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

КАЛЬКО ДЕНИС РОМАНОВИЧ

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

РОЗРОБКА СИСТЕМИ КЕРУВАННЯ РОЗКЛАДОМ ЗАНЯТЬ УНІВЕРСИТЕТУ: БІЗНЕС-ЛОГІКА ТА ВЕБІНТЕРФЕЙС

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

Калько Д.Р. Розробка системи керування розкладом занять університету: бізнес-логіка та вебінтерфейс. Спеціальність 122 «Комп'ютерні науки», освітня програма «Комп'ютерні науки». Донецький національний університет імені Василя Стуса, Вінниця 2026.

У кваліфікаційній (бакалаврській) роботі досліджено існуючі підходи до автоматизації керування розкладом занять у закладах вищої освіти та реалізовано веб-систему на платформі .NET з використанням Blazor Server, що охоплює сервісний шар бізнес-логіки, модуль валідації предметних обмежень та дві клієнтські підсистеми з розмежуванням ролей доступу.

Ключові слова: система керування розкладом, веб-система, Blazor, ASP.NET Core, бізнес-логіка, валідація, REST API, C#.

79 ст., 20 ліст., 18 рис., 3 табл., 2 дод., 44 джерел.

ABSTRACT

Kalko D.R. Development of a University Course Schedule Management System: Business Logic and Web Interface. Specialty 122 «Computer Science», educational program «Computer Science». Vasyl Stus Donetsk National University, Vinnytsia 2026.

In the qualification (bachelor's) work, existing approaches to automating schedule management in higher education institutions are investigated and a web system is implemented on the .NET platform using Blazor Server, encompassing a business logic service layer, a subject-matter constraints validation module, and two client subsystems with role-based access differentiation.

Keywords: schedule management system, web system, Blazor, ASP.NET Core, business logic, validation, REST API, C#.

79 p., 20 list., 18 fig., 3 tabl., 2 app., 44 sources.

ЗМІСТ

ВСТУП

Актуальність теми дослідження. Керування розкладом занять у закладах вищої освіти є складним організаційним процесом, що потребує одночасного врахування значної кількості обмежень. Більшість наявних рішень орієнтовані на іноземні освітні системи або є настільними застосунками без зручного веб-інтерфейсу. Поза увагою дослідників залишається задача побудови веб-системи з розмежуванням ролей доступу, вбудованою перевіркою бізнес-правил та публічним інтерфейсом перегляду розкладу без автентифікації.

Мета дослідження. Розробка веб-системи управління розкладом занять закладу вищої освіти з реалізацією сервісного шару бізнес-логіки, модуля валідації предметних обмежень, контролерів прикладного програмного інтерфейсу та клієнтських підсистем з розмежуванням доступу адміністратора і публічного користувача.

Завдання дослідження. Для досягнення поставленої мети в роботі вирішуються такі завдання:

1. Дослідити існуючі підходи до автоматизації процесу керування розкладом занять у навчальних закладах.
2. Проаналізувати функціональні вимоги до веб-системи керування розкладом занять.
3. Визначити оптимальну архітектуру та технології для реалізації веб-системи.
4. Розробити серверну та клієнтську частини веб-системи.
5. Реалізувати інтерфейс користувача з урахуванням вимог адаптивності та зручності використання.
6. Впровадити систему автентифікації та авторизації для забезпечення безпеки даних.

Об'єкт дослідження. Процес автоматизації керування розкладом занять у закладі вищої освіти.

Предмет дослідження. Методи та засоби реалізації сервісного шару бізнес-логіки, модуля валідації предметних обмежень і клієнтських підсистем з

розмежуванням ролей доступу в багатошаровій веб-системі на платформі .NET з використанням фреймворку Blazor.

Теоретичне та практичне значення одержаних результатів. Теоретичне значення роботи полягає в обгрунтуванні архітектурного підходу до побудови сервісного шару на основі узагальненого сервісу та модульної системи валідації, що може бути застосований при проєктуванні інших інформаційних систем. Практичне значення полягає у розробці функціональної веб-системи: адміністративна підсистема забезпечує повний цикл управління розкладом із перевіркою обмежень, а публічна – зручний доступ до розкладу без реєстрації.

Зв'язок роботи з науковими програмами, планами, темами. Наведені у бакалаврській роботі дослідження пов'язані з фундаментальною 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. Основні результати дослідження були представлені на наукових конференціях. За темою роботи підготовлено такі публікації:

1. Antonov Yu.S., Kalko D.R., Maruniak R.V. Development of a generalized data model and data access architecture for academic scheduling systems using .NET and Blazor. Збірник матеріалів IX Міжнародної науково-практичної конференції «Scientific Practice: Modern and Classical Research Methods». м. Бостон, США, 10 квітня 2026 р;
2. Калько Д.Р., Антонов Ю.С. Побудова шару бізнес-логіки багатошарових застосунків на основі Generic класу. Збірник матеріалів VII Всеукраїнської науково-практичної конференції здобувачів вищої освіти та молодих вчених «Прикладні інформаційні технології». Донецький національний університет імені Василя Стуса, м. Вінниця, 15 травня 2026 р.

Структура кваліфікаційної роботи: кваліфікаційна робота складається із вступу, трьох розділів та висновків до них, списку використаних джерел і двох додатків.

У першому розділі проаналізовано предметну область, існуючі рішення та теоретичні засади патернів проєктування.

У другому розділі сформульовано постановку задачі, визначено вимоги та спроєктовано архітектуру системи.

У третьому розділі реалізовано всі компоненти: сервісний шар, модуль валідації, контролери REST API та дві клієнтські підсистеми.

Кваліфікаційна робота включає 79 сторінок, 18 рисунків, 3 таблиці, 2 додатки та список літератури з 44 джерел.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ ПРОЄКТУВАННЯ БАГАТОШАРОВОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА ОГЛЯД АНАЛОГІВ

1.1 Загальна проблематика автоматизації керування розкладом у ЗВО

Розклад навчальних занять є одним із ключових регулюючих документів закладу вищої освіти (ЗВО), що визначає часовий, просторовий та кадровий розподіл навчального навантаження. Як зазначають автори в роботі [3], правильно складений розклад забезпечує рівномірне завантаження аудиторного фонду, студентських груп і професорсько-викладацького складу. Попри удавану простоту кінцевого результату – таблиці із заняттями – процес його формування є надзвичайно складним завданням, де одночасно перетинаються часові, просторові та ресурсні обмеження. Саме тому проблема автоматизації керування розкладом є предметом систематичних досліджень як в Україні, так і у світовій науковій спільноті [2, 3, 5, 10, 26, 33, 36, 37].

Організація навчального процесу у вітчизняних ЗВО характеризується рядом структурних особливостей, що принципово відрізняють цю предметну область від шкільного розкладу і безпосередньо визначають складність автоматизованих систем. Перша з них – періодична структура навчальних тижнів. На відміну від загальноосвітньої школи, де розклад переважно залишається статичним протягом семестру, у ЗВО широко застосовується чергування навчальних тижнів за схемою «чисельник/знаменник» (верхній/нижній тиждень). Це означає, що система повинна не лише зберігати два паралельних варіанти розкладу, але й автоматично визначати поточний тип тижня на основі календарної дати. Автори в роботі [10] прямо фіксують цей аспект у своєму дослідженні: при розробці автоматизованих систем виникають ситуації, коли необхідно розбити навчальні тижні на верхній та нижній, і відповідний функціональний модуль є обов'язковим елементом будь-якої

повноцінної системи керування розкладом.

Другою суттєвою особливістю є поділ академічних груп на підгрупи для проведення лабораторних, практичних або лекційних занять. Це породжує ситуацію, коли частина групи має заняття в одній аудиторії, а інша – в другій або ж взагалі має вільний час. У науковій літературі наголошують на необхідності підтримки різних групувань студентів – потік, підгрупа, зведена група – та контролю їх переміщення під час формування розкладу, щоб заняття для різних підгруп не дублювалися і не перетиналися [3]. Традиційні методи відображення – паперові стенди, PDF-файли або Excel-таблиці – нездатні ефективно візуалізувати такі розгалуження.

Третьою важливою характеристикою є рольова структура учасників навчального процесу. Як зазначається у дослідженні [7, 12, 14], здійсненому шляхом аналізу понад 60 сайтів закладів вищої освіти України, автор виділяє три принципово різні категорії користувачів системи керування розкладом: студентів, яким потрібен зручний доступ до актуального розкладу своєї групи; викладачів, що потребують перегляду власного навантаження; та диспетчерів деканату, які безпосередньо формують і редагують розклад. Кожна з цих ролей висуває специфічні вимоги до інтерфейсу та рівня доступу, що робить задачу проєктування системи значно складнішою порівняно зі звичайною системою перегляду даних.

Принципово важливим аспектом, що відрізняє сучасні вимоги до таких систем від підходів попереднього покоління, є динамічність навчального процесу. Викладачі можуть хворіти, аудиторії – закриватися на ремонт, заняття – переноситися. У роботі [2] підкреслюють: у будь-якому ЗВО навчається кілька тисяч студентів із різних міст, тож стежити за змінами у розкладі вдається не кожному, а розклад через різні причини є непостійним і може змінюватися протягом семестру. У випадку використання статичних файлів будь-яка зміна, внесена адміністратором в оригінал розкладу, не відображається автоматично на пристроях інших користувачів. Це призводить до десинхронізації інформації між учасниками навчального процесу.

