МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ І ПРИКЛАДНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

МАРУНЯК АНДРІЙ ОЛЕКСАНДРОВИЧ

Допускається до захисту:
в.о. завідувача кафедри
інформаційних технологій,
д-р. техн. наук, професор
_____Наталія ВЕСЕЛОВСЬКА
«____»_____ 2026 р.

РОЗРОБКА СИСТЕМИ КЕРУВАННЯ РОЗКЛАДОМ ЗАНЯТЬ УНІВЕРСИТЕТУ: БАЗА ДАНИХ ТА ШАР ДОСТУПУ ДО ДАНИХ

Спеціальність 122 «Комп’ютерні науки»

Кваліфікаційна (бакалаврська) робота

Керівник:
Юрій АНТОНОВ, доцент кафедри
інформаційних технологій,
к. фіз.-мат. наук, доцент
_____

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

Голова ЕК: _____
(підпис)

Вінниця – 2026

АНОТАЦІЯ

Маруняк А.О. Розробка системи керування розкладом занять університету: база даних та шар доступу до даних. Спеціальність 122 «Комп'ютерні науки», освітня програма «Комп'ютерні науки». Донецький національний університет імені Василя Стуса, Вінниця, 2026.

У кваліфікаційній (бакалаврській) роботі досліджено теоретичні засади проєктування реляційних баз даних та патернів шару доступу до них. Розроблено фізичну схему бази даних та реалізовано шар доступу до даних системи керування розкладом. Програмне рішення побудовано на основі багатошарової архітектури з використанням патернів Repository та Unit of Work, платформи .NET 8, СКБД MySQL та Entity Framework Core.

Ключові слова: розклад занять, реляційна база даних, шар доступу до даних, патерн репозиторій, одиниця роботи, Entity Framework Core.

75 ст., 5 рис., 21 лістинг, 3 табл., 50 джерел.

ABSTRACT

Maruniak A.O. Development of a university schedule management system: database and data access layer. Specialty 122 «Computer Science», educational program «Computer Science». Vasyl' Stus Donetsk National University, Vinnytsia, 2026.

The qualification (bachelor's) work investigates the theoretical foundations of relational database design and data access patterns. A physical database schema was developed, and a data access layer for the schedule management system was implemented. The software solution is based on an N-Layer architecture using Repository and Unit of Work patterns, the .NET 8 platform, MySQL DBMS, and Entity Framework Core.

Keywords: schedule management, relational database, data access layer, repository pattern, unit of work, Entity Framework Core.

75 p., 5 fig., 21 listings, 3 tables, 50 references

ЗМІСТ

ВСТУП

Актуальність теми дослідження: Сучасний етап розвитку інформаційних технологій глибоко трансформує підходи до управління освітніми процесами. Для закладів вищої освіти однією з найважливіших і найскладніших організаційних задач є формування та адміністрування розкладу занять. Цей процес вимагає врахування безлічі факторів: місткості аудиторного фонду, робочого навантаження викладачів, специфіки студентських потоків та уникнення будь-яких часових або просторових колізій.

Традиційні підходи до складання розкладу є надзвичайно трудомісткими та схильними до людських помилок, що зумовлює гостру потребу в автоматизації шляхом розробки сучасних програмних рішень. Проте ефективність, швидкодія та стійкість корпоративних вебзастосунків критично залежать від їхнього внутрішнього архітектурного фундаменту – надійної бази даних та гнучкого шару доступу до даних. Саме архітектура збереження та обробки інформації гарантує цілісність академічних даних і можливість легко розширювати функціонал системи у майбутньому, що робить розробку даної підсистеми вкрай актуальним завданням.

Мета дослідження: розробка реляційної бази даних та створення ефективного шару доступу до даних у структурі вебзастосунку, призначеного для перегляду та формування розкладу занять у вищих навчальних закладах.

Завдання дослідження:

- Дослідити специфіку предметної області;
- Проаналізувати існуючі альтернативи;
- Визначити необхідні дані для збереження;
- Спроєктувати структуру бази даних для обраних даних;
- Розробити архітектуру шару доступу до даних для веб-додатку з підключенням до спроєктованої бази даних.

Об’єкт дослідження: процес автоматизації управління навчальним процесом, зокрема процедура складання та адміністрування розкладу занять у закладах вищої освіти.

Предмет дослідження: методи, моделі та сучасні архітектурні підходи, що застосовуються для проєктування баз даних і побудови шару доступу до даних.

Практичне значення одержаних результатів: створення готової до інтеграції, стійкої до навантажень підсистеми управління даними, яка утворює надійний фундамент для системи управління розкладом.

Зв’язок роботи з науковими програмами, планами, темами:

Наведені у бакалаврські роботі дослідження пов’язані з прикладною науково-дослідною роботою «Розвиток методів комп’ютерно-математичного моделювання складних систем та процесів у сфері освіти та діяльності IT-підприємств» (№ держреєстрації 0126U003221, 2026-2031 рр.)

Апробація результатів дослідження: публікація тез на тему «Development of a generalized data model and data access architecture for academic scheduling systems using .net and blazor» у IX міжнародній науково-практичній конференції, що проводилась «Європейською науковою платформою» у співорганізаторстві з американською компанією LLC «Boston Data Science Group» у м. Бостон, США, 10 квітня 2026 року.

Структура кваліфікаційної роботи: Кваліфікаційна робота складається із вступу, трьох розділів та висновків до них, списку використаних джерел.

У першому розділі проведено огляд предметної області, існуючих альтернатив, технологічного стеку та патернів розробки.

У другому розділі сформовано постановку задачі, схематично спроєктовано базу даних та шар доступу до даних.

У третьому розділі обрано і аргументовано набір інструментів для реалізації проєкту, висвітлено реалізацію повноцінної бази даних та шару доступу до даних з втіленням обраних патернів.

Кваліфікаційна робота включає в себе 75 сторінок, 5 рисунків і список літератури із 50-ти джерел.

РОЗДІЛ 1
ТЕОРЕТИЧНІ ЗАСАДИ ПОБУДОВИ БАЗ ДАНИХ ТА АРХІТЕКТУРИ ДОСТУПУ ДО ІНФОРМАЦІЇ В СИСТЕМАХ УПРАВЛІННЯ НАВЧАЛЬНИМ ПРОЦЕСОМ

1.1 Специфіка предметної області та концептуальні абстракції системи формування розкладу

Процес автоматизації розкладу занять у закладах вищої освіти (ЗВО) належить до класу задач з високим рівнем описової складності та жорсткими вимогами до логічної узгодженості інформації [1]. Предметна область характеризується великою кількістю взаємопов'язаних концептів, станом яких необхідно керувати в реальному часі. Будь-яка сутність у цій системі не є ізольованою, а виступає компонентом спільного інформаційного простору університету [2, 3].

Для побудови концептуальної моделі системи формування розкладу необхідно виділити та формалізувати базові абстракції предметної області:

Фізична особа – узагальнена сутність, що описує людину в межах освітнього процесу. Особливістю цього концепту є динамічна диференціація ролей – одна й та сама особа може виступати у ролі викладача, студента або адміністратора системи із відповідним розмежуванням атрибутивного складу.

Кафедра – організаційно-структурна одиниця ЗВО, яка є базовим класифікатором розподілу академічних ресурсів, кадрового потенціалу та навчального навантаження.

Аудиторія – матеріальний фонд закладу освіти. Кожен екземпляр цієї сутності визначається фізичною місткістю та функціональним типом, що накладає жорсткі обмеження на можливість проведення певних видів занять.

Академічна група – стабільне об'єднання здобувачів вищої освіти. Цей концепт має складну внутрішню архітектуру, оскільки передбачає підтримку ієрархічних зв'язків та горизонтального поділу. Наприклад, поділ може відбуватися на рівні лекційних потоків, що об'єднують декілька груп, або

навпаки – розбиття базових груп на менші підгрупи для проведення практичних і лабораторних занять.

Навчальне навантаження – ключова сполучна сутність, що агрегує вимоги навчального плану дисципліни, закріплює їх за конкретним викладачем (або кафедрою) та визначає обсяг годин і видів занять для певної академічної групи.

Запис розкладу – підсумковий запис, що фіксує конкретне заняття: де, коли, хто викладає і яка група присутня [4].

Взаємозв'язки між зазначеними концептами мають виражену специфіку. Організаційна структура ЗВО має ієрархічний характер. Тому модель даних повинна підтримувати self-referencing (з посиланням самих на себе) зв’язки для відображення підпорядкованості підрозділів та вкладеності академічних груп [5, 6]. Зв'язки між заняттями та суб'єктами процесу часто реалізуються за принципом множинних асоціацій (багато-до-багатьох), коли одне лекційне заняття може проводитися для кількох груп одночасно, або одну дисципліну синхронно викладають декілька фахівців. Окрему теоретичну складність становить часова дискретизація – необхідність моделювання циклічності навчального процесу (поділ на чисельник/знаменник, парні/непарні тижні, фіксовані часові слоти).

Щільність взаємозв'язків у предметній області породжує характерні структурні ризики. Спільне використання фізичних ресурсів – аудиторій і часових слотів – кількома незалежними суб'єктами розкладу створює потенціал для просторово-часових колізій: одна аудиторія або викладач можуть опинитися одночасно у двох точках розкладу. Крім того, архівні записи занять за минулі семестри мають самостійну документальну цінність і існують незалежно від поточного стану кадрового або організаційного реєстру – видалення довідникового запису не знищує породжену ним транзакційну історію.

Аналіз структурних особливостей предметної області дозволяє ідентифікувати три принципові інженерні виклики для архітектури сховища даних. Перший – структурна гнучкість: освітні стандарти динамічно змінюються, тому модель має витримувати еволюцію схеми без руйнування

наявних даних. Другий – масштабованість: розклад університету є інтенсивно читаним ресурсом, і швидкість вибірки повинна залишатися стабільною при зростанні кількості записів і користувачів. Третій – транзакційна цілісність: резервування аудиторії, часового слоту та навчального навантаження є нерозривним актом – часткове збереження означає суперечливий стан даних [4].

1.2 Огляд та порівняльний аналіз існуючих підходів до організації даних у системах керування розкладом

Визначення оптимальної архітектурної стратегії проєктування шару даних потребує критичного аналізу існуючого світового досвіду побудови інформаційних систем керування освітнім процесом. Як репрезентативну вибірку обрано три відомі системи, що демонструють принципово різні підходи до побудови абстракції над реляційними базами даних: UniTime, RosarioSIS та Moodle.

Першим розглянутим підходом є відкрита система академічного планування UniTime [7]. На рівні збереження даних ця система спирається на жорстку реляційну модель, переважно з використанням системи керування базами даних PostgreSQL, взаємодія з якою реалізована за допомогою повноцінного об'єктно-реляційного мапінгу. Зазначена архітектура реалізує патерн Data Mapper, що забезпечує повну абстракцію шару даних від бізнес-логіки. Організація даних у UniTime орієнтована на constraint-based модель, де велика кількість обмежень цілісності та зв'язків керується транзакційно через декларативні механізми, забезпечуючи високий рівень контролю цілісності даних.

Альтернативний підхід демонструє веб-орієнтована інформаційна система RosarioSIS, що містить вбудований модуль управління розкладом [8]. Система підтримує роботу з СКБД PostgreSQL та MySQL. Її архітектура доступу до даних базується на використанні прямих SQL-запитів через інтерфейс PDO, що свідчить про фактичну відсутність абстрактного DAL.

Натомість система реалізує патерн Transaction Script, де SQL-інструкції вбудовані безпосередньо в логіку обробки PHP-скриптів. Таке рішення дозволяє виконувати нативні вибірки з мінімальним накладним часом, проте водночас створює міцну зв'язаність між базою даних та інтерфейсом.

Проміжне архітектурне рішення втілено в Moodle – найпоширенішій у закладах вищої освіти відкритій платформі управління навчанням [9]. На відміну від двох попередніх систем, Moodle використовує власний інфраструктурний шар Moodle Data Manipulation Layer. Ця бібліотека виступає обгорткою над рівнем підключення і надає уніфікований API для взаємодії з різними СКБД (MySQL, PostgreSQL, Oracle, MSSQL) без необхідності зміни логіки додатку. Структура бази даних у цій системі версіонується через механізм XMLDB, що дозволяє автоматично генерувати DDL-скрипти. Важливою особливістю є часткова відмова від зовнішніх ключів на фізичному рівні СКБД як історична спадщина, внаслідок чого відповідальність за посилкову цілісність перенесено на програмний рівень із використанням делегованих транзакцій.

Для систематизації результатів аналізу було проведено порівняння розглянутих систем за критеріями, які характеризують архітектуру доступу до даних, рівень ізоляції бізнес-логіки, підтримку цілісності та гнучкість супроводу системи [10]. Оцінювання виконано на основі аналізу офіційної документації, структури DAL та особливостей реалізації механізмів роботи з базою даних.

Основними критеріями порівняння є:

- Рівень абстракції доступу до даних;
- Підтримка різних СКБД;
- Механізми забезпечення цілісності;
- Підтримка модифікації схеми даних;
- Продуктивність аналітичних запитів;
- Рівень ізоляції DAL та тестованості.

Результати порівняльного аналізу наведено в таблиці 1.1.

Деталізація результатів дозволяє чітко побачити специфіку кожного підходу. У системі RosarioSIS інфраструктурна абстракція відсутня: SQL-запити

вбудовані безпосередньо у скрипти обробки. Це призводить до змішування логіки доступу до даних із шаром представлення та ускладнює тестування. У Moodle застосовано часткову абстракцію через універсальний API, проте використання глобального об'єкта підключення обмежує ізоляцію. Максимальну незалежність шару даних демонструє UniTime. Завдяки об'єктно-реляційному відображенню та абстрактним репозиторіям досягається чітке розділення доменної та інфраструктурної логіки. Щодо підтримки систем керування баз даних, Moodle вирізняється кросплатформеністю, тоді як RosarioSIS та UniTime орієнтовані переважно на PostgreSQL.

Таблиця 1.1 Порівняльний аналіз архітектурних рішень організації даних у системах розкладу

Критерій порівняння	RosarioSIS	Moodle	UniTime
Рівень абстракції DAL	Відсутній	Частковий	Високий
Підтримка різних СКБД	Обмежена	Широка	Обмежена
Механізми забезпечення цілісності	Часткові	Часткові	Розвинені
Підтримка еволюції схеми БД	Обмежена	Висока	Висока
Продуктивність аналітичних запитів	Висока	Середня	Середня
Ізоляція бізнес-логіки від DAL	Низька	Середня	Висока
Тестованість компонентів доступу до даних	Низька	Середня	Висока

Розбіжності спостерігаються і в питаннях забезпечення цілісності та еволюції схем бази даних.

Найвищий рівень контролю цілісності гарантує UniTime за рахунок композитних зовнішніх ключів у БД та перевірки доменних правил до фіксації транзакцій. У Moodle цей показник є середнім через історичну відсутність строгих обмежень зовнішніх ключів на рівні СКБД, що перекладає валідацію на програмний код. Аналогічно, у RosarioSIS цілісність критично залежить від ручних перевірок. Водночас Moodle та UniTime забезпечують швидку та безпечну модифікацію схем завдяки декларативному опису та автоматизованим міграціям. Натомість у RosarioSIS будь-яка зміна структури вимагає трудомісткого ручного оновлення SQL-коду.

Баланс між продуктивністю та архітектурною чистотою також є визначальним. Найвищу швидкість читання даних демонструє RosarioSIS завдяки нативним SQL-вибіркам і відсутності додаткових рівнів абстракції. У Moodle та UniTime продуктивність аналітичних запитів є середньою через накладні витрати на динамічну генерацію об'єктів та трансляцію запитів. Тому вибір архітектурного патерну завжди залишається компромісом: платформи з розвиненою абстракцією вимагають додаткового кешування для компенсації втрат швидкості читання.

Окрему увагу заслуговують вітчизняні розробки.

Документоорієнтований підхід до автоматизації складання розкладу на платформі 1С:Підприємство описано у роботах [11, 12]. Система оперує стандартними об'єктами конфігурації – довідниками (дисципліни, викладачі, аудиторії) та документами (розклад, відомості до розкладу, навчальний план). Перевагою цього рішення є можливість автоматичного заповнення нових документів на основі попередніх, що ефективно скорочує час формування звітності та зменшує кількість ручних помилок керівника. Однак недоліком даної роботи є архітектура платформи 1С:Підприємство котра орієнтована на закрите корпоративне середовище і не передбачає повноцінного веб-доступу, що обмежує інтеграцію з браузерними клієнтами.

Суміжний підхід до автоматизації навчального процесу реалізовано у роботі [13], де розроблено інформаційну систему підтримки діяльності

структурного підрозділу ЗВО на базі мови Python та СКБД MySQL. Система включає модуль автоматичного формування розкладу з підтримкою жорстких та м'яких обмежень планування. Водночас архітектурним недоліком є монолітна організація застосунку: локальний десктопний клієнт на базі бібліотеки Tkinter взаємодіє зі сховищем безпосередньо, без виділеного шару доступу до даних, що ускладнює забезпечення транзакційної цілісності та унеможливлює надійний багатокористувацький доступ через мережу.

