(result);
            _userServiceMock.Verify(s => s.SignUp(dto.FullName,
dto.BirthDate, dto.Email, dto.Username, dto.Password), Times.Once);
        }

        [Fact]
        public async Task SignUp_ReturnsBadRequest_WhenInvalid()
        {
            // Arrange
            var dto = new SignUpRequestDto(««, DateTime.MinValue,
««, ««, ««, ««);
            var error = «Invalid input»;
            _validatorFactoryMock.Setup(v => v.Validate(dto, out
error)).Returns(false);

            // Act
            var result = await _controller.SignUp(dto);

            // Assert
            var badRequest =
Assert.IsType<BadRequestObjectResult>(result);
            Assert.Equal(error, badRequest.Value);
        }

        [Fact]
        public async Task
SignUp_ReturnsBadRequest_WhenServiceThrows()
        {
            // Arrange
            var dto = new SignUpRequestDto(
                «Test User»,
                DateTime.UtcNow.AddYears(-25),
                «test_user@mail.com»,
                «test_user»,
                «qwerty»,
                «qwerty»
            );

            _validatorFactoryMock.Setup(v => v.Validate(dto, out
It.Ref<string>.IsAny)).Returns(true);
            _userServiceMock.Setup(s =>
s.SignUp(It.IsAny<string>(), It.IsAny<DateTime>(),
It.IsAny<string>(), It.IsAny<string>(), It.IsAny<string>()))
```

```
                .Throws(new
InvalidOperationException(«User already exists»));

            // Act
            var result = await _controller.SignUp(dto);

            // Assert
            var                    badRequest                    =
Assert.IsType<BadRequestObjectResult>(result);
            Assert.Equal(«User already exists», badRequest.Value);
        }

        [Fact]
        public async Task SignIn_ReturnsOk_AndSetsJwtCookie()
        {
            // Arrange
            var dto = new SignInRequestDto(«test_user», «qwerty»);
            _validatorFactoryMock.Setup(v  =>  v.Validate(dto,  out
It.Ref<string>.IsAny)).Returns(true);
            _userServiceMock.Setup(s    =>    s.SignIn(dto.Username,
dto.Password)).ReturnsAsync(«mocked-jwt»);

            // Act
            var result = await _controller.SignIn(dto);

            // Assert
            Assert.IsType<OkResult>(result);

            var                  cookieHeader                    =
_controller.Response.Headers[«jwt»].ToString();
            Assert.Contains(«jwt=mocked-jwt», cookieHeader);
        }

        [Fact]
        public async Task SignIn_ReturnsBadRequest_WhenInvalid()
        {
            // Arrange
            var dto = new SignInRequestDto(««, ««);
            var error = «Invalid credentials»;
            _validatorFactoryMock.Setup(v  =>  v.Validate(dto,  out
error)).Returns(false);

            // Act
            var result = await _controller.SignIn(dto);

            // Assert
            var                    badRequest                    =
Assert.IsType<BadRequestObjectResult>(result);
            Assert.Equal(error, badRequest.Value);
        }

        [Fact]
        public void SignOut_DeletesJwtCookie_AndReturnsOk()
```

```
        {
            // Arrange
            var claims = new[] { new Claim(«username», «test_user») };
            _controller.ControllerContext.HttpContext.User = new ClaimsPrincipal(new ClaimsIdentity(claims));

            // Act
            var result = _controller.SignOut();

            // Assert
            Assert.IsType<OkResult>(result);
        }
    }
}
```