З огляду на перелічені особливості предметної області, у науковій літературі прийнято класифікувати обмеження при формуванні розкладу на дві категорії – жорсткі та м'які [26, 33, 36, 37]. Жорсткі обмеження є умовами, які обов'язково повинні бути виконані: один викладач не може вести два заняття одночасно, одна аудиторія не може бути зайнята двома групами одночасно, кількість студентів у групі не повинна перевищувати місткість аудиторії. М'які обмеження – бажані, але не обов'язкові умови: рівномірність розподілу занять упродовж тижня, уникнення «вікон» у розкладі, врахування побажань викладачів щодо часу роботи. У таблиці 1.1 наведено основні типи обмежень з прикладами їх реалізації.

Таблиця 1.1 Основні типи обмежень при формуванні розкладу

Категорія	**Опис обмеження**	**Характер**
Часові	Робочі години, тривалість занять, перерви між парами	Жорсткий / м'який
Просторові	Доступність аудиторій, їх місткість, технічне оснащення	Жорсткий
Ресурсні	Навантаження викладачів, розподіл груп за предметами	Жорсткий / м'який
Логічні	Уникнення накладань занять, послідовність дисциплін	Жорсткий

Якщо розглянути еволюцію інструментів керування розкладом, простежується чітка траєкторія: від ручного складання з аркушем паперу та олівцем – до таблиць Excel і PDF, далі – до спеціалізованих настільних систем і, нарешті, до сучасних веб-орієнтованих рішень. У роботі [3] констатують, що документообіг, який супроводжує формування розкладу у більшості ЗВО, як правило, досі реалізується вручну або у вигляді файлів довільного формату, що суттєво ускладнює процес і підвищує ймовірність помилок. Кожен перехід між стадіями зумовлювався накопиченням критичних обмежень попереднього

підходу: файлові рішення не підтримують багатокористувацьку роботу, настільні системи без серверного компонента унеможливлюють централізоване оновлення даних у реальному часі.

Таким чином, задача автоматизації керування розкладом у ЗВО поєднує в собі організаційну складність (велика кількість взаємопов'язаних сутностей і обмежень), специфіку предметної області (періодичність тижнів, підгрупи, рольова структура) та технічні виклики (необхідність реального часу, кросплатформності та ролевого розмежування доступу). Саме ця сукупність чинників пояснює, чому попри значний обсяг наявних рішень, пошук ефективних підходів до розробки таких систем залишається актуальним завданням [5, 26, 33, 36, 37].

1.2 Аналіз існуючих систем керування розкладом

1.2.1 Корпоративні та університетські рішення міжнародного рівня

Серед міжнародних рішень особливе місце займає відкрита платформа UniTime (University Timetabling), що розробляється за ліцензією Apache License 2.0 та використовується університетами на кількох континентах. Рудова, Мюллер та Мюррей дослідили комплексну задачу складання розкладу на прикладі Університету Пардью з майже 2500 класами, понад 32 000 студентів та 200 аудиторіями. Автори запропонували модель на основі зваженої задачі задоволення обмежень, алгоритм ітеративного пошуку вперед із конфліктною статистикою та алгоритм гілок і меж [23, 36].

Практичне втілення цих підходів описано у роботі Мюллера, Мюррея та Шлюттенхофера: UniTime підтримує розподілену відповідальність факультетів за формування власних частин розкладу, інтерактивне внесення змін та онлайн-розподіл студентів по секціях у реальному часі [37]. Іншим помітним напрямом є врахування гібридних форм навчання при складанні розкладу. Давісон, Хейрі та Зографос запропонували багатоцільову модель бінарного програмування, що

послідовно оптимізує три цілі: максимізацію виконаних запитів на курси, мінімізацію відхилень від бажаного формату навчання та мінімізацію конфліктів у розкладі [26]. Попри широке впровадження змішаних форматів після пандемії COVID-19, жодна з відомих на той момент моделей не враховувала їх явно.

1.2.2 Вітчизняні рішення

Огляд зазначених рішень та критичний аналіз відмінностей між файловими і об'єктно-орієнтованими підходами до архітектури наведено у [14]. Спільним недоліком розглянутих міжнародних рішень з точки зору застосування у вітчизняних ЗВО є їх орієнтованість на специфіку організації навчального процесу, що відрізняється від прийнятих в Україні моделей верхнього/нижнього тижня, структури навчальних планів та рольової ієрархії деканатів і кафедр [4].

Серед вітчизняних розробок вагоме місце займає система керування розкладом на основі змішаної клієнт-серверної архітектури та архітектури трирівневих баз даних [7, 12]. Система складається з кількох взаємодіючих компонентів: Schedule Client – кросплатформний десктопна система для авторизованих користувачів; Schedule Site and Service – публічний веб-сайт і API для зовнішніх систем; Schedule WebService – сервіс генерації звітів. Для синхронізації між двома базами даних використовується механізм реплікації master/slave, що забезпечує відокремлення публічного доступу від конфіденційних даних. Усі зміни, внесені диспетчером, одразу автоматично стають доступними для всіх учасників навчального процесу. Серед ключових архітектурних рішень системи автора – розмежування ролей на рівні СУБД (ролі WebGuest, ScheduleDispatcher та інші), зберігання розкладу у форматі календаря замість таблиці верхнього/нижнього тижня, що дозволяє обліковувати не лише лекції та практики, але й семестрові консультації, сесію та нерегулярні події.

У роботі [6] розроблено автоматизовану систему формування розкладу на платформі 1С «Підприємство», що реалізує документоорієнтований підхід. Система охоплює довідники дисциплін, викладачів, типів занять і груп,

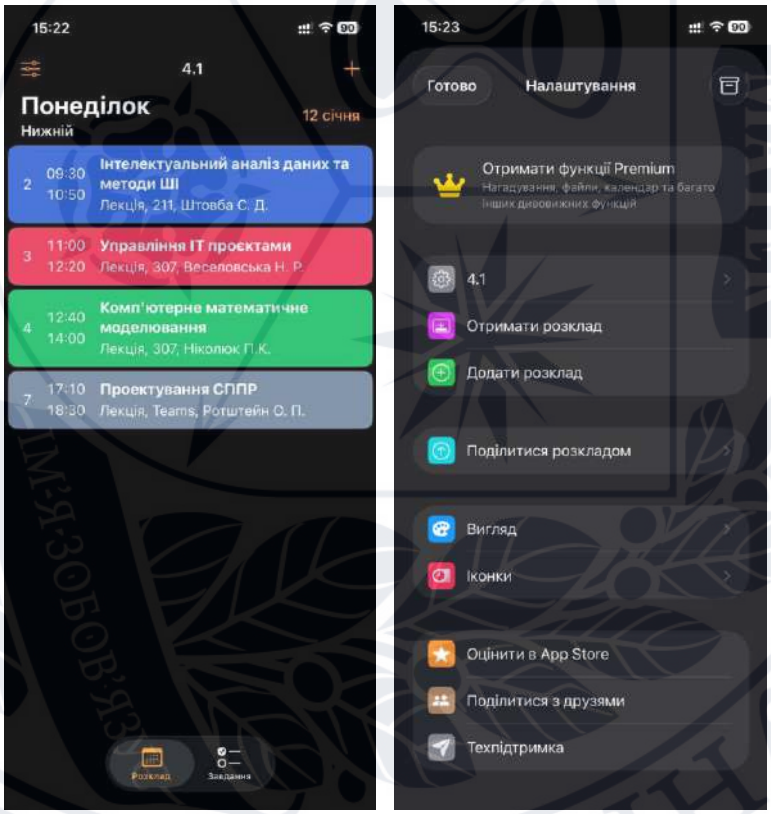
документи розкладу та робочих навчальних планів, а також забезпечує перевірки на конфлікти при збереженні: накладання викладача, групи або аудиторії в один часовий слот. Перевагою є швидкість розробки засобами платформи 1С, однак суттєвим обмеженням є прив'язаність до комерційної платформи, відсутність повноцінного веб-інтерфейсу та складність інтеграції з зовнішніми сервісами.

У роботі [1] розроблено систему формування розкладу навчальних занять на платформі 1С: Підприємство, що реалізує документоорієнтований підхід до організації навчального процесу [1]. Система охоплює кілька взаємопов'язаних етапів: формування нормативно-довідкової інформації – довідників аудиторій, дисциплін та обладнання; ведення навчальних і робочих навчальних планів; розподіл дисциплін між викладачами; безпосереднє формування розкладу з автоматичною перевіркою на накладання груп, викладачів та аудиторій. Система враховує об'єктивні обмеження – доступність ресурсів та нормативні вимоги – і суб'єктивні побажання учасників навчального процесу. Перевагою рішення є швидкість розробки засобами платформи 1С та наявність готових механізмів звітності. Суттєвим обмеженням є прив'язаність до комерційної платформи, відсутність повноцінного веб-інтерфейсу для кінцевих користувачів та складність інтеграції з зовнішніми сервісами.

Задачу документообігу при формуванні розкладу в університеті з 16 факультетами та 15 корпусами вирішено у багатомодульній системі [3]. Система складається з підсистем управління аудиторним фондом, академічними групами, навчальними планами та розповсюдження розкладу. Розповсюдження реалізовано двома способами: у форматі Excel для внутрішнього використання та через публічний інтернет-сервіс. Попри охоплення повного циклу документообігу, система має обмежений інтерфейс – переважно настільний – та не підтримує оновлень у реальному часі для кінцевих користувачів. Рольового розмежування між адміністративним та публічним доступом на рівні архітектури також не передбачено, що обмежує гнучкість керування правами користувачів.