Безпосереднім попередником розроблюваної підсистеми є програмний модуль A.S.T.S. Schedule Module [14, 15]. Модуль забезпечує базовий функціонал перегляду та формування розкладу як модуль системи тестування A.S.T.S. [2]. Разом із тим клієнтська частина системи взаємодіє зі сховищем без абстрактного шару доступу до даних, що обмежує масштабованість та ускладнює інтеграцію в сучасне веб середовище. Саме ці архітектурні обмеження визначають ключовий напрям розробки, що реалізується у даній кваліфікаційній роботі.

1.3 Теоретичні підходи до проєктування реляційних баз даних та абстракції DAL

Вибір моделі збереження даних – один із найважливіших кроків при проєктуванні системи розкладу. Тому на етапі системного аналізу ключовим завданням є вибір такої концепції сховища, яка гарантуватиме транзакційну безпеку, забезпечуватиме швидку обробку складних запитів та дозволятиме гнучко розширювати структуру бази в майбутньому.

Щоб прийняти обґрунтоване рішення, необхідно провести порівняльний аналіз сучасних реляційних та документо-орієнтованих парадигм, оцінити експлуатаційні характеристики популярних СКБД, а також розглянути базові принципи нормалізації відношень. Правильно обрана архітектура сховища дозволить ефективно вирішувати специфічні проблеми просторово-часових

колізій, гарантуючи миттєву валідацію доступності аудиторій та викладачів без ризику подвійного бронювання.

1.3.1 Порівняльний аналіз реляційної та документо-орієнтованої парадигм організації сховищ

Сучасна теорія баз даних виділяє дві фундаментальні архітектурні парадигми організації сховищ: реляційну (SQL) та нереляційну, одним із найпопулярніших представників якої є документо-орієнтована (NoSQL) модель. Для того щоб обрати оптимальне рішення, необхідно розуміти принципи роботи, переваги та обмеження кожного з підходів.

Реляційна парадигма базується на математичному апараті теорії множин та реляційній алгебрі [10]. У цій моделі дані структуруються у вигляді двовимірних таблиць, де кожен рядок представляє окремий запис, а колонка – його властивість. Фундаментальною характеристикою реляційного підходу є наявність жорстко заданої схеми даних та забезпечення зв'язності між таблицями за допомогою механізмів первинних і зовнішніх ключів.

До ключових переваг реляційної моделі належать:

- Гарантія узгодженості даних: Використання нормалізації дозволяє мінімізувати дублювання інформації. Зміна певного атрибута (наприклад, назви посади) відбувається лише в одній довідковій таблиці, що виключає ризик виникнення аномалій оновлення.
- Транзакційна надійність (ACID): Реляційні системи забезпечують атомарність, узгодженість, ізольованість та довговічність транзакцій, що є критично важливим для операцій, які вимагають абсолютної точності (наприклад, фінансові перекази або строге бронювання ресурсів).
- Потужний аналітичний апарат: Наявність декларативної мови SQL та нативних операцій об'єднання дозволяє ефективно агрегувати дані з багатьох таблиць на рівні самого сервера баз даних.

Головними недоліками реляційного підходу є структурна негнучкість та складність горизонтального масштабування.

Для підвищення продуктивності реляційних систем зазвичай застосовують вертикальне масштабування тобто нарощування потужностей одного сервера, що має свої апаратні та фінансові межі.

Документо-орієнтована парадигма (NoSQL), зі свого боку, відмовляється від табличної структури. Вона оперує концепцією безсхемного збереження інформації у вигляді ієрархічних структурованих документів, найчастіше у форматах JSON або BSON. Замість розбиття об'єкта на нормалізовані таблиці, вся пов'язана інформація зберігається як єдиний самодостатній агрегат.

До переваг документо-орієнтованого підходу належать:

- Гнучкість структури: Відсутність жорсткої схеми дозволяє зберігати документи з різним набором полів в одній колекції. Це дає змогу швидко адаптувати додаток до нових вимог без складних міграцій бази даних.
- Висока продуктивність читання/запису: Оскільки документ містить усю необхідну інформацію вкладено (денормалізовано), системі не потрібно виконувати ресурсомісткі операції з'єднання таблиць. Читання об'єкта відбувається за одне звернення до диска.
- Природне горизонтальне масштабування: Незалежність документів один від одного дозволяє легко розподіляти колекції між багатьма серверами в кластері, забезпечуючи високу доступність системи під екстремальними навантаженнями.

Однак ця парадигма має суттєві обмеження. Відсутність нативних операцій об’єднання перекладає логіку об'єднання даних на код додатку. Крім того, денормалізація призводить до значного дублювання інформації. Якщо статус або назва певної сутності зміниться, системі доведеться знайти та оновити тисячі документів, що підвищує ризик часткової втрати цілісності.

Особливо помітною різниця між SQL та NoSQL системами стає в розподілених середовищах. У цьому контексті часто розглядають теорему CAP, що розшифровується як Consistency, Availability, Partition tolerance, котра

стверджує, що для будь-якої розподіленої комп'ютерної системи неможливо одночасно забезпечити виконання більше двох із перелічених трьох властивостей [16, 17]:

- узгодженість даних (усі вузли бачать однакові дані у будь-який момент часу);
- доступність (гарантія того, що кожен запит отримає коректну відповідь);
- стійкість до розділення (попри розділення на ізольовані секції або втрати зв'язку з частиною вузлів, система не втрачає стабільність і здатність коректно відповідати на запити).

Реляційні системи надають перевагу узгодженості, гарантуючи, що всі користувачі бачать однакові дані одночасно. Натомість документо-орієнтовані рішення найчастіше функціонують у межах моделі BASE, допускаючи стан відкладеної або кінцевої узгодженості заради забезпечення максимальної доступності та стійкості до розподілу мережі [18].

Отже, різниця застосування між цими парадигмами залежить саме від топології даних та предметної області де реляційна модель є оптимальною для систем із високою щільністю перехресних посилань і суворими вимогами до цілісності, тоді як документо-орієнтована ідеально підходить для високонавантажених систем із неструктурованими даними або ізольованими агрегатами.

1.3.2 Огляд і порівняння реляційних систем управління базами даних та транзакційних моделей

Вибір системи керування базами даних є одним із ключових етапів проєктування веб-додатку, оскільки саме СКБД визначає механізми зберігання інформації, підтримку транзакцій, продуктивність запитів та можливості масштабування системи. Для обґрунтованого вибору платформи доцільно провести порівняльний аналіз найбільш поширених реляційних СКБД, які

використовуються у сучасних інформаційних системах. У межах дослідження розглянуто IBM Informix, MySQL, PostgreSQL та Microsoft SQL Server.

IBM Informix є корпоративною серверною СКБД від компанії IBM, що позиціонується як рішення для критично важливих OLTP-навантажень із надвисокими вимогами до надійності [19]. Система забезпечує повну підтримку моделі ACID, блокування на рівні рядків, збережені процедури та тригери, а серед нетипових можливостей – вбудовану підтримку часових рядів і просторових даних. Водночас адміністрування вимагає глибокої IBM-специфічної експертизи: власна термінологія та окремий набір діагностичних утиліт суттєво підвищують поріг входу. Комерційна модель ліцензування та значно менша порівняно з відкритими рішеннями спільнота додатково обмежують застосування системи у типових веб-проєктах.

Microsoft SQL Server належить до корпоративних серверних СКБД та забезпечує високий рівень продуктивності, безпеки й підтримки транзакцій [20]. Система добре інтегрується з платформою .NET і містить розвинені засоби аналітики та адміністрування. Недоліком є висока вартість комерційного ліцензування та значні витрати на підтримку інфраструктури, особливо у хмарних середовищах.

PostgreSQL є однією з найпотужніших відкритих реляційних СКБД [21]. Вона підтримує стандарти SQL, складні аналітичні запити, користувацькі типи даних, тригери та механізми розширення функціональності. PostgreSQL забезпечує високий рівень контролю цілісності даних та ефективно працює зі складними JOIN-операціями. Разом із цим система потребує більш ретельного адміністрування та налаштування для досягнення максимальної продуктивності.

MySQL є однією з найпоширеніших реляційних СКБД з відкритим вихідним кодом [22]. Використання рушія InnoDB забезпечує підтримку транзакційної моделі ACID, зовнішніх ключів та механізмів забезпечення посилальної цілісності [23]. Система відзначається високою швидкістю читання даних, простотою розгортання та широкою підтримкою у веб-розробці.

Для систематизації результатів аналізу використано такі критерії порівняння:

- архітектурний тип системи;
- підтримка транзакцій;
- механізми забезпечення цілісності даних;
- продуктивність читання;
- ефективність виконання JOIN-операцій;
- умови ліцензування.

Результати порівняння наведено в таблиці 1.2.

Таблиця 1.2 Порівняльний аналіз характеристик систем управління базами даних

Критерій порівняння	IBM Informix	MySQL	PostgreSQL	MS SQL Server
Архітектурний тип	Серверна	Серверна	Серверна	Серверна
Підтримка транзакцій ACID	Повна	Повна	Повна	Повна
Контроль цілісності даних	Високий	Високий	Високий	Високий
Продуктивність читання	Висока	Висока	Висока	Висока
Ефективність JOIN-операцій	Висока	Висока	Висока	Висока
Масштабованість	Висока	Середня	Висока	Висока
Складність адміністрування	Висока	Низька	Середня	Середня
Умови ліцензування	Комерційна	Open Source	Open Source	Комерційна

Окрім вибору конкретної СКБД, важливим аспектом проєктування є підтримка транзакційної моделі ACID, яка забезпечує надійність роботи системи [23, 24]. Основними властивостями цієї моделі є:

- Атомарність (Atomicity) – транзакція виконується повністю або не виконується взагалі.

- Узгодженість (Consistency) – після завершення транзакції база даних повинна залишатися у коректному стані.
- Ізольованість (Isolation) – паралельні транзакції не повинні впливати одна на одну.
- Довговічність (Durability) – підтверджені зміни не втрачаються навіть у разі збою системи.

Для систем формування розкладу підтримка ACID є особливо важливою, оскільки дозволяє уникати конфліктів при одночасному редагуванні розкладу кількома користувачами, наприклад під час бронювання аудиторій або призначення викладачів на часові слоти. Зокрема, сувора ізоляція гарантує, що два керівника не зможуть параллельно закріпити одне приміщення за різними групами. У разі винекнення колізії механізм повернення у початковий стан автоматично скасує операцію і збереже цілісність довідників.

1.3.3 Математичний апарат нормалізації відношень та усунення інформаційних аномалій

Логічне проєктування схеми реляційної бази даних базується на математичному апараті нормалізації, який є послідовним процесом декомпозиції одного відношення на декілька взаємопов'язаних таблиць з метою усунення надлишковості даних та ліквідації інформаційних аномалій [23]. Математичним підґрунтям нормалізації є теорія функціональних залежностей атрибутів відношення. Послідовне приведення схеми через систему нормальних форм виступає інструментом інженерного захисту цілісності сховища [10].

Перша нормальна форма (1НФ) встановлює базову вимогу реляційної моделі: відношення повинно містити лише атомарні значення атрибутів. Наявність списків, масивів або повторюваних груп у межах однієї комірки є неприпустимою. У контексті розкладу ЗВО приведення до 1НФ вимагає, наприклад, розділення композитного текстового поля ПІБ викладача на окремі

атомарні атрибути (прізвище, ім'я, по батькові) для забезпечення можливості подальшої індексації та точного пошуку збігів.

Друга нормальна форма (2НФ) вимагає виконання умов 1НФ та повної функціональної залежності кожного неключового атрибута від усього складеного первинного ключа, що усуває часткові залежності. Якщо первинний ключ таблиці є композитним тобто складається з декількох полів, то не повинно існувати полів, які залежать лише від однієї частини цього ключа.

Порушення цієї вимоги призводить до аномалій: наприклад, якщо зберігати місткість аудиторії в одній таблиці із записами занять, де ключем є комбінація часового слота та кабінету, дані про місткість будуть багаторазово дублюватися, що спричинить надлишковість.

Третя нормальна форма (3НФ) вимагає відповідності вимогам 2НФ та додатково встановлює умову повної відсутності транзитивних функціональних залежностей неключових атрибутів від первинного ключа.

Це означає, що жодне неключове поле не повинно залежати від іншого неключового поля. На прикладі довідника співробітників, якщо атрибут посади визначає розмір ставки навантаження, то ставка залежить від ключа транзитивно, в цьому випадку – через посаду. Винесення таких залежностей в окрему довідкову таблицю усуває інформаційну надлишковість сховища.

Нормальна форма Бойса-Кодда (НФБК) є більш суворим теоретичним розширенням 3НФ [25]. Вона застосовується до відношень, які мають кілька складених потенційних ключів, що перетинаються між собою. Правило НФБК стверджує, що кожна функціональна залежність у таблиці повинна мати своїм детермінантом потенційний первинний ключ відношення. Вищі рівні декомпозиції – четверта (4НФ, усунення багатозначних залежностей) та п'ята (5НФ, залежності проекції-з'єднання) форми – використовуються для специфічних комбінаторних задач і в практичній інженерії веб-додатків застосовуються вкрай рідко, оскільки надмірна декомпозиція призводить до деградації продуктивності СКБД через потребу виконання надлишкової кількості важких операцій об’єднання.

1.3.4 Методології синхронізації Code-First та DB-First в контексті еволюції схем баз даних

Важливим кроком проєктування інфраструктури є визначення методологічного вектору взаємодії та синхронізації між об'єктно-орієнтованим кодом додатку та реляційною схемою СКБД.

У сучасній інженерії програмного забезпечення цю задачу вирішують за допомогою двох альтернативних підходів: DB-First та Code-First [26, 27].

Підхід DB-First передбачає, що первинним етапом розробки є створення фізичної структури таблиць, зв'язків та обмежень безпосередньо на рівні сервера бази даних. Після цього інструментарій доступу до даних виконує процес зворотного проєктування і автоматично генерує відповідні класи моделей у коді програми.

Переваги: тотальний низькорівневий контроль розробника над типами даних СКБД, конфігурацією індексів, планами виконання інструкцій та збереженими процедурами.

Недоліки: значне ускладнення процесу синхронізації змін структури сховища в розподілених командах розробки, а також високий ризик випадкового затирання специфікованих методів та властивостей об'єктних моделей при повторній автоматичній генерації коду після кожної модифікації таблиць СКБД.

Підхід Code-First базується на протилежній концепції, де первинною точкою проєктування є декларативний опис доменних сутностей у вигляді чистих класів мови програмування високого рівня. На основі аналізу структури цих класів та їхніх конфігурацій інфраструктурний шар додатку автоматично будує фізичну схему бази даних та підтримує її актуальність за допомогою конвеєра міграцій.

Переваги: висока швидкість початкового прототипування системи, можливість декларативного опису обмежень цілісності безпосередньо в коді проєкту, а також просте й надійне версіонування еволюції структури сховища в системах контролю версій разом із кодом додатку.

Недоліки: ризик генерації інструментарієм неоптимального SQL-коду або невідповідних фізичних типів даних за умовчанням, що вимагає від архітектора обов'язкового явного перевизначення правил мапінгу для складних зв'язків.

Еволюційний розвиток схеми бази даних у системах формування розкладу є неминучим процесом.

Навчальний процес у закладах вищої освіти динамічно адаптується під нові державні стандарти, що виражається у появі нових форм контролю, зміні структур підрозділів, або інтеграції інноваційних видів навантаження. Вибір між методологіями Code-First та DB-First визначає інженерний життєвий цикл управління змінами: або розвиток системи керується через фізичний рівень СКБД, або еволюція сховища підпорядкована трансформаціям чистої об'єктної доменної моделі програми.

1.4 Архітектурні патерни організації шару доступу до даних

Організація шару доступу до даних є важливою складовою архітектури сучасних інформаційних систем, оскільки саме від неї залежать рівень зв’язності компонентів, зручність тестування та подальший супровід програмного забезпечення. Під час розробки DAL виникає проблема узгодження об’єктної моделі програми з реляційною структурою бази даних.

Для вирішення цієї проблеми використовують архітектурні патерни доступу до даних. Вони дозволяють відокремити бізнес-логіку від механізмів роботи з базою даних, централізувати формування запитів та забезпечити контроль транзакцій. Крім того такий підхід спрощує модифікацію системи та зменшує залежність прикладного коду від конкретної СКБД. Для систем управління навчальним процесом це особливо важливо, оскільки вони працюють з великою кількістю взаємопов’язаних даних і потребують підтримки їхньої цілісності при одночасній роботі багатьох користувачів.

1.4.1 Object-Relational Impedance Mismatch та стратегії мапінгу: Active Record проти Data Mapper

Головним технологічним бар'єром при розробці шару доступу до даних є Object-Relational Impedance Mismatch тобто об'єктно-реляційна розбіжність яка зумовлена різницею між концепціями об'єктної та реляційної парадигм [28]. Об'єктно-орієнтовані моделі оперують поняттями інкапсуляції, успадкування, поліморфізму та ідентичності об'єктів через посилання в оперативній пам'яті, тоді як реляційні системи базуються на плоских таблицях, зовнішніх ключах та теоретико-множинній логіці кортежів. Для подолання цієї розбіжності в архітектурі корпоративного програмного забезпечення застосовують дві базові стратегії відображення стану об'єктів у табличні структури, класифіковані М. Фаулером, а саме Active Record та Data Mapper.