1.2.3 Користувацько-орієнтовані застосунки

Окремого розгляду як аналог заслуговує мобільний застосунок «Timetable» (Розклад) – класичний приклад автономного персонального органайзера, орієнтованого на кінцевого користувача [42]. Головний екран реалізує патерн «Daily View» зі списком занять на обрану дату (рис. 1.1, а). Кожна картка заняття містить ключові дані: назву дисципліни, час початку та завершення, номер аудиторії та прізвище викладача. Навігація між днями тижня здійснюється горизонтальними свайпами, а бічне меню (рис. 1.1, б) виступає єдиною точкою входу для адміністративних функцій: створення нового розкладу, перемикання між шаблонами та налаштування теми.



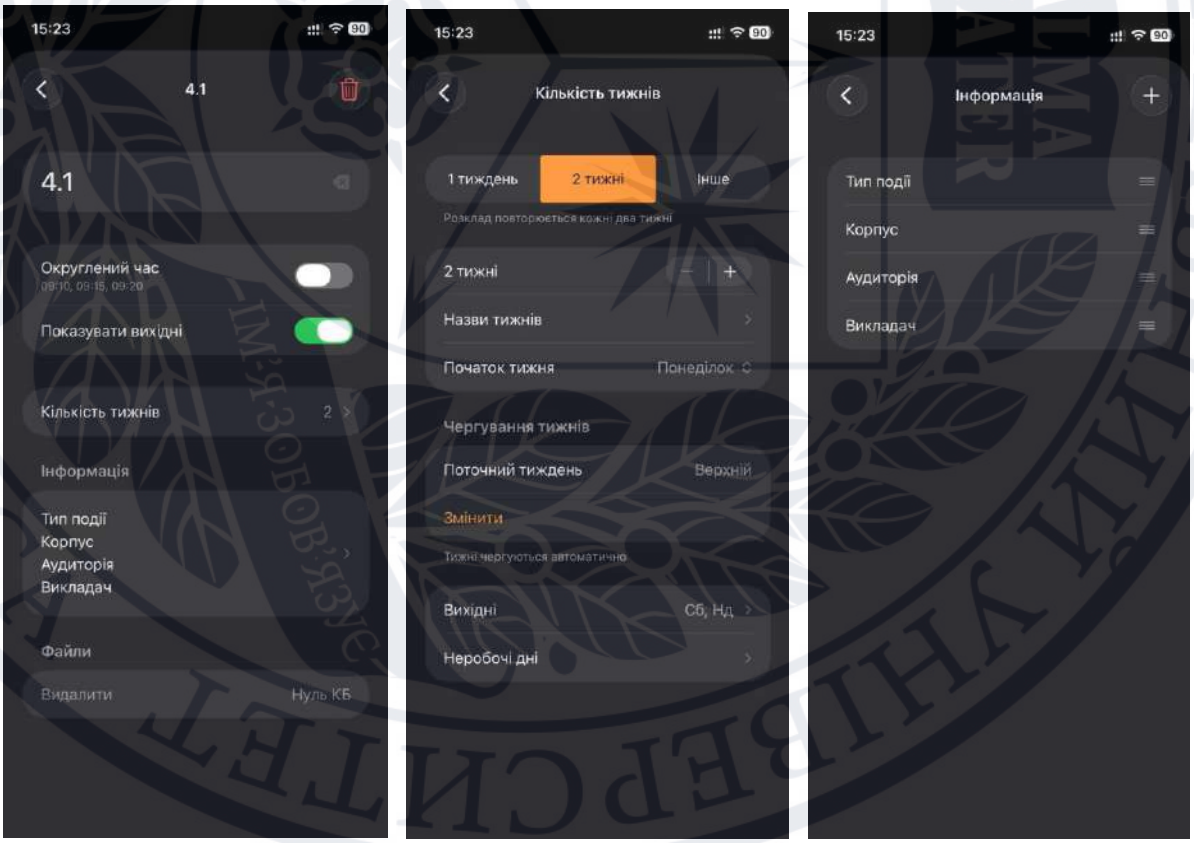
а) б)

Рисунок 1.1 - Застосунок «Розклад»

Застосунок підтримує стандартну для вітчизняних ЗВО модель «чисельник/знаменник»: користувач вручну вказує поточний тип тижня, після

чого алгоритм автоматично розраховує чергування для майбутніх дат. Наповнення розкладу відбувається через ручний конструктор сутностей (рис. 1.2): спочатку створюються довідники предметів, викладачів та аудиторій, а потім формуються конкретні заняття з кольоровим маркуванням.

Незважаючи на продуманий візуальний дизайн, застосунок має критичні обмеження для використання у масштабах університету. По-перше, відсутність кросплатформності: застосунок існує виключно у мобільній екосистемі, що унеможливлює адміністрування розкладу з персонального комп'ютера. По-друге, проблема синхронізації: поширення розкладу реалізовано через передачу статичного файлу-шаблону, тому будь-які зміни в оригіналі не відображаються автоматично на пристроях інших користувачів. По-третє, відсутність реляційної цілісності: при створенні заняття система пропонує невідфільтрований список усіх викладачів, незалежно від обраної дисципліни.



а) б) в)

Рисунок 1.2 - Головні налаштування розкладу

1.2.4 Узагальнення обмежень розглянутих рішень

Проведений аналіз дозволяє систематизувати типові прогалини наявних рішень. Міжнародні корпоративні платформи мають значну функціональну потужність, однак не адаптовані до специфіки вітчизняних ЗВО – зокрема термінології верхнього/нижнього тижня, структури кафедральної ієрархії та особливостей українських навчальних планів. Вітчизняні настільні рішення вирішують задачу адміністрування, але обмежені у доступності: відсутній або нерозвинений веб-інтерфейс для кінцевих користувачів, оновлення не поширюються миттєво [7, 10, 12, 33]. Системи, що описані, наприклад, в роботі [3] покривають документообіг, але публічна частина реалізована у форматі статичного файлу або простого сервісу без ролевого розмежування. Мобільні органайзери забезпечують зручний перегляд, проте принципово не масштабуються на рівень університету через локальне зберігання даних та відсутність серверного джерела правди [42].

Таким чином, у наявних рішеннях бракує саме повноцінної веб-системи, орієнтованого на українські ЗВО, з чітким ролевим розмежуванням між адміністративним інтерфейсом та публічним інтерфейсом перегляду, з динамічним оновленням даних та підтримкою специфіки вітчизняного навчального процесу на рівні бізнес-логіки.

1.3 Теоретичні засади модульної організації та патерни проєктування клієнтських інтерфейсів

Розвиток сучасних веб-технологій призвів до суттєвої трансформації підходів до проєктування програмного забезпечення [35]. Клієнтська частина веб-систем еволюціонувала від простих сторінок відображення контенту до складних інженерних рішень, що за своєю архітектурою та функціональністю наближаються до настільних операційних систем. Зростання складності вимог до інтерактивності, швидкодії та масштабованості диктує необхідність відмови

від монолітних структур на користь модульної організації коду [41]. У цьому підрозділі розглянуто фундаментальні принципи побудови компонентних систем, патерни управління даними та стратегії організації користувацької взаємодії, які є стандартом у сучасній індустрії розробки програмного забезпечення.

В основі сучасних SPA лежить концепція компонентно-орієнтованої архітектури [16, 21]. Це передбачає декомпозицію складного інтерфейсу користувача на набір незалежних, слабопов'язаних модулів. З теоретичної точки зору, компонент – це інкапсульована сутність, яка об'єднує в собі три складові: структуру (HTML-розмітку), стиль (CSS-правила) та поведінку (програмну логіку) [32].

Замість управління тисячами рядків коду в одному файлі, розробник оперує ієрархічним деревом компонентів.

Принципи організації компонентів:

1. Інкапсуляція та ізоляція: Кожен компонент функціонує як «чорна скринька». Внутрішня реалізація модуля прихована від зовнішнього середовища. Компонент взаємодіє з системою виключно через чітко визначені інтерфейси: вхідні параметри та вихідні сигнали. Це дозволяє змінювати внутрішню логіку одного елемента без ризику порушити роботу всієї системи;
2. Повторне використання: Виділення універсальних елементів інтерфейсу (наприклад, полів вводу, таблиць даних, модальних вікон) у окремі бібліотеки дозволяє використовувати їх у різних частинах проєкту. Це відповідає принципу DRY, зменшує обсяг кодової бази та спрощує тестування [38];
3. Композиція проти наслідування: У сучасній теорії проєктування інтерфейсів перевага надається композиції. Складні компоненти будуються шляхом об'єднання простіших, а не через створення глибоких ієрархій наслідування класів. Це забезпечує більшу гнучкість системи при зміні бізнес-вимог.

Важливим аспектом є поділ компонентів на дві категорії: презентаційні, які відповідають лише за візуалізацію отриманих даних, та компоненти-контейнери, які відповідають за логіку отримання даних та управління станом. Такий поділ дозволяє чітко розмежувати відповідальність у системі та спростити модульне тестування візуальної частини [34].

Ключовою відмінністю SPA від класичних веб-сайтів є складна модель життєвого циклу програмних модулів. У класичному вебі сторінка «живе» лише від моменту завантаження до моменту переходу на нову URL-адресу. У SPA компоненти можуть створюватися, оновлюватися та знищуватися динамічно, без перезавантаження сторінки браузера [16, 35].

Теорія життєвого циклу визначає набір етапів, через які проходить кожен елемент інтерфейсу:

- Ініціалізація: Момент створення екземпляра компонента. На цьому етапі відбувається впровадження залежностей та налаштування початкових параметрів;
- Рендеринг: Побудова віртуального дерева DOM (Document Object Model). Система обчислює, як повинен виглядати інтерфейс на основі поточних даних;
- Після-рендерінг: Виконання операцій, що вимагають доступу до вже створених елементів DOM;
- Знищення: Очищення ресурсів перед видаленням компонента з пам'яті. Коректна реалізація цього етапу є критичною для запобігання витокам пам'яті у довготривалих сесіях роботи.

Асинхронність є ще однією фундаментальною характеристикою. Сучасні інтерфейси не повинні блокувати основний потік під час очікування відповіді від сервера або виконання складних обчислень. Використання асинхронних патернів дозволяє інтерфейсу залишатися чутливим до дій користувача навіть під час виконання важких фонових задач [12, 15].

Однією з найскладніших проблем у проєктуванні клієнтських систем є управління станом даних. Стан – це сукупність усіх даних, що визначають

поточне відображення інтерфейсу (дані користувача, вміст форм, статус завантаження, активні фільтри тощо). Оскільки протокол HTTP не зберігає стану, обов’язок збереження контексту лягає на клієнтську систему[33].

Існує кілька рівнів організації стану:

1. Локальний стан: Дані, що стосуються лише конкретного компонента і не впливають на решту системи (наприклад, стан «розкрито/згорнуто» випадаючого списку або текст у полі пошуку до моменту відправки);
2. Глобальний стан: Дані, що використовуються багатьма компонентами одночасно (наприклад, профіль авторизованого користувача, глобальні налаштування теми тощо).

Основна ідея полягає у створенні «Єдиного джерела істини» – централізованого сховища даних [24]. Компоненти не змінюють дані напряму, а відправляють запити на зміну стану. Це забезпечує передбачуваність потоків даних та спрощує відстеження помилок.

Важливим теоретичним аспектом є концепція незмінності. При оновленні стану об'єкти не модифікуються, а замінюються новими копіями зі зміненими значеннями [40]. Це дозволяє механізмам рендерингу швидко визначати необхідність оновлення інтерфейсу шляхом простого порівняння посилань на об'єкти, що значно підвищує продуктивність системи.

Для забезпечення масштабованості та тестованості коду сучасна теорія розробки вимагає чіткого розділення бізнес-логіки та логіки відображення [5, 17, 40]. Це досягається шляхом впровадження багатошарової архітектури на клієнті.

Основні архітектурні шари:

- Шар представлення: Відповідає виключно за візуалізацію даних та трансляцію дій користувача у команди системи. Цей шар не повинен містити складної логіки обробки даних або прямих звернень до мережі;
- Шар абстракції / Сервісний шар: Набір сервісів, які інкапсулюють бізнес-правила та сценарії використання;
- Інфраструктурний шар: Відповідає за комунікацію із зовнішніми джерелами даних (REST API, WebSocket) та локальними сховищами

браузера.

Ключовим механізмом, що зв’язує ці шари, є DI – Впровадження залежностей. Цей патерн реалізує принцип інверсії управління, згідно з яким компоненти не створюють екземпляри необхідних їм сервісів самостійно, а отримують їх із зовнішнього контейнера [38]. Це дозволяє підміняти реалізації сервісів без зміни коду компонентів, що є необхідною умовою для якісного модульного тестування.

Однією з критичних технологій для роботи з великими масивами даних є віртуалізація. Рендеринг тисяч елементів браузері призводить до значного падіння продуктивності. Віртуалізація передбачає відображення лише тієї частини списку, яка фізично поміщається у видиму область екрану. При прокручуванні вміст динамічно підміняється, створюючи ілюзію нескінченного списку. Це дозволяє працювати з масивами даних будь-якого розміру без втрати плавності інтерфейсу [27, 28].

Іншим важливим аспектом є стратегія зворотного зв’язку. Оскільки веб-системи залежать від мережевих затримок, система повинна постійно інформувати користувача про свій стан. Використання патернів «Скелетний екран» замість традиційних спінерів завантаження дозволяє знизити суб’єктивне відчуття очікування, демонструючи структуру контенту ще до його фактичного отримання [20].

Також у сучасній розробці широко застосовується концепція адаптивних систем сіток. На відміну від створення окремих мобільних версій, адаптивний дизайн використовує математичні правила для перерозподілу простору екрану. Це дозволяє інтерфейсу автоматично трансформуватися: наприклад, багатоколонкова таблиця на десктопі може перетворюватися на набір вертикальних карток на мобільному пристрої, зберігаючи при цьому повну функціональність [19, 20].

Підсумовуючи, можна стверджувати, що створення сучасної клієнтської системи вимагає глибокого розуміння теорії програмної інженерії. Використання компонентної моделі, грамотне управління станом, застосування патернів є

необхідним фундаментом для побудови надійних, швидких та зручних у підтримці програмних продуктів. Ці теоретичні засади формують базис, на якому будується практична реалізація будь-якої веб-системи корпоративного рівня.

Висновок до розділу 1

У першому розділі проведено аналіз предметної області та огляд існуючих рішень для управління розкладом занять. Встановлено, що періодичність навчальних тижнів, поділ груп на підгрупи та динамічне відображення змін вимагають від системи високої гнучкості.

Кожен клас розглянутих рішень має характерні сильні сторони: UniTime демонструє розподілену відповідальність за формування розкладу, вітчизняна клієнт-серверна система – чітке розмежування ролей, рішення на платформі 1С – перевірку конфліктів у момент збереження. Спільним недоліком є відсутність повноцінної веб-системи з ізольованим адміністративним інтерфейсом, відкритим публічним переглядом та вбудованою перевіркою бізнес-правил.

Огляд теоретичних засад підтвердив, що компонентна модель, впровадження залежностей та розмежування шарів є стандартом для систем корпоративного рівня і безпосередньо визначають архітектуру обох клієнтських підсистем.

РОЗДІЛ 2. ПОСТАНОВКА ЗАДАЧІ ТА ПОБУДОВА АРХІТЕКТУРИ

2.1 Постановка задачі та межі дослідження

Проєктування та реалізація програмного забезпечення потребує чіткого окреслення меж роботи ще до переходу до архітектурних рішень. Без такого окреслення виникає ризик розмивання відповідальності між компонентами системи, ускладнення оцінки результатів та невідповідності між метою і реалізованим функціоналом.

У межах даної роботи предметом розробки є дві взаємопов'язані складові системи керування розкладом занять закладу вищої освіти: шар бізнес-логіки та клієнтська частина. Шар доступу до даних, структура бази даних, деталі реалізації механізмів збереження, бази даних та конфігурації засобів об'єктно-реляційного відображення не є предметом дослідження цієї роботи.

Таким чином, робота охоплює три логічні рівні: сервісний шар, що реалізує бізнес-правила та координує обробку даних; модуль валідації, що перевіряє коректність операцій із розкладом відповідно до предметних обмежень; контролери прикладного програмного інтерфейсу, що обробляють вхідні запити та формують відповіді; а також дві клієнтські підсистеми з розмежуванням ролей доступу [14].

Розмежування ролей є принциповим архітектурним рішенням цієї роботи. Як зазначалося в аналітичному огляді, система обслуговує три принципово різні категорії зацікавлених сторін із відмінними потребами та рівнем доступу:

1. Студенти – кінцеві користувачі, зацікавлені у зручному та оперативному доступі до актуального розкладу своєї групи без необхідності реєстрації чи автентифікації;
2. Викладачі – потребують можливості перегляду власного персонального розкладу у зручному форматі, також без необхідності входу до системи;
3. Технічний персонал деканату – здійснює повноцінне адміністрування системи: ведення довідників навчального процесу, формування й

редагування розкладу, контроль навантаження викладачів. Ця категорія користувачів потребує автентифікації та отримує розширені права доступу до всіх функцій управління.

Важливо зазначити, що перші дві категорії можуть бути будь-якими зовнішніми відвідувачами, які не мають відношення до конкретного закладу. Система не вимагає від них жодної ідентифікації: достатньо обрати потрібну групу або викладача для перегляду розкладу.

Відповідно до такого поділу зацікавлених сторін варто розробити дві ізольовані клієнтські підсистеми. Перша – адміністративна підсистема, призначена для технічного персоналу. Вона забезпечує повний цикл управління даними: від ведення базових довідників до безпосереднього формування розкладу із перевіркою бізнес-правил. Доступ до неї здійснюється виключно після проходження автентифікації. Друга – публічна підсистема для перегляду розкладу, яка не потребує жодних облікових даних і орієнтована на зручне, швидке отримання інформації про розклад за заданими критеріями.

Таке розділення на дві самостійні підсистеми, що взаємодіють із спільним сервісним шаром через єдиний прикладний програмний інтерфейс, є основним архітектурним рішенням, яке визначає всі подальші проєктні та реалізаційні рішення даної роботи. Саме воно зумовлює структуру контролерів, організацію сервісного шару, підхід до валідації та принципи побудови інтерфейсних компонентів.

2.2 Функціональні та нефункціональні вимоги

Визначення вимог є фундаментальним етапом проєктування будь-якого програмного забезпечення, оскільки саме від їх повноти та точності залежить відповідність кінцевого продукту потребам зацікавлених сторін. Функціональні вимоги описують конкретні дії та поведінку, яку система повинна виконувати, тоді як нефункціональні визначають атрибути якості – тобто те, наскільки добре ці дії виконуються. У контексті даної роботи вимоги формулюються окремо для

кожної з двох клієнтських підсистем, що відображає ізоляцію адміністративного та публічного інтерфейсів [9, 11].