Патерн Active Record передбачає, що об'єкт класу поєднує в собі як безпосередні доменні дані (атрибути рядка таблиці), так і логіку прямої взаємодії з базою даних, включаючи вбудовані методи збереження, оновлення та видалення. Такий підхід є ефективним для систем із нескладною бізнес-логікою, проте він жорстко порушує принцип єдиної відповідальності (Single Responsibility Principle).

Натомість патерн Data Mapper реалізує повну ізоляцію доменної моделі від шару збереження. Об'єкти бізнес-логіки залишаються чистими і не мають жодних відомостей про існування конкретної СКБД чи структуру її таблиць, а за трансляцію стану об'єктного графа у реляційні відношення відповідає незалежний інфраструктурний шар мапера. Такий архітектурний підхід є еталонним для складних систем.

Завдяки строгому розділенню відповідальності, розробники можуть незалежно еволюціонувати реляційну схему та доменну модель. Крім того, абстрагування від фізичного сховища критично спрощує модульне тестування алгоритмів генерації розкладу.

1.4.2 Патерни Repository та Unit of Work

Для структурування інфраструктурного коду шару DAL та забезпечення слабкої зв'язаності між бізнес-компонентами системи та механізмами фізичного збереження даних застосовується інтеграція патернів Repository (Репозиторій) та Unit of Work (Одиниця роботи) [29, 3]. Зазначені патерни виступають логічною надбудовою над об'єктно-реляційним мапером типу Data Mapper, забезпечуючи високий рівень абстракції та спрощуючи архітектуру додатку [29].

Патерн Repository інкапсулює логіку створення низькорівневих інфраструктурних запитів і надає вищим архітектурним шарам інтерфейс, що імітує звичайну колекцію об'єктів у пам'яті.

Централізація вибірок в інтерфейсах репозиторіїв захищає бізнес-сервіси від протікання інфраструктурних деталей. Окрім захисту від дублювання коду, це дозволяє модульне тестування підміняючи реальні сховища їхніми тестовими заглушками без потреби розгортання реального екземпляра сервера баз даних.

Патерн Unit of Work відповідає за вирішення проблеми транзакційної атомарності при роботі з декількома репозиторіями в межах однієї бізнес-сесії. Він підтримує єдиний контекст, відстежує всі модифікації в об'єктах протягом виконання операції та гарантує, що фіксація змін у базі даних відбудеться як одна неподільна транзакція СКБД. У системах управління розкладом це є критично важливим, оскільки процес створення одного заняття вимагає одночасного оновлення статусу аудиторії, доступності викладача та списання відповідних годин із сутності навчального навантаження. Якщо хоча б одна з цих пов'язаних операцій завершується інфраструктурною помилкою, Unit of Work здійснює повний відкат транзакції, запобігаючи дестабілізації сховища та появі неузгоджених даних. Окрім гарантії цілісності даних при каскадних оновленнях, даний патерн оптимізує швидкодію системи. Замість десятків окремих звернень до сховища, усі зміни накопичуються в пам'яті та фіксуються єдиним транзакційним пакетом.

1.4.3 Об’єкт передачі даних та контрактний підхід

При проектуванні багаторівневих веб-додатків постає завдання розмежування внутрішньої структури бази даних та зовнішнього представлення інформації. Для вирішення цієї задачі застосовується контрактний підхід на основі патерну Data Transfer Object. DTO являє собою просту структуру даних, позбавлену будь-якої бізнес-логіки, призначену виключно для транспортування інформації між шарами системи [28].

Використання DTO вирішує три ключові проблеми. По-перше, запобігає надмірній вибірці – замість повного графа сутності передається лише необхідний набір атрибутів. По-друге, усуває проблему циклічних посилань, що виникають в ORM при моделюванні двосторонніх зв'язків. По-третє, гарантує стабільність зовнішніх інтерфейсів – внутрішні зміни схеми бази даних локалізуються всередині DAL і не руйнують UI-компоненти.

Реалізація DTO та логіки маппінгу між доменними моделями і об'єктами передачі даних належить до шару бізнес-логіки і виходить за межі даної роботи. Шар доступу до даних забезпечує передумову для застосування цього патерну, надаючи чисті доменні класи без інфраструктурних залежностей, що робить їх придатними для подальшої проекції у DTO на вищих рівнях системи.

Висновок до першого розділу

У першому розділі виконано системний аналіз предметної області та формалізовано теоретичні засади організації сховища даних для інформаційних систем управління університетським розкладом. Встановлено, що специфіка освітнього процесу, яка характеризується високою щільністю перехресних взаємозв'язків та циклічністю, вимагає від архітектури бази даних суворого контролю посилальної цілісності та запобігання просторово-часовим колізіям на інфраструктурному рівні.

Порівняльний огляд архітектурних рішень в існуючих відкритих системах (UniTime, RosarioSIS, Moodle) та вітчизняних (система на базі 1С:Підприємство, модуль A.S.T.S. Schedule) – дозволив виявити ключові технологічні прогалини. Зокрема, показано, що відсутність шару абстракції (Transaction Script) призводить до змішування інфраструктурного коду з шаром представлення, ускладнюючи супровід. Аналіз локальних розробок додатково підтвердив, що закрита корпоративна платформа та відсутність нормалізованого реляційного сховища зі спеціалізованим DAL обмежують масштабованість і транзакційну надійність систем рівня університету. Використання кастомних ізольованих обгорток типу DBAL часто перекладає контроль цілісності з рівня СКБД на програмний код. Водночас монолітне використання повного об'єктно-реляційного відображення здатне створювати продуктивний overhead при виконанні складних аналітичних запитів.

Цей аналіз доводить необхідність пошуку гнучких гібридних підходів до організації доступу до даних.

Систематизація математичного апарату реляційної алгебри та транзакційних моделей дозволила формалізувати теоретичні вимоги до майбутнього сховища. Визначено, що для усунення інформаційної надлишковості та мінімізації ризиків виникнення аномалій модифікації логічна схема має задовольняти критерії нормалізації. Забезпечення цілісності даних при паралельному доступі диспетчерів вимагає від майбутньої платформи суворого дотримання транзакційної моделі ACID.

Зіставлення методологій Code-First та DB-First дозволило окреслити можливі вектори управління змінами схеми бази даних під впливом динамічних освітніх стандартів.

Дослідження софтверних патернів (Data Mapper, Repository, Unit of Work, DTO) сформувало теоретичну базу для розуміння методів подолання об'єктно-реляційної розбіжності. Проаналізовано підходи до ізоляції доменної моделі, оптимізації інформаційного трафіку та централізації логіки запитів.

РОЗДІЛ 2
ВИЗНАЧЕННЯ ВИМОГ ТА ПРОЄКТУВАННЯ АРХІТЕКТУРИ ШАРУ ДОСТУПУ ДО ДАНИХ

2.1 Формування технічного завдання та вимог до підсистеми збереження даних

Проєктний етап розробки архітектури сучасних інформаційних систем, орієнтованих на автоматизацію управління навчальним процесом у закладах вищої освіти, вимагає детальної формалізації вимог до підсистеми збереження та доступу до даних [30]. Специфіка даної предметної області полягає у високій щільності зв'язків між сутностями, що вимагає дотримання необхідного рівня нормалізації. Шар доступу до даних разом із реляційною базою даних утворює інфраструктурний фундамент застосунку, від надійності якого залежить стійкість системи під навантаженням, збереженість персональних та академічних даних, а також відсутність логічних суперечностей у сформованому розкладі [31]. Постановка задачі передбачає розробку вимог, що унеможливлюють дестабілізацію сховища при конкурентному доступі кількох користувачів і забезпечують гнучке керування інформаційними графами предметної області.

З точки зору функціональних можливостей, підсистема збереження даних повинна оперувати складною ієрархічною структурою сутностей і забезпечувати повний цикл базових операцій маніпулювання даними (CRUD) [10].

Специфіка предметної області вимагає від DAL підтримки таких функціональних параметрів:

- Управління багатошаровими довідниками: База даних повинна забезпечувати цілісне збереження організаційної структури університету, кадрового складу та аудиторного фонду з урахуванням його місткості й технічного обладнання [2].

- Моделювання та збереження навчально-методичних планів: Шар даних має підтримувати реляційні зв'язки між освітніми програмами, галузями знань, спеціальностями та безпосередньо нормативними чи вибірковими дисциплінами із чітким закріпленням кількості кредитів, семестрів та форм контролю [1].
- Фіксація академічного навантаження викладачів: Підсистема зобов'язана об’єднувати інформацію про розподіл годин навчальних планів між конкретними співробітниками кафедр у межах визначених семестрів та потоків студентських груп.
- Транзакційний запис семестрового розкладу: Головною функціональною задачею є збереження підсумкових занять із координацією дати, часового слоту (номера пари), аудиторії, викладача та множини залучених академічних груп чи підгруп однією цільною транзакцією.

Найважливішою інженерною вимогою до підсистеми збереження даних на стику функціональних та нефункціональних параметрів є кінцеве забезпечення просторово-часової узгодженості розкладу.

Хоча логічна перевірка накладань занять здійснюється сервісами бізнес-логіки, саме шар доступу до даних має виступати останнім фізичним бар'єром, що запобігає виникненню колізій подвійного бронювання при паралельній роботі кількох керівників розкладу [32]. Адже в умовах високого навантаження і спроб одночасного запису, саме жорсткі правила СКБД рятують систему від збереження некоректних даних. На рівні архітектури сховища це трансформується у вимогу щодо створення унікальних композитних індексів та обмежень, які на рівні рушія СКБД забороняють існування суперечливих станів, таких як призначення одного викладача або однієї аудиторії на один і той самий часовий слот у межах однієї календарної дати.

Нефункціональні вимоги визначають критерії якості, швидкодії та експлуатаційної надійності інфраструктурного шару системи під час активного використання:

1. Суворе дотримання транзакційної моделі ACID: Будь-яка модифікація розкладу є комплексною операцією, яка зачіпає масиви пов'язаних таблиць. Шар доступу до даних повинен гарантувати атомарність таких дій: у разі виникнення помилки на будь-якому етапі запису підсистема має виконати повний відкат для запобігання дестабілізації сховища [33].
2. Стійкість до конкурентних запитів та ізоляція транзакцій: Оскільки адміністрування розкладу виконується багатьма операторами одночасно, DAL повинен забезпечувати захист від аномалій паралельного доступу, зокрема стану «втраченого оновлення» [34], коли зміни одного диспетчера перезаписуються даними іншого. Цю вимогу закрито на двох рівнях: рушій InnoDB забезпечує блокування на рівні рядків у межах транзакції, а реєстрація DbContext зі Scoped-lifetime ізолює контекст змін у межах одного HTTP-запиту, унеможливлюючи змішування станів між паралельними сесіями.
3. Висока швидкодія при виконанні складних багатотабличних вибірок: Процес формування сітки розкладу для клієнтської частини вимагає миттєвої агрегації інформації шляхом об'єднання великої кількості таблиць. Час відгуку підсистеми на операції читання має бути мінімальним, що накладає вимогу щодо обов'язкового проєктування та утримання некластерних індексів на всіх полях зовнішніх ключів (Foreign Keys) [35].
4. Захист історичної цілісності даних: Навчальний процес є динамічним, проте нормативні зміни не повинні руйнувати архівні дані розкладів за минулі семестри. Відтак, технічне завдання вимагає конфігурації жорстких обмежень посилальної цілісності за стратегією блокування каскадних операцій видалення, що локалізує зміни в межах поточного часового періоду й унеможливлює випадкове знищення історичних звітів [32].

2.2 Концептуальне моделювання та структура даних

Перехід від сформованих вимог до програмної реалізації вимагає побудови чіткої концептуальної моделі сховища, яка формалізує всі сутності предметної області та зв'язки між ними. Для початку пропонується спроєктувати логічну та фізичну структуру реляційної бази даних. Така пріоритетність гарантує, що фундамент системи створюється з урахуванням суворих правил нормалізації, усуваючи ризики дублювання даних чи виникнення аномалій під час їх оновлення.

Спроєктована схема бази даних виступає центральним сховищем, що агрегує нормативно-довідкову інформацію та забезпечує транзакційну підтримку процесу складання розкладу [36].

2.2.1 Виділення ключових сутностей предметної області

Структура бази даних проєктується у вигляді системи нормалізованих відношень, які концептуально поділяються на довідкові та транзакційні.

Серед довідкових сутностей – таблиця співробітників, що зберігає персональні дані, посаду та належність до кафедри. Аудиторний фонд представлений сутністю приміщень, кожен екземпляр якої визначає фізичний номер, місткість та класифікаційний тип аудиторії (наприклад, лекційна чи комп'ютерний клас).

Контингент здобувачів освіти формалізується через сутність академічних груп, що містить назву, рік вступу та кількість студентів – ключовий параметр для розрахунку необхідної місткості аудиторій.

Взаємозв’язки між ключовими сутностями блоку кадрових даних продемонстровано на рисунку 2.1.

Окремий структурний блок утворюють концептуальні моделі навчального процесу який продемонстровано на рисунку 2.2.

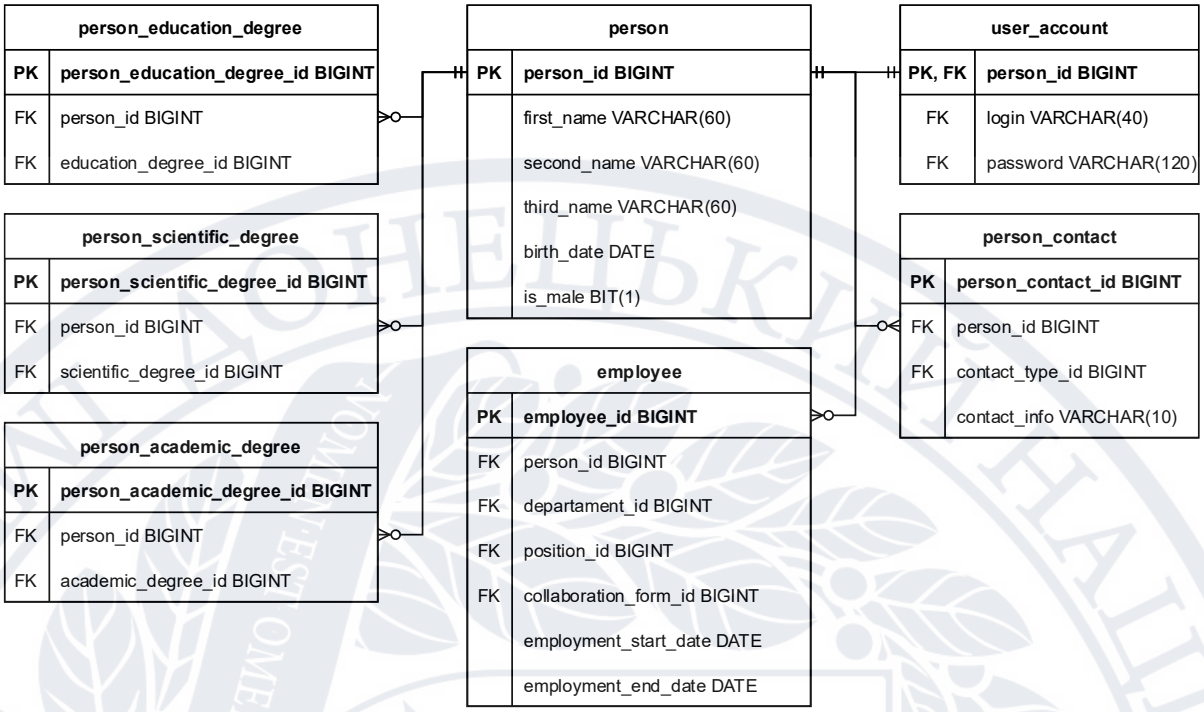

Рисунок 2.1 – EER діаграма фрагменту бази даних: Блок кадрових даних

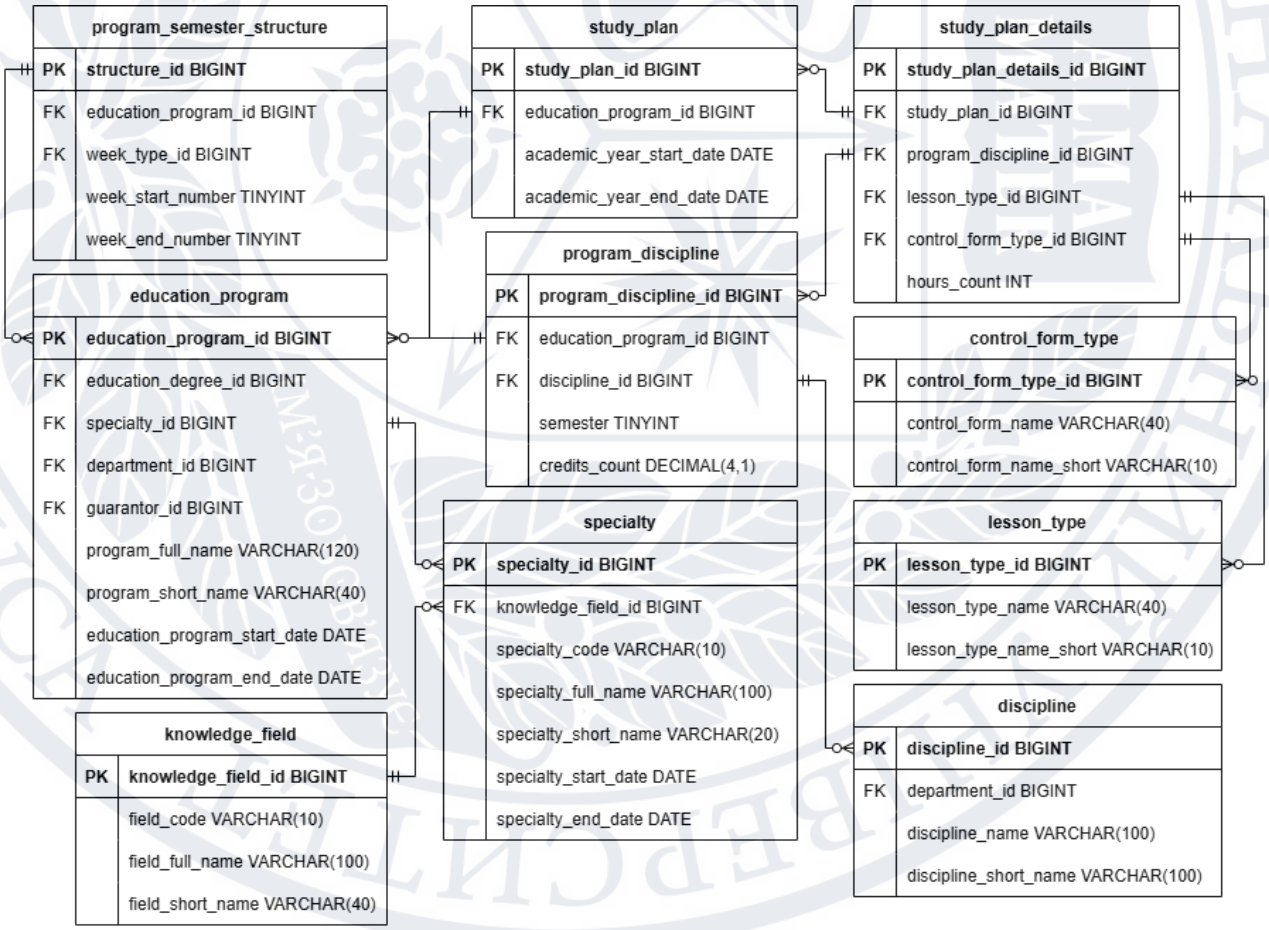

Рисунок 2.2 – EER діаграма фрагменту бази даних: Блок навчального процесу

Сутність деталей навчального плану виступає агрегатором, який об'єднує конкретну дисципліну, тип заняття та нормативну кількість виділених годин. Для дискретизації навчального часу проєктується довідник часових слотів, який визначає порядковий номер пари, день тижня та ознаку парності на чисельник або знаменник.

Ключовою транзакційною сутністю є запис розкладу – саме вона зв'язує всі довідники в одне ціле. Вона проєктується як композиція посилань на конкретного викладача, аудиторію, часовий слот та елемент навчального плану, додатково фіксуючи точну календарну дату проведення заняття.

2.2.2 Формалізація зв'язків та обмежень цілісності

Для забезпечення структурної надійності сховища концептуальні зв'язки формалізуються через механізми первинних та зовнішніх ключів.

Більшість відношень у системі проєктується за принципом «один-до-багатьох» (1:N). Наприклад, один працівник може бути призначений на багато занять, але конкретний запис розкладу завжди посилається лише на одного викладача.

Окремої уваги вимагає зв'язок між заняттям та академічними групами. Оскільки одна лекція може проводитися для кількох груп одночасно, або цілого потоку, а одна група відвідує багато різних занять між сутностями розкладу та студентської групи виникає відношення «багато-до-багатьох» (N:M) [37, 38]. У реляційній моделі цей архітектурний виклик вирішується шляхом введення проміжної сутності, яка зберігає пари ідентифікаторів розкладу та групи.

Для виконання вимог щодо захисту архівних даних планується накладення жорстких обмежень на зовнішні ключі центральної транзакційної таблиці де застосовуватиметься стратегія блокування каскадного видалення. Це гарантуватиме, що рушій СКБД заблокує операцію видалення викладача або аудиторії з довідника, якщо вони вже фігурують у збереженому розкладі.

Водночас для проміжної асоціативної сутності груп доцільно застосувати каскадне видалення для коректного очищення зв'язків при скасуванні самого заняття [32]. Запобігання просторово-часовим колізіям на фізичному рівні досягається шляхом проєктування унікальних композитних індексів, що математично унеможливлює створення дублікатів перетинів ресурсів. Загальна концептуальна структура блоку розкладу наведена на рисунку 2.3.

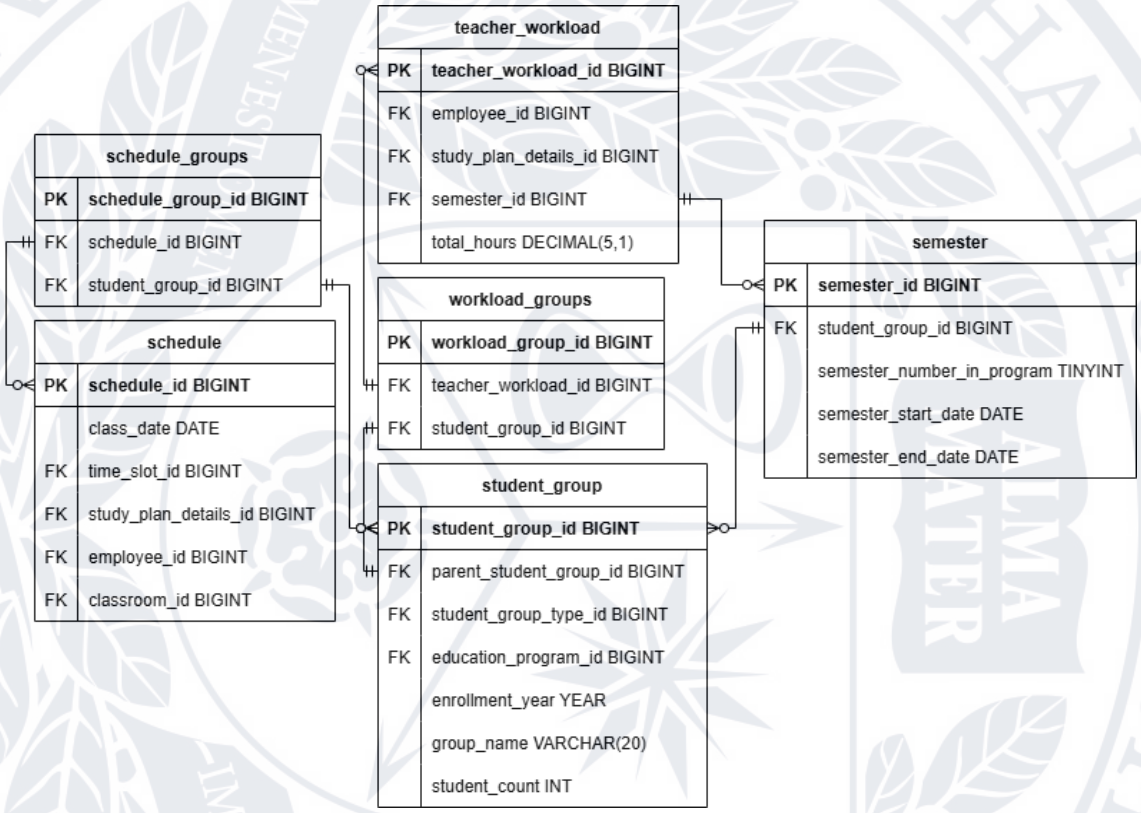

Рисунок 2.3 – EER діаграма фрагменту бази даних: Блок розкладу

2.3 Проєктування архітектури шару доступу до даних

Спроєктована концептуальна модель реляційної бази даних потребує створення надійного програмного інтерфейсу для взаємодії з нею. Архітектура шару доступу до даних розробляється з урахуванням принципів слабкої зв'язаності та високої згуртованості, щоб забезпечити масштабованість системи та зручність її подальшого супроводу [39]. Саме тому прямі звернення до таблиць замінюються абстрагованими контрактами, що надійно ізолює бізнес-логіку генерації розкладу від низькорівневих деталей роботи рушія СКБД.

2.3.1 Архітектурний підхід та ізоляція компонентів

Програмний застосунок будується за багатошаровою архітектурою, де шар доступу до даних виділяється в окремий інфраструктурний модуль [31].

Головною архітектурною вимогою є те, що компоненти вищого рівня не повинні мати прямої залежності від специфіки конкретної СКБД чи класів ORM-фреймворку [40]. Взаємодія між шаром бізнес-логіки та інфраструктурним шаром відбувається виключно через абстрактні контракти визначені на рівні домену. Це дозволяє абстрагувати логіку збереження та проводити модульне тестування бізнес-правил без підключення до реальної бази даних [41]. Логіку ізольованої перевірки бізнес-правил відображено на рисунку 2.4.

Рисунок 2.4 – UML діаграма процесу модульного тестування бізнес-логіки з використанням інфраструктурних заглушок

2.3.2 Специфікація інфраструктурних патернів

На практиці ефективна реалізація підсистеми неможлива без гнучкого і стійкого інфраструктурного фундаменту. З метою централізації логіки формування запитів, надійного управління транзакціями та ізоляції бізнес-логіки від інфраструктури, у шар доступу до даних необхідно інтегрувати класичні патерни корпоративного програмного забезпечення.

Застосування такого архітектурного підходу повністю усуває жорстку зв'язність між сервісним рівнем застосунку та низькорівневими механізмами реляційної бази даних, що значно спрощує подальшу підтримку системи, створює умови для модульного тестування компонентів та гарантує суворе дотримання транзакційної цілісності. Центральними елементами цієї концепції є такі шаблони проектування:

Узагальнений репозиторій (Generic Repository): Проєктується базовий параметризований контракт, який абстрагує стандартні операції маніпулювання даними для будь-якої доменної сутності, що суттєво мінімізує дублювання коду завдяки одноразовій реалізації логіки доступу на інфраструктурному рівні. Параметризація типом сутності робить контракт універсальним: один інтерфейс однаково обслуговує як прості довідникові таблиці, так і складні транзакційні сутності. Спеціалізовані репозиторії успадковують базову реалізацію і розширюють її лише предметно-специфічними методами вибірки без дублювання загальної логіки.

Одиниця роботи (Unit of Work): Для координації багатотабличних операцій проєктується єдиний інтерфейс-координатор. Він виступає загальним фасадом для всіх репозиторіїв системи та гарантує, що сукупність змін, накопичених в межах одного бізнес-процесу, фіксується в базі даних як єдина атомарна транзакція. У разі виникнення системного винятку патерн забезпечує коректне відхилення операції та повернення стану об'єктів. Це є критично важливим для предметної області системи розкладу, де будь-яка модифікація зачіпає одночасно кілька взаємопов'язаних таблиць, і часткове збереження змін

неминуче призвело б до порушення цілісності сховища. Додатково, використання відкладеної ініціалізації репозиторіїв усередині цього патерну оптимізує споживання оперативної пам'яті сервером [42].

Загальну структурну концепцію взаємодії цих патернів та абстракцій інфраструктурного шару проілюстровано на рисунку 2.5.

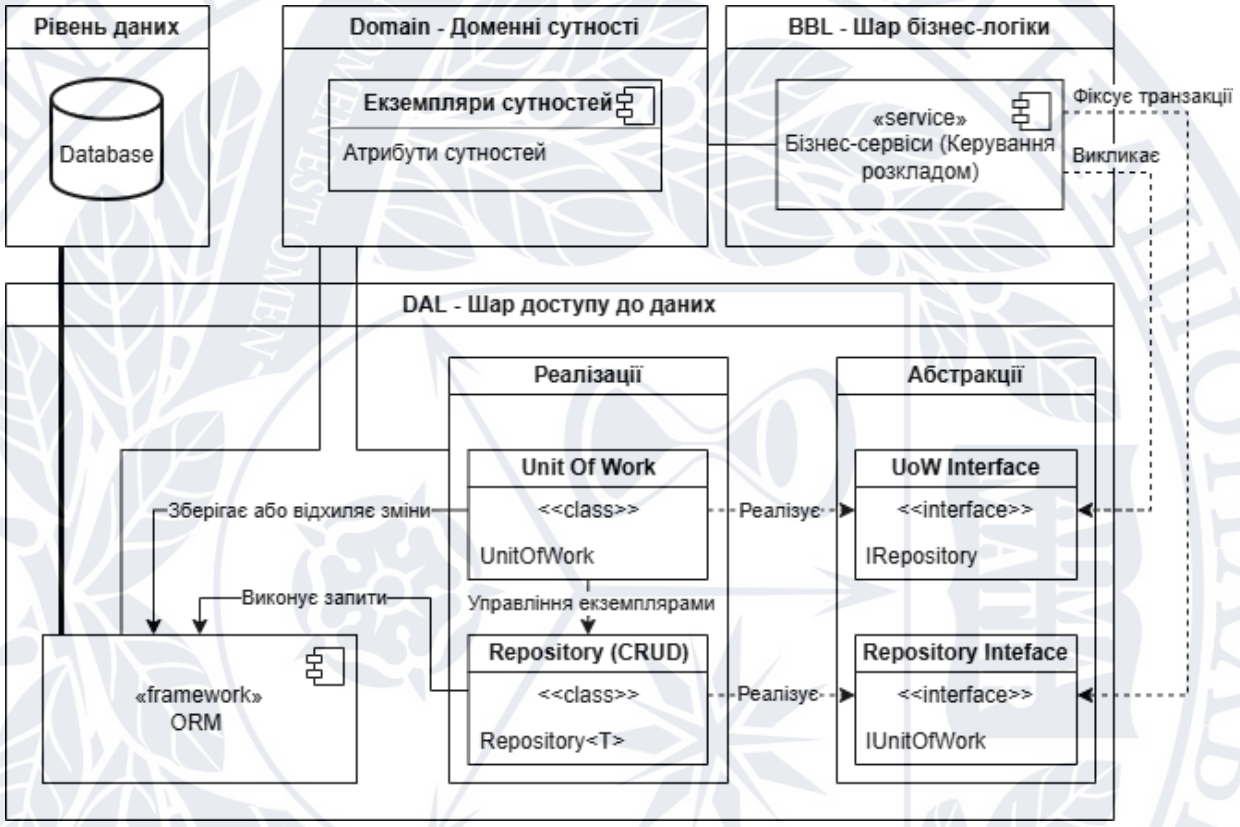

Рисунок 2.5 – Структурна схема взаємодії компонентів шару доступу до даних

2.3.3 Відображення реляційної схеми на об'єктну модель

Оскільки фізична схема бази даних є первинним джерелом правди та оптимізується безпосередньо на рівні SQL без прихованого втручання ORM-фреймворку, для проєкту обрано методологію Database-First – схема розробляється наперед засобами візуального проєктування реляційних баз даних, а обраний ORM-фреймворк використовуватиметься виключно для генерації об'єктних моделей на основі готової реляційної схеми [26].

Враховуючи використання цього підходу, виникає архітектурна необхідність точного відображення реляційних атрибутів на властивості об'єктно-орієнтованих доменних моделей. Для досягнення чистоти коду архітектура виключає використання атрибутів конфігурації безпосередньо у класах сутностей. Натомість застосовується підхід розділеної конфігурації через Fluent API [43]. Для кожної доменної моделі проєктується окремий конфігураційний клас. У цих спеціалізованих класах декларативно визначаються всі правила мапінгу: назви таблиць, первинні ключі, конвертація типів, зв'язки та стратегії каскадних операцій. Такий розподіл дозволяє зберігати доменні класи ідеально чистими структурами даних, тоді як уся інфраструктурна логіка відображення зосереджується в ізольованих файлах, що повною мірою задовольняє принцип єдиної відповідальності.

Висновок до другого розділу

У другому розділі сформовано технічне завдання та визначено повний комплекс функціональних і нефункціональних вимог до підсистеми збереження даних. Встановлено, що критичними інженерними обмеженнями є гарантія транзакційної атомарності при комплексних операціях редагування розкладу, захист від аномалій паралельного доступу та збереження цілісності архівних даних.