2.2.1 Функціональні вимоги адміністративної підсистеми

Адміністративна підсистема орієнтована на технічний персонал деканату та надає повний набір інструментів для формування й підтримки актуального розкладу. Доступ до неї можливий виключно після успішної автентифікації, що є базовою умовою захисту адміністративних функцій.

До функціональних вимог адміністративної підсистеми належать:

1. Автентифікація та керування сеансом: вхід до системи за обліковими даними, можливість завершення сеансу. Усі операції зі зміни даних доступні виключно автентифікованому користувачу [31];
2. Керування довідниками навчального процесу: підсистема забезпечує повний цикл операцій – перегляд, створення, редагування та видалення – для всіх базових сутностей предметної області. Для кожного довідника передбачена можливість фільтрації;
3. Керування навчальним навантаженням викладачів: підсистема дозволяє призначати викладачам навантаження за конкретними дисциплінами й типами занять у розрізі семестрів, а також переглядати зведений стан навантаження для контролю його відповідності нормам;
4. Формування та редагування розкладу: адміністратор може додавати заняття до розкладу, визначаючи групу або підгрупу, дисципліну, тип заняття, викладача, аудиторію, часовий слот, день тижня та тип тижня. Передбачено редагування і видалення існуючих записів розкладу;
5. Вбудована перевірка бізнес-правил: при кожній спробі зберегти або змінити запис розкладу система автоматично перевіряє коректність операції відповідно до встановлених предметних обмежень [3, 10]. У разі порушення будь-якого з правил адміністратор отримує відповідне повідомлення про помилку.

2.2.2 Функціональні вимоги публічної підсистеми

Відкритість публічного інтерфейсу є принциповою архітектурною вимогою: розклад занять є суспільно значущою інформацією, доступ до якої не повинен вимагати ідентифікації користувача. Водночас відсутність автентифікації не знижує рівень захисту системи – публічна підсистема фізично ізольована від адміністративної і не має доступу до операцій зміни даних на жодному рівні архітектури.

До функціональних вимог публічної підсистеми належать:

1. Перегляд розкладу у режимі студента: користувач обирає факультет, спеціальність і навчальну групу, після чого отримує актуальний розклад занять у табличному форматі [9];
2. Перегляд розкладу у режимі викладача: користувач обирає кафедру та викладача, після чого отримує його персональний розклад занять з відображенням груп та аудиторій;
3. Автоматичне визначення поточного тижня: система самостійно обчислює, який тиждень є поточним на основі дати початку навчання, що зберігається для кожної групи;
4. Фільтрація та навігація: обидва режими перегляду підтримують ієрархічну фільтрацію, що послідовно звужує вибірку - від найзагальнішої категорії до конкретного об'єкту. Передбачено також навігацію між тижнями для перегляду розкладу на майбутні або минулі дати.

2.2.3 Нефункціональні вимоги

Нефункціональні вимоги визначають атрибути якості системи і встановлюють стандарти того, наскільки добре реалізовані функції виконуються в реальних умовах експлуатації. Нефункціональні вимоги охоплюють такі характеристики, як продуктивність, безпека, зручність використання, надійність

та розширюваність [9, 11]. Нефункціональні вимоги визначають якість програмного продукту в цілому, а не окремих його функцій. Їх виконання є необхідною умовою практичної придатності системи до реального використання.

Для розробленої системи визначено такі нефункціональні вимоги:

1. Продуктивність: обидві підсистеми повинні забезпечувати відчутну швидкість реакції на дії користувача. Операції читання даних мають виконуватися без помітних затримок. Операції зміни даних у адміністративній підсистемі допускають незначно більший час очікування, зумовлений виконанням валідації;
2. Безпека та розмежування доступу: функції зміни даних повністю ізольовані у межах адміністративної підсистеми і недоступні з публічного інтерфейсу на рівні архітектури. Публічна підсистема не отримує жодного доступу до операцій запису незалежно від дій користувача [31];
3. Зручність використання: інтерфейс обох підсистем повинен бути інтуїтивно зрозумілим без попереднього навчання. Адміністративний інтерфейс забезпечує зрозумілу навігацію між розділами. Публічний інтерфейс орієнтований на мінімальну кількість кроків від відкриття сторінки до отримання потрібного розкладу;
4. Адаптивність до різних пристроїв: публічна підсистема повинна коректно відображатися на пристроях із різними розмірами екрана - від мобільних телефонів до настільних комп'ютерів, оскільки основна аудиторія переглядає розклад саме зі смартфонів [19, 20];
5. Розширюваність та супроводжуваність: архітектура системи повинна допускати додавання нових правил валідації, нових сутностей довідників і нових інтерфейсних компонентів без значних змін у наявному коді. Це досягається через дотримання принципу відкритості/закритості та чітке розділення відповідальності між шарами

У таблиці 1.1 наведені основні типи обмежень, які повинні враховуватися при розробці системи.

Виходячи з проведеного аналізу предметної області та формалізації задачі, можна сформулювати загальну постановку задачі даної роботи:

Розробити шар бізнес-логіки та клієнтську частину системи управління розкладом навчального закладу, що забезпечує:

1. Зручний та інтуїтивно зрозумілий інтерфейс для різних категорій користувачів;
2. Гнучкі інструменти пошуку, фільтрації та представлення інформації про розклад;
3. Комплексне управління базовими даними для розкладу з можливістю фільтрації, включаючи:
   - Ведення обліку навчальних груп, підгруп та потоків;
   - Управління викладацьким складом та їх навчальним навантаженням;
   - Облік аудиторного фонду з урахуванням типу приміщень та технічного оснащення;
   - Управління навчальними планами та дисциплінами;
4. Розмежування прав доступу через реалізацію двох ізольованих підсистем: адміністративної та публічної;
5. Перевірку коректності даних розкладу через набір бізнес-правил, що охоплюють часові, просторові та логічні обмеження [5, 26, 33, 36, 37];
6. Інструменти для перегляду розкладу за різними критеріями (за групами, викладачами).

2.3 Обгрунтування вибору технологічної платформи та архітектури клієнтської частини системи

Вибір технологічного стеку для реалізації інтерфейсу користувача є одним із фундаментальних етапів проєктування будь-якої інформаційної системи. Це

рішення визначає не лише візуальну складову кінцевого продукту, але й його архітектурну гнучкість, масштабованість, швидкодію та зручність подальшої підтримки. В умовах високої динаміки розвитку веб-технологій та різноманіття клієнтських пристроїв, ключовими вимогами до клієнтської частини сучасних систем управління даними стають: кросплатформність, забезпечення миттєвої актуалізації даних, висока інтерактивність [16, 35, 44].

Для реалізації подібних вимог у сучасній інженерії проводиться детальний порівняльний аналіз доцільності використання різних архітектурних підходів, зокрема дилеми «Native vs Web», а також вибір конкретного фреймворку для реалізації шару представлення [34].

Сучасне середовище користувачів є вкрай різноманітним: смартфони різних поколінь, планшети, ноутбуки та стаціонарні ПК [19]. Орієнтація на мобільні рішення вимагала б підтримки щонайменше декількох окремих кодових баз – Android, iOS та веб-панелі для адміністраторів, – що неминуче призводить до дублювання бізнес-логіки та ризику розсинхронізації алгоритмів обробки даних. Додатково, життєвий цикл нативних застосунків жорстко прив'язаний до екосистем магазинів застосунків, де публікація навіть критичних виправлень може зайняти кілька днів через процедури модерації.

Ефективною альтернативою є архітектура системи з єдиною сторінкою, де завантажується одна HTML-оболонка, а вміст динамічно оновлюється через взаємодію з сервером по програмному інтерфейсу [15, 35]. Такий підхід забезпечує три ключові переваги:

- універсальність – клієнтський код виконується у будь-якому браузері та автоматично адаптується до розміру екрана [19, 20];
- архітектурну ізоляцію – клієнтська і серверна частини розробляються незалежно, маючи єдиною точкою дотику контракти даних [22];
- миттєве розгортання – оновлення інтерфейсу стає доступним усім користувачам одразу після публікації на веб-сервері без будь-яких процедур модерації.

На ринку веб-розробки наразі існує кілька усталених підходів до побудови подібних систем: поєднання серверного фреймворку мовою C# з клієнтськими фреймворками на JavaScript, повністю JavaScript-орієнтований стек або використання фреймворків, що дозволяють застосовувати єдину мову на всіх рівнях [16, 21, 32]. У таблиці 2.1 наведено порівняльний аналіз чотирьох альтернативних конфігурацій за ключовими критеріями, що є значущими для даного проєкту.

Комбінація ASP.NET Core з екосистемою Blazor є єдиним з розглянутих варіантів, що забезпечує повну єдність мовного середовища між сервером і клієнтом [15, 16]. Це дозволяє використовувати спільні об'єкти передачі даних в обох клієнтських підсистемах без підтримки паралельних описів різними мовами, а будь-яка неузгодженість між моделями виявляється на етапі компіляції. Платформа .NET забезпечує високу продуктивність, зрілу екосистему бібліотек, вбудовану підтримку впровадження залежностей та автентифікації, а ASP.NET Core надає засоби для побудови контролерів прикладного програмного інтерфейсу з маршрутизацією та обробкою помилок [25, 43].