Побудовано концептуальну модель реляційної бази даних, в якій виділено довідкові та транзакційні сутності предметної області. Формалізовано зв'язки між ними: більшість відношень реалізовано за принципом «один-до-багатьох», а зв'язок між записами розкладу та академічними групами через асоціативну проміжну сутність, що вирішує відношення «багато-до-багатьох». Визначено стратегії каскадних операцій: обмежувальне видалення для захисту архівних записів та каскадне – для очищення асоціативних зв'язків при скасуванні заняття. Концептуальну структуру предметної області проілюстровано EER-діаграмами (рисунки 2.1–2.3).

Спроєктовано архітектуру шару доступу до даних на основі багатошарового підходу з повною ізоляцією інфраструктурного модуля від шарів бізнес-логіки через абстрактні контракти. Визначено склад інфраструктурних патернів: узагальнений репозиторій централізує логіку формування запитів і забезпечує тестованість компонентів тоді як патерн Unit of Work координує багатотабличні операції в межах єдиної атомарної транзакції. Підхід розділеної конфігурації через Fluent API гарантує чистоту доменних класів та дотримання принципу єдиної відповідальності.

Таким чином, результати другого розділу утворюють повноцінну архітектурну специфікацію, яка слугує безпосередньою основою для практичної реалізації бази даних та шару доступу до даних у третьому розділі роботи.

РОЗДІЛ 3
РЕАЛІЗАЦІЯ БАЗИ ДАНИХ ТА ШАРУ ДОСТУПУ ДО ДАНИХ

3.1 Обґрунтування вибору технологічного стеку та платформи розробки

Проєктування стабільного інфраструктурного шару вимагає використання інструментів, які здатні забезпечити високу продуктивність, безпеку типів та гнучкість масштабування. Вибір технологічного стеку для підсистеми доступу до даних базується на необхідності виконання вимог, сформованих у попередньому підрозділі, та орієнтований на створення надійної корпоративної архітектури [30].

3.1.1 Вибір платформи та мови програмування

Основною технологічною платформою для розробки програмної частини підсистеми обрано екосистему .NET 8 у поєднанні з об'єктно-орієнтованою мовою програмування C# [44].

Кросплатформена природа .NET 8 дозволяє розгортати підсистему як у середовищі Windows, так і на базі Linux-контейнерів без зміни кодової бази.

Важливим фактором є сувора статична типізація мови C#, яка мінімізує ризик виникнення помилок під час мапінгу реляційних даних у доменні моделі, гарантуючи передбачуваність роботи програми [45]. Крім того, екосистема .NET має потужний нативний контейнер впровадження залежностей. Це є критично важливим для реалізації слабкозв'язаної багатошарової архітектури, оскільки дозволяє легко впроваджувати абстракції патернів Repository та Unit of Work у бізнес-сервіси, забезпечуючи логічну ізоляцію шару доступу до даних від верхніх рівнів застосунку. Вирішальною перевагою C# є технологія LINQ, що забезпечує типобезпечне формування запитів.

3.1.2 Вибір системи керування базами даних

Для організації фізичного сховища даних обрано реляційну систему керування базами даних MySQL.

Специфіка освітнього процесу формує щільний інформаційний граф із великою кількістю перехресних зв'язків між викладачами, аудиторіями, дисциплінами та академічними групами для якого реляційна парадигма є найбільш доцільним рішенням [10]. Використання підсистеми зберігання InnoDB забезпечує повну підтримку транзакційної моделі ACID та блокування на рівні рядків [46, 47], що гарантує цілісність даних при конкурентному доступі кількох диспетчерів. Підтримка обмежень зовнішніх ключів зі стратегією блокування каскадного видалення та композитних унікальних індексів формує надійний бар'єр проти подвійного бронювання ресурсів. Незважаючи на вищі показники масштабованості PostgreSQL, ця перевага є нерелевантною для системи, що розрахована на роботу в межах одного університету з обмеженою кількістю користувачів і не потребує горизонтального масштабування.

3.1.3 Технології об'єктно-реляційного мапінгу

Для подолання об'єктно-реляційної розбіжності та абстрагування взаємодії між програмною частиною і БД інтегровано сучасний фреймворк EF Core.

Важливою архітектурною особливістю проєкту є застосування методології Database-First замість популярного підходу Code-First [26]. Такий вибір є свідомим інженерним рішенням, де первинною точкою правди виступають нативні DDL-скрипти. Це надає архітектору тотальний контроль над типами даних СКБД, оптимізацією індексів та правилами каскадних операцій без прихованого втручання ORM-фреймворку. Інструментарій зворотного проєктування використовується виключно для автоматичної трансляції готової реляційної схеми у чисті об'єктні моделі даних та конфігурації мапінгу. Це

суттєво економить час розробки, гарантуючи при цьому стовідсоткову відповідність програмних сутностей фізичним таблицям.

Окрім цього, EF Core надає можливість формувати складні багатотабличні запити з використанням типізованого синтаксису LINQ [48]. Це переносить перевірку коректності SQL-запитів на етап компіляції, унеможливлює SQL-ін'єкції та значно спрощує розробку узагальнених репозиторіїв для агрегації сітки розкладу [49].

3.1.4 Відповідність інструментарію визначеним вимогам

Етап переходу від аналітичного опису архітектури до її безпосереднього програмного втілення вимагає детальної верифікації кожного інфраструктурного компонента. Системний підхід до розробки програмного забезпечення передбачає, що вибір засобів об'єктно-реляційного відображення, середовища розробки та системи керування базами даних має базуватися на їхній здатності задовольняти виявлені критерії продуктивності, масштабованості та безпеки. Саме тому виникає потреба чіткого архітектурного співставлення потенціалу платформи із кожним окремим викликом автоматизації навчального процесу.

Це дозволяє обґрунтувати доцільність конкретних інструментів розробки та гарантує передбачувану і стабільну поведінку системи під реальним навантаженням.

Для наочної демонстрації того, як саме обраний технологічний стек закриває сформовані на етапі системного аналізу вимоги, розроблено узагальнюючу матрицю відповідності (таблиця 3.1). Вона відображає послідовну трансформацію абстрактних бізнес-правил предметної області у конкретні технічні рішення на рівні бази даних та інфраструктурного коду. Наведена матриця слугує архітектурним підтвердженням того, що спроєктований шар доступу до даних є не просто набором програмних компонентів, а цілісним, логічно обґрунтованим інженерним рішенням, спрямованим на забезпечення

максимальної надійності, транзакційної безпеки та просторово-часової узгодженості даних університетського розкладу.

Таблиця 3.1 Матриця відповідності вимог до підсистеми збереження даних інфраструктурним інструментам

Категорія вимоги	Зміст вимоги з технічного завдання	Технічний інструмент реалізації в БД / DAL
Функціональна	Запобігання просторово-часовим колізіям (подвійне бронювання аудиторій, накладки викладачів).	Створення унікальних композитних ключів (UNIQUE KEY) на рівні СКБД MySQL (наприклад, комбінація ідентифікаторів розкладу та групи).
Функціональна	Захист історичної цілісності даних від випадкового руйнування (наприклад, при зміні складу кафедри).	Конфігурація обмежень зовнішніх ключів за стратегією ON DELETE RESTRICT у DDL-скриптах та класах конфігурації Fluent API.
Нефункціональна	Забезпечення транзакційної атомарності (ACID) при комплексному редагуванні розкладу або навантаження.	Використання рушія InnoDB та механізму DbContext.SaveChangesAsync() в інкапсульованому патерні Unit of Work.
Нефункціональна	Оптимізація швидкодії при виконанні складних JOIN-запитів сітки розкладу для факультету.	Автоматичне створення некластерних індексів (HasIndex / KEY) на всіх полях зовнішніх ключів (Foreign Keys) транзакційних таблиць.

3.2 Реалізація фізичної схеми бази даних

Відповідно до методології Database-First, розробка системи розпочалась не з написання коду, а зі створення фізичної схеми реляційної бази даних. Саме DDL-скрипт виступає єдиною первинною точкою правди: всі подальші рівні системи – об'єктні моделі, конфігурації ORM, сервіси – є відображенням того, що визначено на рівні СКБД. Такий підхід забезпечує повний і явний контроль

над типами даних, іменуванням колонок, стратегіями зовнішніх ключів та індексами ще до написання першого рядка C#-коду.

Увесь скрипт починається з директиви SET NAMES UTF8, що встановлює кодування з'єднання, і всі таблиці створюються з ENGINE=InnoDB DEFAULT CHARSET=utf8. Вибір рушія InnoDB є принциповим: саме він забезпечує підтримку транзакцій, блокування на рівні рядків та механізму зовнішніх ключів, без яких реалізація вимог цілісності даних, описаних у другому розділі, була б неможливою [19].

Схема охоплює 44 таблиці, які логічно поділяються на шість тематичних блоків. Розглянемо кожен із них.

Першим у скрипті визначено блок часової дискретизації навчального процесу. Таблиця semester_week_type зберігає типи тижнів: навчальний, канікули, сесія тощо. Кожен тип має повну назву (week_type_name, до 60 символів) і скорочений код (week_type_code, до 10 символів).

Обидва поля захищені унікальними обмеженнями, а перша пара додатково об'єднана у складений унікальний ключ – це унеможливлює ситуацію, коли одна назва відповідає різним кодам. Таблиця time_slot фіксує конкретні часові слоти пар: порядковий номер, час початку та завершення, ознаку активності та дату введення в дію поля when_applied.

Це дозволяє системі підтримувати зміну розкладу дзвінків у різні роки без втрати архівних даних. Сутність time_slot продемонстровано у лістингу 3.1.

Лістинг 3.1 – DDL код створення таблиці часового слоту

```
CREATE TABLE IF NOT EXISTS `time_slot`
(
     `time_slot_id` BIGINT UNSIGNED AUTO_INCREMENT PRIMARY KEY
    ,`slot_number` TINYINT(1) UNSIGNED NOT NULL
    ,`start_time` TIME NOT NULL
    ,`end_time` TIME NOT NULL
    ,`is_active` BOOL NOT NULL
    ,`when_applied` DATE NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Центральною таблицею блоку персональних та кадрових даних є person – вона зберігає базову інформацію про фізичну особу: ім'я, прізвище, по батькові (кожне до 60 символів), дату народження та стать. Таблиця навмисно позбавлена будь-якої ролевої інформації: один і той самий запис person може бути пов'язаний і з employee (працівник), і з user_account (обліковий запис системи). Такий підхід відображає реальність – людина в університеті може виступати одночасно в кількох ролях.

Кваліфікаційні характеристики викладача реалізовані через три окремі асоціативні таблиці: вчені звання (person_academic_degree), наукові ступені (person_scientific_degree) і освітні ступені (person_education_degree). Це принципово відрізняється від плоскої моделі, де все зберігалося б в одному рядку employee. Причина архітектурна: викладач може мати кілька вчених звань або поєднувати два наукові ступені, і асоціативна структура точно відображає такий стан без дублювання.

Кожна з цих таблиць має складений унікальний ключ на парі (person_id, degree_id), що на рівні СКБД унеможливлює дублювання одного й того самого ступеня для однієї особи.

Таблиця person_contact зберігає контактну інформацію з посиланням на тип контакту (contact_type – email, телефон тощо). Унікальний ключ на трійці (person_id, contact_type_id, contact_info) дозволяє мати кілька контактів одного типу, проте виключає точні дублікати.

Таблиця user_account особлива тим, що її первинний ключ – це одночасно і зовнішній ключ на person. Таким чином, один запис person не може мати двох облікових записів. Також тут застосовано один з небагатьох випадків використання каскадного видалення коли при видаленні особи з системи її обліковий запис видаляється автоматично, що логічно: без особи немає сенсу зберігати логін і пароль.

Таблиця employee є кадровою карткою викладача. Вона пов'язує конкретну особу (person) з підрозділом (department), посадою (position) та формою співпраці (collaboration_form – штат, сумісництво, погодинна оплата). Поле

employment_end_date допускає значення NULL – якщо воно заповнене, це означає, що співробітник вже не працює в університеті.

Організаційна структура університету реалізована через таблицю department із self-referencing зовнішнім ключем. Поле parent_department_id є NULL-able і посилається на саму таблицю department, що продемонстровано у лістингу 3.2.

Лістинг 3.2 – Фрагмент DDL для створення таблиці department

```
CREATE TABLE IF NOT EXISTS `department`
(
     `department_id` BIGINT UNSIGNED AUTO_INCREMENT PRIMARY KEY
    ,`department_type_id` BIGINT UNSIGNED NOT NULL
    ,`building_id` BIGINT UNSIGNED NOT NULL
    ,`parent_department_id` BIGINT UNSIGNED NULL
    ,`full_name` VARCHAR(120) NOT NULL
    ,`short_name` VARCHAR(20) NOT NULL
    ,UNIQUE(`full_name`)
    ,UNIQUE(`short_name`)
    ,FOREIGN      KEY      (`department_type_id`)       REFERENCES
`department_type`(`department_type_id`)
          ON UPDATE CASCADE ON DELETE RESTRICT
    ,      FOREIGN      KEY       (`building_id`)       REFERENCES
`building`(`building_id`)
          ON UPDATE CASCADE ON DELETE RESTRICT
    ,    FOREIGN    KEY    (`parent_department_id`)    REFERENCES
`department`(`department_id`)
          ON UPDATE CASCADE ON DELETE RESTRICT
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Це дозволяє представити будь-яку глибину ієрархії – від університету на верхньому рівні до лабораторії кафедри на нижньому – в одній таблиці без жодної зміни схеми. Унікальні обмеження на full_name і short_name гарантують відсутність підрозділів-дублікатів. В свою чергу заборона каскадного видалення для self-referencing ключа є критично важливою: вона не дозволяє видалити підрозділ, якщо він є батьківським для інших підрозділів.

Кожен підрозділ прив'язаний до корпусу (building), який у свою чергу має повну адресу через ланцюжок building → address → street → city → region. Така

нормалізація адресних даних є надмірною для більшості систем, проте виправдана в контексті ЗВО, де корпуси університету можуть знаходитися в різних містах.

Аудиторний фонд представлений таблицею classroom з характеристиками: тип (classroom_type – лекційна, комп'ютерна, лабораторна), кількість місць, наявність дошки і проєктора. Додаткове обладнання аудиторії зберігається в асоціативній таблиці classroom_equipment_to_classroom – вона фіксує не лише наявність обладнання, а й його кількість через поле classroom_equipment_amount.

Для зовнішніх ключів цієї таблиці налаштоване каскадне видалення де при видаленні аудиторії або одиниці обладнання всі відповідні записи очищуються автоматично.

Блок навчального плану та освітньої програми є найбільш розгалуженим за кількістю таблиць і відображає нормативну основу навчального процесу. Ланцюжок залежностей виглядає так: галузь знань (knowledge_field) → спеціальність (specialty) → освітня програма (education_program) → дисципліна програми (program_discipline) → деталь навчального плану (study_plan_details).

education_program – це конкретна освітня програма кафедри, яка прив'язана до ступеня (education_degree), спеціальності, підрозділу та гаранта програми (викладач). Всі чотири зовнішні ключі мають сувору заборону на каскадне видалення – жоден з цих елементів не можна видалити, поки програма на нього посилається.

program_discipline є таблицею, яка пов'язує освітню програму з конкретною дисципліною і додає атрибути цього зв'язку: номер семестру і кількість кредитів. Без цієї сутності неможливо відобразити той факт, що одна й та сама дисципліна може входити в різні освітні програми з різною кількістю кредитів у різних семестрах.

study_plan_details – найбільш деталізована нормативна таблиця. Вона розбиває рядок навчального плану за типами занять: одна дисципліна в одному семестрі може мати окремі рядки для лекцій, практичних і лабораторних. Поле

hours_count в свою чергу фіксує нормативну кількість годин для конкретного типу заняття.

program_semester_structure визначає шаблонну структуру тижнів для кожної освітньої програми: які тижні є навчальними, які – сесійними, які – канікулярними. Ця таблиця використовується при генерації дат занять: саме завдяки ній майбутні сервіси зможуть коректно розставляти заняття на навчальні тижні необхідного типу.

У блоці груп, семестрів та навантаження таблиця student_group має ту саму self-referencing архітектуру, що й department. Поле parent_student_group_id дозволяє моделювати тристепеневу ієрархію: потік → група → підгрупа. Потік не є групою в звичайному розумінні, а лише об'єднанням груп для проведення лекцій; підгрупи утворюються з основної групи для практичних і лабораторних занять. Поле student_count є NULL-able – для потоку ця величина не визначена, для підгрупи може відрізнятися від основної групи.

Таблиця semester (семестр) зберігає конкретні дати семестрів для конкретного потоку: semester_start_date, semester_end_date та номер семестру в програмі. Зв'язок із student_group, а не з освітньою програмою, є свідомим рішенням: різні потоки одного напряму підготовки можуть починати і закінчувати семестр у різні дати.

Сутність teacher_workload фіксує, який викладач веде конкретний тип заняття з конкретної дисципліни для конкретного потоку в конкретному семестрі, із зазначенням загальної кількості годин.

Пов'язана таблиця workload_groups розкриває, для яких саме груп виконується це навантаження – так само через асоціативну структуру.