Для побудови клієнтської частини обрано Blazor у серверній моделі рендерингу, де логіка компонентів виконується на сервері з передачею змін інтерфейсу клієнту через постійне з'єднання. Порівняно з моделлю WebAssembly ця модель забезпечує відсутність значного початкового завантаження, повний доступ до серверних ресурсів без додаткових HTTP-викликів та спрощену конфігурацію автентифікації в адміністративній підсистемі [16, 31].

Entity Framework Core виконує роль точки дотику між сервісним шаром та шаром доступу до даних, надаючи об'єктно-реляційне відображення без необхідності написання SQL-запитів вручну. Оскільки шар доступу до даних не є предметом даної роботи, Entity Framework Core розглядається виключно як контракт, через який сервісний шар отримує та зберігає дані: безпосередня конфігурація контексту бази даних і міграції виходять за межі дослідження.

Таблиця 2.1 Порівняльна таблиця варіантів технологічного стеку

Критерій	**ASP.NET Core + Blazor Server**	**React + Node.js**	**Angular + .NET API**	**Django + Vue.js**
Мова програмування	C#	JavaScript	TypeScript / C#	Python / JavaScript
Тип взаємодії з клієнтом	SignalR, серверний рендеринг	REST API + клієнтський рендеринг	REST API + клієнтський рендеринг	REST API + клієнтський рендеринг
Швидкість розробки	Висока (єдина мова)	Середня	Середня	Середня
Продуктивність	Висока	Висока	Висока	Середня
Масштабованість	Середня	Висока	Висока	Середня
Інтеграція з БД	Нативна через Entity Framework	Через сторонні ORM	Нативна через Entity Framework	Нативна через Django ORM
Спільний код між шарами	Повна (C# на клієнті й сервері)	Відсутня	Часткова	Відсутня

Для оформлення інтерфейсу використовується Bootstrap – широко поширена бібліотека стилів, що забезпечує адаптивну сітку та готові елементи керування. Вибір Bootstrap зумовлений його простою інтеграцією зі стандартними Razor-компонентами без необхідності встановлення та налаштування сторонніх бібліотек компонентів, а також коректною роботою на пристроях із різними розмірами екрана, що відповідає нефункціональній вимозі адаптивності публічної підсистеми [19, 20].

Таким чином, обраний стек забезпечує технологічну єдність між усіма компонентами системи, відповідає сформульованим функціональним і

нефункціональним вимогам та мінімізує кількість різнорідних технологій, що потребують одночасного знання при розробці та супроводі.

2.4 Архітектури системи

2.4.1 Багатошарова архітектура

Архітектура системи управління розкладом повинна враховувати вимоги масштабованості, гнучкості та зручності супроводу [18, 41]. Для розробки системи управління розкладом обрано багатошарову архітектуру з поділом на наступні рівні:

1. Рівень представлення (Presentation Layer) – відповідає за взаємодію з користувачем, відображення даних та отримання введених даних. На цьому рівні реалізовано веб-інтерфейс з використанням технології Blazor Server, що реалізує користувацький інтерфейс з мінімальним навантаженням на клієнтську частину;
2. Рівень бізнес-логіки (Business Logic Layer) – містить компоненти, що реалізують основну функціональність системи, обробку даних, валідацію, формування розкладу та інші алгоритми. Цей рівень реалізовано як набір сервісів, кожен з яких відповідає за певну функціональну область;
3. Рівень доступу до даних (Data Access Layer) – забезпечує взаємодію з базою даних, абстрагує деталі зберігання даних від бізнес-логіки;
4. Рівень даних (Data Layer) – включає базу даних та систему управління базами даних (MySQL), що забезпечує зберігання всієї інформації системи.

Така архітектура забезпечує:

- Розділення відповідальності – кожен рівень виконує чітко визначені функції, що спрощує розробку та тестування;

- Гнучкість та масштабованість – можливість незалежного масштабування кожного рівня;
- Тестованість – можливість тестування кожного компонента окремо;
- Простоту супроводу – зміни в одному компоненті мінімально впливають на інші частини системи.

2.4.2 Потік даних

Взаємодія між компонентами системи організована за допомогою шаблону впровадження залежностей, що забезпечує слабке зв'язування між компонентами та спрощує тестування і заміну реалізацій [39].

Основний потік даних організовано наступним чином (рис. 2.1):

1. Користувач взаємодіє з інтерфейсом, що генерується компонентами Blazor;
2. Компоненти Blazor викликають методи відповідних сервісів;
3. Сервіси містять бізнес-логіку та використовують репозиторії для доступу до даних;
4. Репозиторії взаємодіють з контекстом бази даних для виконання операцій з даними;
5. Результати операцій повертаються назад через ланцюжок компонентів до інтерфейсу.

Суцільні стрілки позначають запити: користувач звертається до компонентів Blazor, які викликають методи сервісів, а сервіси – методи репозиторіїв для виконання операцій з базою даних. Пунктирні стрілки позначають зворотний шлях відповідей: результати операцій повертаються від бази даних через репозиторії та сервіси до компонентів Blazor, які оновлюють інтерфейс користувача:

Для забезпечення асинхронної роботи та відсутності блокування інтерфейсу при виконанні тривалих операцій використовується асинхронне програмування з використанням Task-based Asynchronous Pattern [15].

Взаємодія між компонентами системи відбувається наступним чином:

Бізнес-логіка системи управління розкладом реалізована через сервісний шар, який виступає посередником між контролерами та сховищем даних [6, 41]. Такий підхід забезпечує розділення відповідальності між компонентами системи, підвищує тестованість та спрощує подальшу підтримку системи.

Рис. 2.1. Основний потік даних у системі

Принцип інверсії залежностей реалізується через використання інтерфейсів та контейнера залежностей, що дозволяє легко замінювати реалізації компонентів під час тестування або розвитку системи. Це забезпечує гнучкість архітектури та спрощує процес рефакторингу [18, 39].

Сервісний шар виступає центральним компонентом архітектури, інкапсулюючи всю бізнес-логіку системи. Базова структура сервісів побудована на принципах об'єктно-орієнтованого програмування, зокрема наслідування та поліморфізму. Абстрактний базовий клас визначає контракт для всіх операцій CRUD (Create, Read, Update, Delete), а спеціалізовані сервіси розширюють цю функціональність відповідно до специфічних вимог предметної області [6].

Ключовими функціями сервісного шару є:

- Валідація бізнес-правил на рівні домену;
- Управління транзакціями та забезпечення ACID-властивостей;
- Координація роботи з репозиторіями та іншими сервісами;
- Трансформація даних між різними рівнями абстракції.

Контролери виступають точкою входу для обробки HTTP-запитів від клієнтів і є ключовими компонентами архітектури WebAPI системи. Вони забезпечують інтерфейс між клієнтською частиною та бізнес-логікою системи [30].

Використання Data Transfer Objects (DTO) представляє реалізацію патерну «Об'єкт передачі даних», який забезпечує інкапсуляцію даних для передачі між різними рівнями системи або розподіленими компонентами системи [22]. Маппінг між DTO та доменними об'єктами реалізує патерн Mapper, який забезпечує двостороннє перетворення об'єктів різних типів [40].

Для забезпечення цілісності даних при складних операціях, що зачіпають кілька сутностей, використовується патерн «Одиниця роботи» (Unit of Work). Він забезпечує атомарність транзакцій та дозволяє відкотити зміни у разі помилки. UnitOfWork інкапсулює репозиторії для всіх сутностей системи та надає єдиний інтерфейс для роботи з ними [29].

Усі операції, що змінюють стан системи, виконуються через цей клас, що забезпечує узгодженість даних та спрощує управління транзакціями. Важливим аспектом забезпечення цілісності даних є також перевірка залежностей при видаленні записів. Наприклад, перед видаленням відділу система перевіряє, чи не пов'язаний він з аудиторіями, викладачами або іншими сутностями, і блокує видалення, якщо знайдені залежні записи.

2.4.3 Фізична структура рішення

Програмне рішення організовано у вигляді набору ізольованих проєктів у межах одного сховища коду, де кожен проєкт відповідає певному шару або підсистемі і має чітко визначені залежності від інших (рис. 2.1). Границі між шарами забезпечуються на рівні структури рішення: клієнтська підсистема отримує посилання лише на проєкти сервісного шару та спільних об'єктів передачі даних, але залежність від шару доступу до даних не задекларована – компілятор унеможливлює її використання. Такий підхід реалізує обмеження архітектурних рішень, описаних у попередньому підрозділі, засобами системи збірки, а не лише домовленостями в команді (рис. 2.2).

Спільний проєкт об'єктів передачі даних (UniSS-Shared) підключається одночасно до сервісного шару та обох клієнтських підсистем, утворюючи єдиний

контракт між серверною і клієнтською сторонами. Будь-яка зміна у структурі об'єкта передачі даних одразу відображається як помилка компіляції у всіх залежних проєктах, що унеможливлює розсинхронізацію моделей між підсистемами. Обидві клієнтські підсистеми підключають той самий сервісний шар через спільний метод розширення без дублювання конфігурації.

Така організація проєктів відображає архітектурні рішення підрозділу 2.4.1 на рівні структури рішення: кожен шар має рівно одне місце у сховищі коду, а перелік залежностей між проєктами є вичерпним описом дозволених взаємодій між шарами.

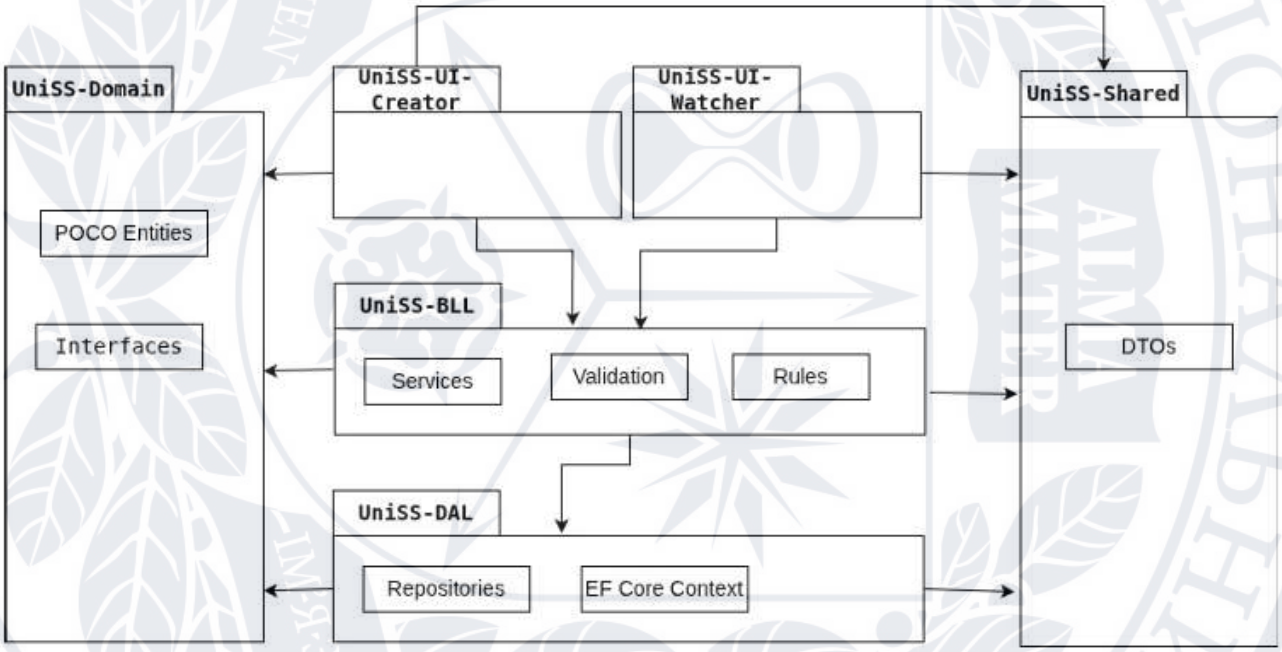


Рисунок 2.2 – Взаємодія пакетів у системі [13]

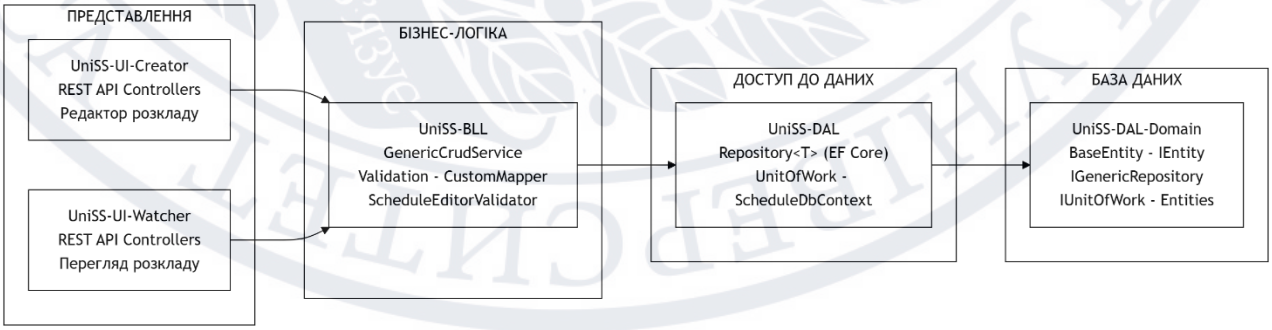


Рисунок 2.3 – Архітектура системи за шарами

2.5 Архітектурні та проєктувальні патерни

У даній роботі застосовано сукупність взаємодоповнюючих патернів, кожен з яких вирішує конкретну проєктну задачу на відповідному рівні системи [18, 41].

Впровадження залежностей: об'єкт отримує необхідні залежності ззовні через конструктор, а не створює їх самостійно. Платформа .NET надає вбудований контейнер, що керує часом життя об'єктів. В системі застосовуються три режими: одиничний – для сервісів стану, обмежений запитом – для сервісів бізнес-логіки, тимчасовий – для допоміжних об'єктів. Усі залежності між шарами є явними [39].

Сервісний шар: сервісний шар зосереджує всю бізнес-логіку системи і виступає посередником між контролерами та шаром доступу до даних. Контролери делегують обробку запитів сервісам, репозиторії не знають про правила предметної області. Саме сервісний шар координує виклик валідатора, перетворення даних та виклик репозиторіїв у потрібному порядку [6, 41].

Узагальнений сервіс з типовими операціями: система містить понад двадцять довідникових сутностей, кожна з яких потребує однотипного набору операцій читання, створення, оновлення та видалення. Для усунення дублювання типові операції реалізовано один раз у параметризованому базовому класі, а конкретний тип сутності та об'єкта передачі даних передаються як параметри типу. Спеціалізовані сервіси успадковують базову реалізацію та розширюють її лише специфічними методами [6].

Об'єкт передачі даних та маппер: між шарами дані передаються у вигляді спеціальних об'єктів передачі даних, а не доменних сутностей. Це приховує внутрішню структуру бази даних від зовнішніх споживачів, дозволяє передавати лише потрібні поля та унеможливлює випадкову зміну стану доменного об'єкта з рівня представлення. Перетворення між сутностями та об'єктами виконує маппер – окремий компонент, що інкапсулює всі правила відображення полів [22].

Стратегія для правил валідації: кожне бізнес-правило валідації розкладу реалізовано як окремий незалежний клас, що відповідає єдиному контракту. Валідатор отримує колекцію правил через впровадження залежностей та послідовно виконує кожне, збираючи порушення у спільний звіт. Додавання нового правила не потребує змін у існуючих класах – достатньо зареєструвати новий клас у контейнері залежностей [18, 41].

Компонентний підхід у клієнтській частині: кожен елемент інтерфейсу реалізований як самодостатній Blazor-компонент з власною розміткою та логікою. Компоненти вкладаються та повторно використовуються на різних сторінках. Для типових сторінок довідників виділено базові компоненти, від яких успадковуються конкретні сторінки – за аналогією з узагальненим сервісом на рівні бізнес-логіки [16, 17].

Архітектурний стиль REST у контролерах: контролери дотримуються стилю REST, де тип операції визначається методом запиту. Контролери є тонким шаром маршрутизації без бізнес-логіки: отримують запит, делегують сервісу та повертають стандартизовану відповідь з відповідним кодом стану або структурованим описом помилки [25, 30, 43].

Сукупність цих патернів забезпечує систему, де зміни в одному компоненті мінімально впливають на інші, а розширення функціональності зводиться до додавання нових класів.

2.6 Проєктування сервісного шару

Сервісний шар є центральним елементом бізнес-логіки системи і виступає єдиною точкою входу для всіх операцій із даними предметної області. Контролери прикладного програмного інтерфейсу не містять жодної доменної логіки та звертаються виключно до сервісів; репозиторії, у свою чергу, не мають доступу до правил предметної галузі та відповідають лише за збереження й отримання даних [6].

Оскільки переважна більшість сутностей системи потребує однотипного набору операцій, основою сервісного шару є параметризований узагальнений сервіс, що реалізує єдиний інтерфейс для всіх сутностей. Цей інтерфейс визначає п'ять базових операцій: отримання всіх записів, отримання запису за ідентифікатором, створення, оновлення та видалення. Усі п'ять операцій є асинхронними та підтримують токен скасування для коректного завершення у разі переривання запиту.

Реалізація цього узагальненого сервісу отримує через механізм впровадження залежностей чотири компоненти: одиницю роботи для доступу до репозиторіїв, реєстратор подій для журналювання, маппер для перетворення між доменними сутностями й об'єктами передачі даних, а також валідатор для перевірки коректності даних перед збереженням. Конкретний репозиторій для кожного типу сутності визначається у спадкоємному класі через перевизначення відповідної властивості. Аналогічно, правила перетворення між сутністю та об'єктом передачі даних визначаються у спадкоємному класі, а не в базовому – це дозволяє враховувати специфіку кожної предметної сутності без ускладнення базової логіки [6, 22].

Потік виконання операції створення нового запису є таким: сервіс спочатку передає отриманий об'єкт передачі даних валідатору; якщо той повертає звіт з помилками, сервіс негайно зупиняє виконання та сигналізує про помилку валідації; якщо перевірки пройдені успішно, сервіс перетворює об'єкт передачі даних на доменну сутність, передає її до репозиторію для збереження, фіксує транзакцію та повертає результат у вигляді об'єкта передачі даних. Операція оновлення наслідує аналогічний порядок, але перед перетворенням додатково отримує існуючий запис з репозиторію, щоб переконатись у його наявності. Операція видалення валідацію не виконує, проте може додатково перевіряти наявність залежних записів, що унеможливлює порушення реляційної цілісності [22, 29].

Для кожного довідника системи реалізовано окремий спеціалізований сервіс, що успадковує узагальнений сервіс та у більшості випадків лише визначає

посилання на відповідний репозиторій. За потреби спеціалізований сервіс додає власні методи пошуку – наприклад, пошук за повною назвою або за скороченням [27, 28].