Таблиця schedule є центральним вузлом усієї бази даних, заради якого побудовані всі попередні блоки. Кожен рядок цієї таблиці є фактом конкретного заняття: дата (class_date), часовий слот, деталь навчального плану (а через неї – дисципліна і тип заняття), викладач та аудиторія. DLL код даної таблиці представлено у лістингу 3.3.

Лістинг 3.3 – Скрипт створення таблиці «розклад»

```
CREATE TABLE IF NOT EXISTS `schedule`
(
     `schedule_id` BIGINT UNSIGNED AUTO_INCREMENT PRIMARY KEY
    ,`class_date` DATE NOT NULL
    ,`time_slot_id` BIGINT UNSIGNED NOT NULL
    ,`study_plan_details_id` BIGINT UNSIGNED NOT NULL
    ,`employee_id` BIGINT UNSIGNED NOT NULL
    ,`classroom_id` BIGINT UNSIGNED NOT NULL
    ,FOREIGN     KEY     (`study_plan_details_id`)      REFERENCES
`study_plan_details`(`study_plan_details_id`)
        ON UPDATE CASCADE ON DELETE RESTRICT
    ,FOREIGN        KEY         (`employee_id`)         REFERENCES
`employee`(`employee_id`)
        ON UPDATE CASCADE ON DELETE RESTRICT
    ,FOREIGN        KEY         (`classroom_id`)        REFERENCES
`classroom`(`classroom_id`)
        ON UPDATE CASCADE ON DELETE RESTRICT
    ,FOREIGN        KEY         (`time_slot_id`)        REFERENCES
`time_slot`(`time_slot_id`)
        ON UPDATE CASCADE ON DELETE RESTRICT
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Усі чотири зовнішні ключі мають сувору заборону на каскадне видалення, без винятків – жоден з елементів розкладу не може бути видалений, поки він фігурує в записах занять.

Відношення «багато-до-багатьох» між заняттям і академічними групами реалізоване через schedule_groups. Ця таблиця заслуговує на окрему увагу через асиметрію своїх зовнішніх ключів. Для наочності представлено лістинг 3.4.

Лістинг 3.4 – DDL код створення таблиці schedule_groups

```
CREATE TABLE IF NOT EXISTS `schedule_groups`
(
    `schedule_group_id` BIGINT UNSIGNED AUTO_INCREMENT PRIMARY KEY
    ,`schedule_id` BIGINT UNSIGNED NOT NULL
    ,`student_group_id` BIGINT UNSIGNED NOT NULL
    ,FOREIGN        KEY         (`schedule_id`)         REFERENCES
`schedule`(`schedule_id`)
        ON UPDATE CASCADE ON DELETE CASCADE
    ,FOREIGN       KEY        (`student_group_id`)       REFERENCES
`student_group`(`student_group_id`)
```

```
        ON UPDATE CASCADE ON DELETE RESTRICT
    ,UNIQUE      KEY      `uk_schedule_group`      (`schedule_id`,
`student_group_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Перший ключ – CASCADE – означає, що при видаленні запису заняття всі його групові прив'язки видаляються автоматично. Це правильна поведінка: schedule_groups не несе самостійного змісту без батьківського schedule. Другий ключ – RESTRICT – захищає групу: не можна видалити групу, якщо вона є учасником хоча б одного заняття. Унікальний ключ uk_schedule_group на парі (schedule_id, student_group_id) унеможливлює дублювання одного і того самого запису – СКБД фізично відхилить спробу додати групу до заняття вдруге [32].

Варто підсумувати підхід до обмежень цілісності, застосований у схемі в цілому. Переважна більшість зовнішніх ключів у системі налаштована на ON DELETE RESTRICT і ON UPDATE CASCADE [32]. Перше означає, що видалити батьківський запис неможливо, якщо на нього є посилання – це захищає архівні дані розкладу і навантаження від випадкового руйнування. Друге означає, що при зміні ідентифікатора батьківського запису (що в практиці рідко, але теоретично можливо через BIGINT UNSIGNED AUTO_INCREMENT) зміна каскадно поширюється на дочірні таблиці.

Виняток з правила RESTRICT зроблено лише у двох випадках. По-перше, user_account → person має CASCADE, оскільки обліковий запис не має сенсу без прив'язаної особи.

По-друге, schedule_groups → schedule має CASCADE, що є технічно необхідним для коректного очищення асоціативних записів при видаленні заняття. classroom_equipment_to_classroom також використовує CASCADE для обох ключів, оскільки запис про обладнання аудиторії не є самостійною сутністю.

Унікальні індекси застосовані на всіх довідникових полях, де дублювання є логічно неприпустимим: назви посад, дисциплін, підрозділів, логіни користувачів. На полях зовнішніх ключів транзакційних таблиць (schedule,

teacher_workload) визначено некластерні індекси – вони прискорюють операції об’єднання (JOIN) при формуванні сітки розкладу, де необхідно об'єднати до шести таблиць одним запитом [35].

Таким чином, реалізована фізична схема відповідає всім вимогам, сформульованим у другому розділі. Вона забезпечує цілісність довідникових даних, захист архівних записів від руйнування, запобігання дублюванню через унікальні ключі та оптимізацію читання через індекси. Підготовлений DDL-скрипт є самодостатнім і відтворюваним документом – запустивши його на будь-якому сервері MySQL з підтримкою InnoDB, розробник отримає повністю готову до використання схему даних без жодних додаткових маніпуляцій.

3.3 Налаштування об'єктно-реляційного відображення

Після того як фізичну схему бази даних реалізовано та верифіковано, постає наступне завдання: надати програмному коду зручний і типобезпечний інтерфейс для роботи з нею. Саме тут вступає в роль Entity Framework Core – об'єктно-реляційний маппер, який відповідає за трансляцію між реляційними таблицями MySQL і об'єктними класами C#. У проєкті це завдання вирішується у два кроки: спочатку налаштовується центральний компонент доступу – ScheduleDbContext, а потім для кожної сутності окремо описуються правила відображення через механізм Fluent API.

Центральним компонентом доступу до БД виступає ScheduleDbContext. Даний клас успадковує DbContext з EF Core і є єдиною точкою входу до бази даних з боку програмного коду. Він реєструє кожну таблицю схеми як властивість типу DbSet<T> де T – відповідний доменний клас. Наприклад, таблиця schedule стає DbSet<Schedule> Schedules, таблиця student_group – DbSet<StudentGroup> StudentGroups і так далі. Усього в контексті зареєстровано 44 властивостей DbSet, що відповідає кількості таблиць у фізичній схемі. Таке централізоване оголошення колекцій дозволяє розробнику абстрагуватися від

низькорівневих реляційних зв'язків і взаємодіяти з базою даних виключно через об'єктні моделі.

Клас надає два конструктори. Перший – безпараметричний – залишений для підтримки сценаріїв міграцій та внутрішніх інструментів EF Core. Другий приймає DbContextOptions<ScheduleDbContext> і є робочим: саме через нього DI-контейнер ASP.NET Core передає рядок підключення та налаштування СКБД. Метод OnConfiguring залишено порожнім із коментарем – конфігурація виноситься повністю в DalServiceExtensions, що є вірним підходом для веб-застосунків: строки підключення не повинні бути захардкоджені у класі контексту. Таке архітектурне розділення дозволяє гнучко підставляти різні конфігураційні рядки залежно від поточного середовища розгортання системи.

Найважливіший метод контексту – OnModelCreating. Основним його призначенням є ініціалізація та застосування індивідуальних конфігурацій для кожної окремої таблиці підсистеми розкладу. Замість нагромадження тисяч рядків коду в одному файлі, цей метод викликом єдиної інструкції автоматично реєструє всі правила з поточної збірки. Завдяки цьому архітектура стає максимально гнучкою: при розширенні системи розробнику достатньо лише додати новий файл конфігурації, не втручаючись у код самого контексту. Його підсумкова реалізація наведена у лістингу 3.5.

Лістинг 3.5 – Метод OnModelCreating

```
    protected override void OnModelCreating(ModelBuilder
modelBuilder)
    {
        modelBuilder
            .UseCollation("utf8mb4_0900_ai_ci")
            .HasCharSet("utf8mb4");


modelBuilder.ApplyConfigurationsFromAssembly(typeof(ScheduleDbCont
ext).Assembly);

    }
```

Перші два рядки встановлюють кодування і порівняння на рівні всієї схеми – це відповідає DEFAULT CHARSET=utf8 у DDL-скрипті. Ключовим є виклик ApplyConfigurationsFromAssembly: він автоматично знаходить у поточній збірці всі класи, що реалізують IEntityTypeConfiguration<T>, і застосовує їх до моделі. Завдяки цьому метод OnModelCreating залишається незмінним незалежно від кількості таблиць у схемі. Щоб додати конфігурацію для нової сутності, достатньо створити новий файл – контекст не потребує жодних правок. Це пряма реалізація принципу відкритості/закритості у шарі доступу до даних.

EF Core надає два способи описати правила відображення: анотації даних у вигляді атрибутів безпосередньо на класах сутностей (наприклад, [Column("time_slot_id")], [Required]) або інструментарій Fluent API через окремі конфігураційні класи. У проєкті свідомо обрано другий підхід.

Причина полягає в тому, що розміщення атрибутів у доменних класах спричиняє змішування різних обов'язків в одному місці. Клас TimeSlot повинен описувати концептуальну сутність «часовий слот», а не містити інформацію про системні назви колонок у MySQL чи специфічні типи даних рушія бази. Fluent API дозволяє залишати доменні класи чистими структурами даних, зосереджуючи всю інфраструктурну логіку відображення в ізольованих конфігураційних файлах. Будь-яка зміна у фізичній схемі бази даних – перейменування стовпця або зміна типу даних – повністю локалізується в одному місці й не торкається моделей предметної області.

Перш ніж розглядати самі конфігурації, варто звернути увагу на те, як виглядає доменний клас, на який відображається таблиця. Як приклад наведено клас Schedule у лістингу 3.6.

Лістинг 3.6 – Доменний клас сутності Schedule

```
public class Schedule
    : BaseEntity
{
    public ulong ScheduleId { get; set; }
    public DateOnly ClassDate { get; set; }
```

```
    public ulong TimeSlotId { get; set; }
    public ulong StudyPlanDetailsId { get; set; }
    public ulong EmployeeId { get; set; }
    public ulong ClassroomId { get; set; }

    public virtual StudyPlanDetail StudyPlanDetails { get; set; } =
null!;
    public virtual Employee Employee { get; set; } = null!;
    public virtual Classroom Classroom { get; set; } = null!;
    public virtual TimeSlot TimeSlot { get; set; } = null!;

    public virtual ICollection<ScheduleGroup> ScheduleGroups { get;
set; } = new List<ScheduleGroup>();

    public override ulong Id => ScheduleId;
}
```

Клас успадковує BaseEntity і перевизначає властивість Id, повертаючи ScheduleId – саме через цю абстракцію IGenericRepository<T> може працювати з будь-якою сутністю без знання її конкретного первинного ключа. Чотири скалярні властивості – TimeSlotId, StudyPlanDetailsId, EmployeeId, ClassroomId – є прямим відображенням зовнішніх ключів з DDL-скрипту. Чотири virtual-властивості навпроти них – TimeSlot, StudyPlanDetails, Employee, Classroom – є навігаційними: EF Core заповнює їх при завантаженні через Include. Колекція ScheduleGroups відображає відношення «один-до-багатьох» з асоціативною таблицею schedule_groups. Клас не містить жодного атрибута чи анотації – уся інфраструктурна логіка відображення зосереджена виключно у ScheduleConfiguration, що є прямим наслідком вибору Fluent API. Такий підхід гарантує строгу ізоляцію доменних моделей, оскільки вони залишаються повністю незалежними від специфіки обраної реляційної бази даних.

Кожен окремий конфігураційний клас реалізує узагальнений інтерфейс IEntityTypeConfiguration<T> і містить єдиний метод конфігурації Configure(EntityTypeBuilder<T> builder) [49].

Усі правила відображення описуються через декларативні ланцюжки викликів на об'єкті builder. Приклад оформлення класу конфігурації представлено у лістингу 3.7.

Лістинг 3.7 – Конфігурації сутності TimeSlot

```
public class TimeSlotConfiguration
    : IEntityTypeConfiguration<TimeSlot>
{
    public void Configure(EntityTypeBuilder<TimeSlot> builder)
    {
        builder.HasKey(e => e.TimeSlotId).HasName("PRIMARY");
        builder
            .ToTable("time_slot")
            .HasCharSet("utf8mb3")
            .UseCollation("utf8mb3_general_ci");
        builder.Property(e                                          =>
e.TimeSlotId).HasColumnName("time_slot_id");
        builder.Property(e => e.EndTime)
            .HasColumnType("time")
            .HasColumnName("end_time");
        builder.Property(e                                          =>
e.SlotNumber).HasColumnName("slot_number");
        builder.Property(e => e.StartTime)
            .HasColumnType("time")
            .HasColumnName("start_time");
        builder.Property(e => e.WhenApplied)
            .HasColumnType("date")
            .HasColumnName("when_applied");
        builder.Property(e => e.IsActive)
            .HasColumnType("bool")
            .HasColumnName("is_active");
    }
}
```

Розберімо ключові архітектурні деталі цієї конфігурації. Виклик HasName("PRIMARY") явно задає ім'я обмеження первинного ключа, що гарантує суворе збігання з системним іменуванням у MySQL і запобігає автоматичній генерації внутрішньої назви з боку ORM. Метод ToTable("time_slot") відображає C#-клас TimeSlot (PascalCase) на фізичну таблицю time_slot (snake_case), завдяки чому система не намагатиметься шукати стандартно плюралізовану таблицю TimeSlots.

Особливої уваги заслуговує жорстке визначення типів даних для полів, що відповідають за хронологію. Виклики HasColumnType("time") для початку та кінця пари, а також HasColumnType("date") для дати застосування розкладу є

критично важливими. Без цих прямих вказівок Entity Framework Core міг би застосувати менш ефективні або надлишкові стандартні типи (наприклад, datetime(6)), що призвело б до розбіжностей із попередньо спроєктованою БД. Аналогічно конфігурується поле статусу через HasColumnType("bool").

Таке детальне налаштування кожної властивості забезпечує стовідсоткове і точне відтворення фізичної схеми сховища на рівні інфраструктурного коду.

В свою чергу конфігурація такої сутності як підрозділ є складнішою – вона відображає self-referencing зв'язок, який у реляційній моделі реалізується через parent_department_id. Реалізація цього наведена у лістингу 3.8.

Лістинг 3.8 – Фрагмент конфігурації сутності Department

```
builder.HasOne(d => d.ParentDepartment)
    .WithMany(p => p.ChildDepartments)
    .HasForeignKey(d => d.ParentDepartmentId)
    .HasConstraintName("department_ibfk_2");
```

У доменному класі Department навігаційна властивість ParentDepartment вказує на батьківський підрозділ, а колекція ChildDepartment – на всі дочірні. Такий двосторонній зв'язок дозволяє обходити ієрархію в обох напрямках: від факультету вниз до лабораторій і від лабораторії вгору до факультету. Додатково тут описано два унікальні індекси на полях FullName і ShortName через HasIndex(...).IsUnique(), що точно відповідає UNIQUE(full_name) і UNIQUE(short_name) у DDL-скрипті.

Конфігурація StudentGroupConfiguration використовує ту саму self-referencing техніку для ієрархії потік → група → підгрупа. Приклад представлено у лістингу 3.9.

Лістинг 3.9 – Фрагмент конфігурації сутності StudentGroup

```
builder.HasOne(d => d.ParentGroup)
    .WithMany(p => p.ChildGroups)
    .HasForeignKey(d => d.ParentGroupId)
    .OnDelete(DeleteBehavior.Restrict)
    .HasConstraintName("student_group_ibfk_3");
```

Принципова схожість конфігурацій department і student_group невипадкова: обидві таблиці мають ідентичну архітектурну концепцію ієрархії, яку DDL-скрипт реалізує однотипно.

Конфігурація таблиці розкладу є найбільш критичною з точки зору цілісності, тому її варто розглянути детальніше. Ключова особливість – чотири некластерних індекси на всіх полях зовнішніх ключів. Демонастрацію цих ключів можна побачити у лістингу 3.10.

Лістинг 3.10 – Фрагмент конфігурації сутності Schedule з визначенням індексів

```
builder.HasIndex(e => e.TimeSlotId, "time_slot_id");
builder.HasIndex(e => e.StudyPlanDetailsId,
"study_plan_details_id");
builder.HasIndex(e => e.EmployeeId, "employee_id");
builder.HasIndex(e => e.ClassroomId, "classroom_id");
```

Ці індекси безпосередньо відповідають оголошеним у DDL-скрипті і є необхідними для прийнятної швидкодії при формуванні сітки розкладу – запит на отримання розкладу для факультету об'єднує таблиці schedule, time_slot, study_plan_details, discipline, employee, person, classroom і schedule_groups одночасно.

Усі чотири зовнішні ключі налаштовані відповідно до DDL скрипта, з застосуванням заборони на каскадне видалення записів, це представлено на у лістигу 3.11.

Лістинг 3.11 – Фрагмент конфігурації сутності Schedule з конфігурацією зовнішнього ключа до сутності TimeSlot

```
builder.HasOne(d => d.TimeSlot)
    .WithMany(p => p.Schedules)
    .HasForeignKey(d => d.TimeSlotId)
    .OnDelete(DeleteBehavior.Restrict)
    .HasConstraintName("schedule_ibfk_1");
```

DeleteBehavior.Restrict тут – не просто відображення DDL, а програмна гарантія цілісності: EF Core підніме виняток на рівні застосунку ще до того, як запит дійде до СКБД, якщо операція видалення порушить посилальну цілісність.

Конфігурація schedule_groups є найцікавішою з архітектурної точки зору, бо відображає асиметрію стратегій видалення, описану в лістингу 3.12.

Лістинг 3.12 – Фрагмент конфігурації сутності ScheduleGroup з різними правилами щодо видалення

```
builder.HasIndex(e => new { e.ScheduleId, e.StudentGroupId },
"uk_schedule_group")
    .IsUnique();

builder.HasOne(d => d.Schedule)
    .WithMany(p => p.ScheduleGroups)
    .HasForeignKey(d => d.ScheduleId)
    .OnDelete(DeleteBehavior.Cascade)
    .HasConstraintName("schedule_group_ibfk_1");

builder.HasOne(d => d.StudentGroup)
    .WithMany(p => p.ScheduleGroups)
    .HasForeignKey(d => d.StudentGroupId)
    .OnDelete(DeleteBehavior.Restrict)
    .HasConstraintName("schedule_group_ibfk_2");
```

Композитний унікальний індекс HasIndex(e => new { e.ScheduleId, e.StudentGroupId }).IsUnique() відображає UNIQUE KEY uk_schedule_group(schedule_id, student_group_id) зі скрипту. Це не просто перевірка на рівні програмного коду – СКБД фізично відхилить спробу вставити дублікат, і EF Core коректно транслює цю поведінку у виняток DbUpdateException.

Щоб ScheduleDbContext коректно функціонував у веб-застосунку, його необхідно зареєструвати в контейнері впровадження залежностей. Це виконується через статичний метод розширення DalServiceExtensions.AddDalServices, як наведено у лістингу 3.13.

Лістинг 3.13 – Повний код реєстрації ScheduleDbContext у контейнері Dependency injection

```
public static IServiceCollection AddDalServices(this
IServiceCollection services, string connectionString)
{
    services.AddDbContext<ScheduleDbContext>(options =>
        options.UseMySql(connectionString,
ServerVersion.AutoDetect(connectionString)));

    services.AddScoped<IUnitOfWork, UnitOfWork>();

    return services;
}
```

Метод UseMySql з ServerVersion.AutoDetect автоматично визначає версію MySQL сервера при першому підключенні і налаштовує відповідні можливості провайдера. Це знімає проблему ручного зазначення версії сервера, яка може відрізнятися між середовищами розробки, тестування та продакшену. AddScoped<IUnitOfWork, UnitOfWork> реєструє UnitOfWork зі Scoped-lifetime – це принциповий момент: в межах одного HTTP-запиту існує рівно один екземпляр ScheduleDbContext, і всі репозиторії, які UnitOfWork агрегує, поділяють його [48]. Саме це гарантує, що всі зміни, накопичені в різних репозиторіях протягом одного запиту, фіксуються єдиною транзакцією при виклику CommitAsync. Якщо ж на буль-якому етапі обробки запиту виникає помилка то транзакція суцільно відхиляється. Це надійно захищає систему від збереження неповних або некоректних фрагментів розкладу. Таким чином, налаштування об'єктно-реляційного відображення в проєкті базується на трьох взаємопов'язаних рішеннях.

Перше – розділена конфігурація через IEntityTypeConfiguration<T> для кожної сутності, що забезпечує чистоту доменних класів і локалізацію змін.

Друге – автоматична реєстрація конфігурацій через ApplyConfigurationsFromAssembly, що прибирає ручне управління списком відображень і підтримує принцип відкритості/закритості.

Третє – Scoped-lifetime для DbContext через DalServiceExtensions, що гарантує транзакційну ізоляцію на рівні HTTP-запиту. Разом ці рішення забезпечують точну, підтримувану та масштабовану відповідність між фізичною схемою MySQL і об'єктною моделлю C#.

3.4 Створення шару доступу до даних

Після налаштування об'єктно-реляційного відображення наступним кроком є побудова програмного інтерфейсу, через який решта системи взаємодіє з базою даних. У проєкті цей інтерфейс реалізований у вигляді двох взаємодоповнюючих абстракцій: репозиторіїв, що інкапсулюють логіку запитів до конкретних таблиць, та UnitOfWork, що координує їх роботу в межах єдиної транзакції. Разом вони утворюють шар доступу до даних, що повністю ізолює ScheduleDbContext від будь-якого коду поза межами інфраструктурного проєкту UniSS.Dal.

Система репозиторіїв побудована на двох рівнях [47]. Перший – узагальнений репозиторій Repository<T>, що реалізує стандартні CRUD-операції для будь-якої сутності. Другий – спеціалізовані репозиторії, що успадковують перший і розширюють його предметно-орієнтованими методами вибірки. Таке розділення є осмисленим архітектурним рішенням: базові операції реалізуються один раз і більше не дублюються, а специфічна логіка доступу зосереджена там, де вона семантично належить.

Клас Repository<T> є фундаментом усього шару доступу. Він реалізує інтерфейс IGenericRepository<T> і накладає обмеження where T : class, IEntity – тобто будь-яка сутність у системі повинна реалізовувати IEntity, що гарантує наявність властивості Id типу ulong. Це відповідає BIGINT UNSIGNED AUTO_INCREMENT у DDL-скрипті: між фізичною схемою та типовою системою C# існує точна відповідність.Конструктор отримує DbContext і одразу ініціалізує захищене поле _dbSet через виклик _dbContext.Set<T>().

Саме через _dbSet виконуються всі запити – це стандартний підхід EF Core, що дозволяє уникнути прямих посилань на DbContext у методах класу. Конструктор та метод GetAllAsync проілюстровано у лістингу 3.14.

Лістинг 3.14 – Фрагмент коду конструктора загального репозиторію та методу отримання усіх записів

```
protected readonly DbContext _dbContext;

protected readonly DbSet<T> _dbSet;

public Repository(DbContext dbContext)
{
    _dbContext = dbContext ?? throw new
ArgumentNullException(nameof(dbContext));
    _dbSet = _dbContext.Set<T>();
}

public async Task<IEnumerable<T>> GetAllAsync(
Func<IQueryable<T>, IIncludableQueryable<T, object>> include = null)
{
    IQueryable<T> query = _dbSet.AsNoTracking();
    if (include != null)
    {
        query = include(query);
    }
    return await query.ToListAsync();
}
```

Метод GetAllAsync є найгнучкішим у класі – він приймає необов'язковий параметр include типу Func<IQueryable<T>, IIncludableQueryable<T, object>>.

Виклик AsNoTracking() означає, що EF Core не реєструє завантажені об'єкти у ChangeTracker – для операцій лише на читання це дає помітний приріст продуктивності [48]. Параметр include дозволяє викликачу передати ланцюжок Include/ThenInclude прямо у виклику, не змінюючи базовий клас.

Метод GetByIdAsync використовує FindAsync замість FirstOrDefaultAsync, а InsertAsync реєструє нову сутність у ChangeTracker без негайного збереження, що продемонстровано у лістингу 3.15.

Лістинг 3.15 – Метод отримання за id та додавання запису в Repository<T>

```
public async Task<T> GetByIdAsync(ulong id)
{
    return await _dbSet.FindAsync(id);
}
public async Task<T> InsertAsync(T entity)
{
    if (entity == null)
        throw new ArgumentNullException(nameof(entity));
    var result = await _dbContext.AddAsync(entity);
    return result.Entity;
}
```

Різниця між FindAsync і FirstOrDefaultAsync суттєва: FindAsync спочатку шукає сутність у ChangeTracker – якщо вона вже завантажена в поточному контексті, звернення до бази даних не відбувається. InsertAsync і UpdateAsync також не викликають SaveChangesAsync – вони реєструють зміни у ChangeTracker до виклику UnitOfWork.CommitAsync. DeleteAsync завантажує сутність через GetByIdAsync і передає до _dbSet.Remove, залишаючи обробку відсутності запису верхньому рівню.

ScheduleRepository є найважливішим за семантикою. Кожен з п'яти методів фільтрує _dbSet за відповідним зовнішнім ключем. Два з них (GetByClassDateAsync і GetByClassroomAsync) наведено у лістингу 3.16.

Лістинг 3.16 – Фрагмент реалізації специфікованих методів у репозиторії сутності Schedule

```
    public async Task<IEnumerable<Schedule>>
GetByClassDateAsync(DateOnly date)
    {
        return await _dbSet.Where(s => s.ClassDate ==
date).ToListAsync();
    }
    public async Task<IEnumerable<Schedule>>
GetByClassroomAsync(ulong classroomId)
    {
        return await _dbSet.Where(s => s.ClassroomId ==
classroomId).ToListAsync();
    }
```

Решта методів – GetByTimeSlotAsync, GetByStudyPlanDetailsAsync, GetByEmployeeAsync – реалізовані аналогічно з відповідним полем фільтрації. Варто звернути увагу, що всі методи звертаються до полів зовнішніх ключів безпосередньо, а не через навігаційні властивості. Це свідоме рішення: для перевірки конфліктів у системі валідації потрібна лише наявність записів з певними ідентифікаторами – завантажувати пов'язані сутності через Include тут зайво і погіршує продуктивність. Саме ці методи використовуються у правилах ClassroomConflictRule і EmployeeScheduleConflictRule при перевірці подвійного бронювання перед збереженням нового заняття.

StudentGroupRepository демонструє, як ієрархічна природа сутності відображається у методах доступу. Найцікавіші з них – GetRootGroupsAsync (метод для пошуку кореневого запису) і GetByParentGroupAsync (метод для пошуку батьківської групи). Їх продемонстровано у лістингу. 3.17.

Лістинг 3.17 – Фрагмент реалізації специфікованих методів у репозиторії сутності StudentGroupRepository

```
public async Task<IEnumerable<StudentGroup>>
GetByParentGroupAsync(ulong parentGroupId)
    {
        return await _dbSet.Where(sg => sg.ParentGroupId ==
parentGroupId).ToListAsync();
    }
    public async Task<IEnumerable<StudentGroup>>
GetRootGroupsAsync()
    {
        return await _dbSet.Where(sg => sg.ParentGroupId ==
null).ToListAsync();
    }
```

GetRootGroupsAsync перекладає на рівень SQL умову WHERE parent_student_group_id IS NULL – саме так вибираються потоки з ієрархічної таблиці, описаної у підрозділі 3.2. GetByParentGroupAsync повертає безпосередніх нащадків – наприклад, всі групи потоку або всі підгрупи академічної групи. Разом ці два методи дозволяють обходити ієрархію рівень за

рівнем без рекурсивних запитів. Решта методів – SearchByNameAsync, GetByEducationProgramAsync, GetByEnrollmentYearAsync – реалізовані за тим самим шаблоном простої фільтрації за відповідним полем.

Важливо розуміти, як саме реалізовані репозиторії стають доступними для вищих рівнів системи. Жоден зовнішній компонент не посилається безпосередньо на конкретний клас – наприклад, ScheduleRepository. Натомість взаємодія відбувається виключно через інтерфейси, оголошені у UniSS-DAL-Domain. Як приклад наведено інтерфейс IStudentGroupRepository у лістингу 3. 18.

Лістинг 3.18 – Інтерфейс що описує StudentGroupRepository

```
public interface IStudentGroupRepository : 
IGenericRepository<StudentGroup>
{
    Task<IEnumerable<StudentGroup>> SearchByNameAsync(string name);
    Task<IEnumerable<StudentGroup>> 
GetByStudentGroupTypeAsync(ulong studentGroupTypeId);
    Task<IEnumerable<StudentGroup>> 
GetByEducationProgramAsync(ulong educationProgramId);
    Task<IEnumerable<StudentGroup>>    GetByParentGroupAsync(ulong 
parentGroupId);
    Task<IEnumerable<StudentGroup>> GetRootGroupsAsync();
    Task<IEnumerable<StudentGroup>>  GetByEnrollmentYearAsync(short 
enrollmentYear);
}
```

Інтерфейс успадковує IGenericRepository<StudentGroup> і тим самим автоматично включає базові CRUD-методи, додаючи до них предметно-специфічні. Саме цей контракт, а не клас StudentGroupRepository, фігурує як тип властивості в IUnitOfWork – IStudentGroupRepository StudentGroupRepository { get; }. Завдяки цьому будь-який компонент, що отримує IUnitOfWork через DI, бачить лише інтерфейс і не має жодної залежності від EF Core чи MySQL. Підміна реальної реалізації тестовою заглушкою зводиться до одного рядка у конфігурації DI-контейнера.

UnitOfWork є центральним координатором шару доступу до даних, що об'єднує всі репозиторії під спільним управлінням транзакцій [4]. Клас реалізує інтерфейс IUnitOfWork і отримує ScheduleDbContext через конструктор. Кожен зі спеціалізованих репозиторіїв оголошений як приватне поле і ініціалізується відкладено через оператор «??=». У лістингу 3.19 наведено приклад такої ініціалізації для ScheduleRepository та ScheduleGroupRepository.

Лістинг 3.19 – Фрагменти класу UnitOfWork

```
private IScheduleRepository _scheduleRepository;
private IScheduleGroupRepository _scheduleGroupRepository;
private IStudentGroupRepository _studentGroupRepository;
// … решта полів аналогічно
public IScheduleRepository ScheduleRepository =>
    this._scheduleRepository                ??=                new
ScheduleRepository(_dbContext);
public IScheduleGroupRepository ScheduleGroupRepository =>
    this._scheduleGroupRepository           ??=                new
ScheduleGroupRepository(_dbContext);
public IStudentGroupTypeRepository StudentGroupTypeRepository =>
    this._studentGroupTypeRepository        ??=                new
StudentGroupTypeRepository(_dbContext);
// … решта полів аналогічно
```

Такий підхід означає, що репозиторій створюється лише при першому зверненні до нього в межах поточного HTTP-запиту. В системі з 44 репозиторіями це суттєво: якщо конкретний запит потребує лише операцій з розкладом, решта репозиторіїв не будуть ніколи інстанційовані. Принципово важливим є те, що всі вони поділяють один і той самий _dbContext – саме це гарантує коректну роботу транзакцій. Така архітектурна конфігурація усуває ризики часткових оновлень, коли зміни в одній таблиці фіксуються, а в іншій – губляться через використання ізольованих екземплярів підключень під час каскадних запитів. Окрім іменованих репозиторіїв, UnitOfWork надає узагальнений доступ через метод Repository(), наведений у лістингу 3.20.

Лістинг 3.20 – Реалізація узагальненого методу доступу до репозиторію Repository<T>() у класі UnitOfWork

```
public IGenericRepository<T> Repository<T>()
    where T : class, IEntity
{
    this._repositories ??= new Hashtable();

    var type = typeof(T).Name;

    if (!this._repositories.ContainsKey(type))
    {
        var repositoryInstance = new
Repository<T>(this._dbContext);
        this._repositories.Add(type, repositoryInstance);
    }

    return (IGenericRepository<T>)this._repositories[type];
}
```

Метод використовує Hashtable як кеш екземплярів: при першому запиті до типу T створюється Repository<T> і зберігається; при наступних зверненнях повертається вже наявний. Це дозволяє звертатися до будь-якої таблиці без оголошення окремого репозиторію – достатньо використати конструкцію _unitOfWork.Repository<TimeSlot>().

Два методи UnitOfWork безпосередньо відповідають за управління транзакцією. Їх реалізацію наведено у лістингу 3.21.

Лістинг 3.21 – Реалізація методів фіксації та відкату змін у класі UnitOfWork

```
public async Task<int> CommitAsync(CancellationToken
cancellationToken = default)
{
    return await
this._dbContext.SaveChangesAsync(cancellationToken);
}

public Task RollbackAsync()
{
    foreach (var entry in this._dbContext.ChangeTracker.Entries())
    {
        entry.State = EntityState.Unchanged;
    }
```

```
    return Task.CompletedTask;
}
```

CommitAsync делегує збереження до _dbContext.SaveChangesAsync. EF Core накопичує всі зміни, виконані через репозиторії, у ChangeTracker, і лише цей виклик транслює їх в єдину транзакцію бази даних [33]. Це означає, що навіть якщо протягом одного запиту були викликані InsertAsync для нового запису Schedule і кілька InsertAsync для відповідних записів ScheduleGroup – жодна з цих операцій не зафіксується окремо: або всі разом, або жодна.

RollbackAsync реалізований на рівні ChangeTracker без жодного SQL-запиту – він переводить стан усіх відслідковуваних записів у EntityState.Unchanged, фактично скасовуючи всі накопичені зміни. Такий підхід ефективніший за явний BeginTransaction/Rollback на рівні СКБД, оскільки не потребує підтримки відкритої транзакції на сервері протягом усього часу обробки запиту.

UnitOfWork також реалізує IDisposable – при завершенні HTTP-запиту DI-контейнер ASP.NET Core автоматично викликає Dispose, що закриває з'єднання з базою даних і звільняє пов'язані ресурси. Розробник не зобов'язаний керувати цим вручну.

Таким чином, реалізований шар доступу до даних побудований на чіткому розділенні між контрактами і реалізацією. Repository<T> покриває стандартні операції один раз і для всіх сутностей, спеціалізовані репозиторії розширюють його предметно-специфічними запитами без жодного дублювання, а UnitOfWork з відкладеною ініціалізацією та методами CommitAsync і RollbackAsync гарантує транзакційну атомарність будь-якої комплексної операції над базою даних. Ізоляція контрактів у проєкті UniSS-DAL-Domain від реалізацій у UniSS.Dal забезпечує незалежність вищих рівнів системи від конкретної СКБД та інфраструктурних деталей EF Core.

Для підтвердження коректності реалізованих доменних моделей у проєкті створено окремий тестовий модуль UniSS-DAL-Test на базі фреймворку NUnit.

Тестове покриття охоплює всі 44 сутності предметної області – для кожної з них реалізовано окремий тест-клас, що успадковує базовий абстрактний клас EntityTest. Такий підхід верифікує коректність об'єктних моделей та їхніх властивостей незалежно від підключення до реальної бази даних, що є прямим наслідком архітектурного рішення щодо ізоляції доменних класів від інфраструктурних залежностей.

Висновок до третього розділу

У третьому розділі обрано технологічний стек та реалізовано повноцінну підсистему збереження та доступу до даних відповідно до архітектурних рішень другого розділу.

Платформа .NET 8 з мовою C# забезпечує кросплатформене розгортання та строгу статичну типізацію, що мінімізує ризики помилок при об'єктно-реляційному мапінгу. СКБД MySQL з рушієм InnoDB обрано як оптимальне рішення для систем з щільним графом перехресних посилань, що потребують повної підтримки моделі ACID та блокування на рівні рядків. Entity Framework Core з методологією Database-First забезпечує тотальний контроль архітектора над фізичною схемою при одночасній автоматизації генерації об'єктних моделей.

Фізична схема бази даних охоплює 44 таблиці, розподілені між шістьма тематичними блоками, реалізовані у вигляді DDL-скриптів MySQL з рушієм InnoDB. Диференційований підхід до стратегій зовнішніх ключів – заборона видалення для довідникових та транзакційних таблиць і каскадне видалення для асоціативних сутностей забезпечує захист архівних даних при збереженні коректності очищення залежних записів [32]. Унікальні індекси на ключових полях утворюють бар'єр проти дублювання даних на рівні СКБД.

Об'єктно-реляційне відображення реалізовано через окремі класи IEntityTypeConfiguration<T> з автоматичною реєстрацією через ApplyConfigurationsFromAssembly, що тримає доменні класи чистими від інфраструктурних деталей і локалізує зміни схеми в одному файлі.

Реєстрація ScheduleDbContext зі Scoped-lifetime гарантує єдиний екземпляр контексту в межах HTTP-запиту.

Узагальнений Repository<T> з підтримкою AsNoTracking() покриває стандартні CRUD-операції для всіх сутностей, а спеціалізовані репозиторії розширюють його предметно-специфічними методами вибірки, включно з обходом ієрархічних структур. Взаємодія вищих рівнів відбувається виключно через інтерфейси проєкту UniSS-DAL-Domain – IUnitOfWork є єдиним контрактом доступу до репозиторіїв, що реєструється у DI-контейнері. UnitOfWork з відкладеною ініціалізацією репозиторіїв через ??= та методами CommitAsync і RollbackAsync гарантує транзакційну атомарність комплексних операцій.

ВИСНОВКИ

У кваліфікаційній роботі вирішено задачу проєктування та розробки реляційної бази даних і шару доступу до даних для системи керування розкладом занять університету.

У першому розділі проведено системний аналіз предметної області та формалізовано вимоги до сховища даних. Порівняльний огляд існуючих систем виявив ключові архітектурні прогалини у підходах до організації DAL. Досліджено теоретичні засади реляційної моделі, нормалізації відношень, транзакційної моделі ACID, методологій Code-First та DB-First, а також патернів організації шару доступу до даних: Data Mapper, Repository, Unit of Work та DTO.

У другому розділі сформовано технічне завдання та визначено повний комплекс функціональних і нефункціональних вимог до підсистеми. Побудовано концептуальну модель реляційної бази даних та спроєктовано архітектуру шару доступу до даних на основі багатошарового підходу з патернами Generic Repository та Unit of Work.

У третьому розділі обґрунтовано вибір технологічного стеку: платформа .NET 8 з мовою C#, СКБД MySQL з рушієм InnoDB та Entity Framework Core з методологією Database-First. Реалізовано фізичну схему бази даних у вигляді DDL-скриптів, що охоплює 44 таблиці з диференційованими стратегіями зовнішніх ключів та унікальними індексами. Налаштовано об'єктно-реляційне відображення через окремі класи конфігурацій Fluent API з автоматичною реєстрацією. Реалізовано узагальнений репозиторій, спеціалізовані репозиторії та UnitOfWork з транзакційною атомарністю. Досвід інтеграції даних патернів пройшов наукову апробацію та висвітлений у відповідній публікації [50].

Результатом роботи є готова до інтеграції підсистема управління даними, що утворює надійний інфраструктурний фундамент для системи керування університетським розкладом і повністю відповідає визначеним вимогам щодо цілісності, транзакційної безпеки та масштабованості.

СПИСОК ВИКОРИСТАНИХ ПОСИЛАНЬ

1. Antonov Yu.S., Kalko D.R., Maruniak A.O. Development of a generalized data model and data access architecture for academic scheduling systems using .NET and Blazor. Scientific practice: modern and classical research methods : collection of scientific papers with proceedings of the IX International Scientific and Practical Conference (Boston, April 10, 2026). Boston-Vinnytsia : Primedia eLaunch & UKRLOGOS Group LLC, 2026. P. 114–117.
2. Антонов Ю. Реалізація підсистеми керування розкладом університету на основі змішаної архітектури. Вісник Хмельницького національного університету. Серія: Технічні науки. 2025. Том 357, № 5.1. С. 13–22. Режим доступу: https://doi.org/10.31891/2307-5732-2025-357-1
3. Bashab A., Ibrahim A.O., Hashem I.A.T., Aggarwal K., Mukhlif F., Ghaleb F.A., Abdelmaboud A. Optimization Techniques in University Timetabling Problem: Constraints, Methodologies, Benchmarks, and Open Issues. Computers, Materials & Continua. 2023. Vol. 74, No. 3. P. 6461–6484. Режим доступу: https://doi.org/10.32604/cmc.2023.034051
4. Prajapati M. ASP.NET MVC - Generic Repository Pattern and Unit of Work. International Journal of All Research Writings. 2019. Vol. 1, Issue 1. P. 23–25. Режим доступу: https://www.ijarw.com/PublishedPaper/IJARW1005.pdf
5. Elmasri R., Navathe S. B. Fundamentals of Database Systems. 7th ed. Pearson Education, 2016. 1268 p.
6. Priyanshu. Design Tree-Based Hierarchy Structure in Database Using Self-Referencing Foreign Key [Електронний ресурс]. Medium. 2024. Режим доступу: https://medium.com/@priyanshupardhi/design-tree-based-hierarchy-structure-in-database-using-self-referencing-foreign-key-9a0b84a4a801 (дата звернення: 20.04.2026).
7. UniTime – University Scheduling System. Офіційна документація та репозиторій у вільному доступі [Електронний ресурс]. Режим доступу: https://github.com/UniTime/unitime (дата звернення: 20.04.2026).

8. RosarioSIS – Student Information System. Офіційна документація та репозиторій у вільному доступі [Електронний ресурс]. Режим доступу: https://gitlab.com/rosariosis/rosariosis (дата звернення: 20.04.2026).
9. Moodle Developer Resources. Офіційна документація та відкрита для перегляду схема бази даних [Електронний ресурс]. Режим доступу: https://moodledev.io/docs/apis/core/dml (дата звернення: 29.05.2026).
10. Connolly T. M., Begg C. E. Database Systems: A Practical Approach to Design, Implementation and Management. Pearson Education, Limited, 2011. 1400 p.
11. Войтко Б. С., Мічківський С. М. Система формування розкладу навчальних занять з використанням платформи 1С: Підприємство. Збірник тез доповіді Прикладні інформаційні технології. Вінниця: Донецький національний університет імені Василя Стуса, 2020. С. 194–195.
12. Римар П.В., Войтко Б.С. Розробка автоматизованої системи формування розкладу навчальних занять з використанням 1С: «Підприємство». Вісник Хмельницького національного університету. 2021. №5.
13. Лещенко В. О., Зелінська В. О. Інформаційна система для підтримки діяльності структурного підрозділу закладу вищої освіти. Вінниця : Донецький національний університет імені Василя Стуса, 2024.
14. Комп'ютерна програма: «Schedule Module For Antonov Students Test System» (Авторське свідоцтво) Авт. свід. №103272 від 18.03.2021, видане Українським інститутом інтелектуальної власності; Заявл. №с202100796 від 15.02.2021.
15. A.S.T.S. Schedule module [Електронний ресурс]. Режим доступу: /http://antonov.donnu.org/ (дата звернення: 15.02.2026).
16. Brewer E. A. Towards robust distributed systems. The nineteenth annual ACM symposium on Principles of distributed computing, Portland, Oregon, USA, 16–19 July 2000. New York : ACM, 2000. Режим доступу: https://www.researchgate.net/publication/221343719_Towards_robust_distributed_systems

17. Теорема CAP [Електронний ресурс]. Режим доступу: https://uk.wikipedia.org/wiki/Теорема_CAP (дата звернення: 30.04.2026).
18. Pritchett D. BASE: An Acid Alternative. Queue. 2008. Vol. 6, No. 3. P. 48–55. Режим доступу: https://doi.org/10.1145/1394127.1394128
19. IBM Corporation. IBM Informix 12.10 Documentation [Електронний ресурс]. Режим доступу: https://www.ibm.com/docs/en/informix-servers/12.10.0 (дата звернення: 30.04.2026).
20. Microsoft Learn. What Is SQL Server? SQL Server Technical Documentation [Електронний ресурс]. Режим доступу: https://learn.microsoft.com/en-us/sql/sql-server/what-is-sql-server (дата звернення: 01.05.2026).
21. PostgreSQL Global Development Group. PostgreSQL: About [Електронний ресурс]. Режим доступу: https://www.postgresql.org/about/ (дата звернення: 01.05.2026).
22. MySQL Documentation. What is MySQL? MySQL 8.0 Reference Manual [Електронний ресурс]. Режим доступу: https://dev.mysql.com/doc/refman/8.0/en/what-is-mysql.html (дата звернення: 01.05.2026).
23. Codd E. F. A relational model of data for large shared data banks. Communications of the ACM. 1983. Vol. 26, No. 1. P. 64–69. Режим доступу: https://doi.org/10.1145/357980.358007 (дата звернення: 02.05.2026).
24. MySQL Documentation. InnoDB and the ACID Model. MySQL 8.0 Reference Manual [Електронний ресурс]. Режим доступу: https://dev.mysql.com/doc/refman/8.0/en/mysql-acid.html (дата звернення: 02.05.2026).
25. Date C. J. Database in Depth: Relational Theory for Practitioners. Beijing : O'Reilly, 2005. 208 p.
26. Lerman J. Programming Entity Framework: Building Data Centric Apps with the Ado.Net Entity Framework. O'Reilly Media, Incorporated, 2010.
27. Programming entity framework: Code first / ed. by M. Rowan. Sebastopol, CA : O'Reilly Media, 2011.

28. Fowler M. Patterns of Enterprise Application Architecture. Addison-Wesley Professional, 2002. 560 p.
29. Microsoft Learn. Implementing the Repository and Unit of Work Patterns in an ASP.NET MVC Application [Електронний ресурс]. Режим доступу: https://learn.microsoft.com/en-us/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application (дата звернення: 05.05.2026).
30. Sommerville I. Software Engineering. 10th ed. Harlow : Pearson Education Limited, 2016. 816 p.
31. Microsoft Learn. Common web application architectures. Azure Architecture Center [Електронний ресурс]. Режим доступу: https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures (дата звернення: 06.05.2026).
32. MySQL Documentation. FOREIGN KEY Constraints. MySQL 8.4 Reference Manual [Електронний ресурс]. Режим доступу: https://dev.mysql.com/doc/refman/8.4/en/create-table-foreign-keys.html (дата звернення: 06.05.2026).
33. Microsoft Learn. Using Transactions. Entity Framework Core Documentation [Електронний ресурс]. Режим доступу: https://learn.microsoft.com/en-us/ef/core/saving/transactions (дата звернення: 07.05.2026).
34. Mihalcea V. A Beginner's Guide to Database Locking and the Lost Update Phenomena [Електронний ресурс]. Режим доступу: https://vladmihalcea.com/a-beginners-guide-to-database-locking-and-the-lost-update-phenomena/ (дата звернення: 07.05.2026).
35. MySQL Documentation. Using Foreign Keys. MySQL 8.0 Reference Manual [Електронний ресурс]. Режим доступу: https://dev.mysql.com/doc/refman/8.0/en/example-foreign-keys.html (дата звернення: 10.05.2026).
36. Wieringa R. J. Design Science Methodology for Information Systems and Software Engineering. London : Springer, 2014. 332 p.

37. Khrienov A. Association Patterns in Database Design: One-to-Many, Many-to-Many, and Beyond [Електронний ресурс]. Medium. 2025. Режим доступу: https://medium.com/@artemkhrenov/association-patterns-in-database-design-one-to-many-many-to-many-and-beyond-06aaa1b8ddd6 (дата звернення: 11.05.2026).
38. DataCamp. Many to Many Relationships: A Guide to Database Design [Електронний ресурс]. 2026. Режим доступу: https://www.datacamp.com/blog/many-to-many-relationship (дата звернення: 11.05.2026).
39. Microsoft Learn. Architectural Principles. Modern Web Applications with ASP.NET Core [Електронний ресурс]. Режим доступу: https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/architectural-principles (дата звернення: 05.05.2026).
40. Microsoft Learn. Common Web Application Architectures [Електронний ресурс]. Режим доступу: https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures (дата звернення: 05.05.2026).
41. Osidach V. Understanding GenericRepository in C# and .NET 8 [Електронний ресурс]. Medium. 2024. Режим доступу: https://medium.com/@valentin.osidach/understanding-genericrepository-in-c-and-net-8-b44db3b0b7db (дата звернення: 11.05.2026).
42. Microsoft Learn. DbContext Lifetime, Configuration, and Initialization. EF Core [Електронний ресурс]. Режим доступу: https://learn.microsoft.com/en-us/ef/core/dbcontext-configuration/ (дата звернення: 06.05.2026).
43. Joshi B. Reverse Engineering EF Core Model and Data Validation Techniques [Електронний ресурс]. Режим доступу: https://www.binaryintellect.net/articles/4ad8c945-168a-4d1f-9284-c637cc1a7fe4.aspx (дата звернення: 14.05.2026).
44. Microsoft Learn. When to Choose .NET 8 for Docker Containers [Електронний ресурс]. Режим доступу: https://learn.microsoft.com/en-

us/dotnet/architecture/microservices/net-core-net-framework-containers/net-core-container-scenarios (дата звернення: 08.05.2026).

45. Epözdemir J. Exploring Type Safety in C# [Електронний ресурс]. Dev Genius. 2024. Режим доступу: https://blog.devgenius.io/exploring-type-safety-in-c-291e4313fe3d (дата звернення: 25.04.2026).
46. MySQL Documentation. InnoDB Locking. MySQL 8.4 Reference Manual [Електронний ресурс]. Режим доступу: https://dev.mysql.com/doc/refman/8.4/en/innodb-locking.html (дата звернення: 08.05.2026).
47. MySQL Documentation. Internal Locking Methods. MySQL 8.0 Reference Manual [Електронний ресурс]. Режим доступу: https://dev.mysql.com/doc/refman/8.0/en/internal-locking.html (дата звернення: 08.05.2026).
48. Microsoft Learn. Security Considerations (Entity Framework) [Електронний ресурс]. Режим доступу: https://learn.microsoft.com/en-us/dotnet/framework/data/adonet/ef/security-considerations (дата звернення: 08.05.2026).
49. Microsoft Learn. SQL Queries. Entity Framework Core Documentation [Електронний ресурс]. Режим доступу: https://learn.microsoft.com/en-us/ef/core/querying/sql-queries (дата звернення: 09.05.2026).
50. Маруняк А.О. Антонов Ю.С. Інтеграція патернів Repository та Unit of Work у Data Access Layer системи керування розкладом закладу вищої освіти. Прикладні інформаційні технології: матеріали VII Всеукр. наук-практ. конф. Здобувачів вищої освіти та молодих вчених (м. Вінниця, 15 трав. 2026 р.). Вінниця, 2026. С. 105-107.