Сервіс розкладу є найскладнішим компонентом сервісного шару, оскільки операція зі збереженням запису розкладу зачіпає кілька пов'язаних таблиць одночасно. Окрім стандартних операцій, цей сервіс надає низку методів фільтрованого отримання даних, що є необхідними як для адміністративної підсистеми, так і для публічної.

Усі сервіси разом зі своїми валідаторами реєструються в контейнері залежностей у межах окремого методу розширення, що групує всі реєстрації сервісного шару в одному місці. Такий підхід дозволяє підключити весь сервісний шар до будь-якого з двох систем єдиним викликом, не дублюючи конфігурацію, а при необхідності - легко замінити конкретну реалізацію сервісу на іншу без змін у коді системи.

2.7 Проєктування модуля валідації розкладу

Коректність операцій із розкладом є критичною вимогою до системи, тому модуль валідації розкладу є обов'язковим елементом сервісного шару, що виконується перед кожною операцією збереження або оновлення запису розкладу [3, 10].

Модуль валідації побудовано на основі двох взаємодоповнюючих патернів: стратегії та звітної моделі [41]. Кожне бізнес-правило реалізовано як окремий клас, що відповідає єдиному контракту з одним асинхронним методом виконання. Цей метод отримує об'єкт передачі даних для перевірки та спільний звіт, до якого записує виявлені порушення. Принциповим є те, що правило записує порушення до звіту і повертає без переривання виконання. Це дозволяє валідатору послідовно виконати всі зареєстровані правила за один прохід і повернути повний перелік порушень, а не лише перше знайдене. Адміністратор отримує інформацію про всі проблеми конкретного запису одразу, без

необхідності повторних спроб збереження після усунення кожного порушення окремо.

Базовий валідатор отримує колекцію правил через механізм впровадження залежностей. Реєстрація правил у контейнері залежностей є єдиним місцем, де треба змінити конфігурацію при додаванні нового правила – сам валідатор і сервісний шар при цьому залишаються незмінними. Це забезпечує відповідність принципу відкритості/закритості: система відкрита для розширення новими правилами і закрита для модифікації існуючого коду [18, 41].

Звітна модель містить список записів про порушення, де кожен запис включає назву поля, до якого відноситься порушення, та текстовий опис причини. Після виконання всіх правил валідатор повертає звіт сервісному шару. Якщо звіт містить хоча б одне порушення, сервіс зупиняє виконання та передає структурований перелік помилок контролеру, який формує відповідь клієнту з описом кожного порушення.

Правила валідації розкладу поділяються на два рівні. Перший – узагальнені правила, що перевіряють загальні вимоги, спільні для багатьох сутностей: активність пов'язаних записів на вказану дату, наявність обов'язкових зв'язків. Другий – доменно-специфічні правила, унікальні для предметної області розкладу: вони перевіряють складні умови, що потребують звернення до кількох репозиторіїв одночасно та порівняння даних між різними сутностями системи.

У таблиці 2.2 наведено перелік проєктованих правил валідації розкладу з описом їх предметного змісту та джерел даних, що залучаються при перевірці.

Окремо від правил валідації збереження реалізовано також перевірку конфліктів у режимі редагування розкладу на стороні клієнтської підсистеми. Ця перевірка виявляє ситуації, коли одна аудиторія одночасно відображається у розкладах кількох груп, і надає адміністратору попередження ще до спроби збереження.

Таблиця 2.2 Опис доменно-сцецифічних правил валідації

Правило	**Опис**
Перевірка місткості аудиторії	Сумарна кількість студентів усіх призначених груп не перевищує місткість аудиторії
Конфлікт аудиторії	Аудиторія не зайнята іншим заняттям у той самий часовий слот та дату
Конфлікт викладача	Викладач не проводить інше заняття у той самий часовий слот та дату
Конфлікт студентської групи	Жодна з призначених груп не перебуває на іншому занятті у той самий час
Належність групи до освітньої програми	Усі призначені групи належать до тієї самої освітньої програми, що й рядок навчального плану
Відповідність годин навантаженню	Сумарна кількість запланованих годин викладача за дисципліною в семестрі не перевищує встановлене навантаження
Активність викладача	Викладач є активним співробітником на дату проведення заняття

2.8 Проєктування контракту WebAPI

Прикладний програмний інтерфейс є єдиною точкою взаємодії між клієнтськими підсистемами та сервісним шаром. Від якості його проєктування залежить зручність споживання даних клієнтом, передбачуваність поведінки при помилках та можливість розширення інтерфейсу без порушення сумісності. У даній роботі прикладний програмний інтерфейс побудовано відповідно до архітектурного стилю REST із використанням засобів ASP.NET Core [25, 43].

Маршрутизація та структура ресурсів. Кожен контролер відповідає одному ресурсу предметної області та отримує базову адресу виду api/[назва контролера]. Усі контролери анотовані атрибутами, що позначають їх як

компоненти прикладного програмного інтерфейсу та визначають шаблон маршруту. Тип операції визначається методом запиту:

- GET запити для отримання даних;
- POST запити для створення нових записів;
- PUT запити для оновлення існуючих записів;
- DELETE запити для видалення записів.

Для операцій, що потребують ідентифікатора запису, він передається як частина адреси. Для спеціалізованих операцій фільтрації використовуються додаткові сегменти адреси, що однозначно описують критерій вибірки: наприклад, отримання розкладу за викладачем або за часовим слотом реалізовано через окремі маршрути з відповідним суфіксом [43].

Усі методи контролерів повертають типізований результат, що дозволяє повертати як успішну відповідь з об'єктом, так і HTTP-код помилки з описом причини. Усі методи контролерів обгортають виклики сервісів у блоки обробки виключень: виключення валідації та відсутнього запису обробляються явно з конкретними кодами стану, решта – з кодом 400 та журналюванням деталей помилки [25, 31].

Контролер розкладу є найбагатшим за складом методів, оскільки розклад є центральною сутністю системи і потребує різноманітних способів вибірки. Окрім стандартних операцій читання та зміни, він надає низку інших потрібних операцій пошуку. Для адміністративного інтерфейсу передбачено окремий метод отримання розкладу у форматі редактора: він повертає зведені дані по кафедрі та семестру, що є необхідним для побудови табличного представлення розкладу з можливістю редагування. Також реалізовано метод обчислення конкретних дат занять на основі параметрів семестру та типу тижня, та метод отримання і видалення контекстів розкладу – логічних одиниць, що об'єднують розклад конкретної групи у конкретному семестрі [10, 28].

Оскільки адміністративна та публічна клієнтські підсистеми є окремими системами, що можуть бути розгорнуті на різних портах, прикладний програмний інтерфейс налаштовує політику міждоменного доступу, що дозволяє

запити з адрес клієнтських систем. Це є необхідною умовою коректної роботи браузерних клієнтів, оскільки браузер блокує запити між різними джерелами за замовчуванням [24].

Висновок до розділу 2

У другому розділі сформульовано постановку задачі, визначено вимоги до системи та виконано проєктування всіх компонентів, що є предметом даної роботи.

Функціональні вимоги систематизовано окремо для адміністративної та публічної підсистем. Єдине мовне середовище C#/.NET усуває клас помилок невідповідності між серверною та клієнтською моделями. Архітектурне рішення про дві ізольовані підсистеми реалізує розмежування ролей на рівні структури рішення.

Проєктування сервісного шару визначило ієрархію сервісів на основі узагальненого базового класу, що усуває дублювання коду. Модуль валідації на основі незалежних правил зі звітною моделлю гарантує повний перелік порушень за один прохід та відкритість для розширення. Контракт прикладного програмного інтерфейсу визначив маршрути для обох підсистем з розмежуванням відкритих і захищених адрес на рівні самого інтерфейсу.

РОЗДІЛ 3. ПРАКТИЧНА РЕАЛІЗАЦІЯ ВЕБ ІНТЕРФЕЙСУ СИСТЕМИ ДЛЯ КЕРУВАННЯ РОЗКЛАДОМ

3.1 Конфігурація системи

Точкою входу адміністративної підсистеми є файл Program.cs, де послідовно реєструються всі компоненти системи. Рядок підключення до бази даних зчитується з конфігурації та передається методу розширення шару доступу до даних AddDalServices, що реєструє контекст бази даних з автоматичним визначенням версії сервера MySQL та одиницю роботи з часом життя в межах запиту (ліст. 3.1). Після цього одним викликом AddBllServices підключається весь сервісний шар. Такий підхід дозволяє підключити обидва шари до будь-якого з систем без дублювання конфігурації.

Лістинг 3.1 Метод розширення AddDalServices

```
1: public static IServiceCollection AddDalServices(
2:     this IServiceCollection services, string
   connectionString) {
3:     services.AddDbContext<ScheduleDbContext>(options =>
4:         options.UseMySql(connectionString,
5:             ServerVersion.AutoDetect(connectionString)));
6:     services.AddScoped<IUnitOfWork, UnitOfWork>();
7:     return services; }
```

Метод AddBllServices організовує реєстрацію сервісів та їх валідаторів за двома категоріями (ліст. 3.2). Для простих довідникових сутностей, де предметних правил перевірки немає, валідатор ініціалізується з порожньою колекцією правил – це зберігає єдиний механізм для всіх сервісів без умовної логіки та спеціальних розгалужень. Для сутностей зі складними правилами валідатор створюється через статичний фабричний метод самого сервісу, що отримує одиницю роботи через контейнер залежностей. Такий підхід зосереджує знання про потрібні правила безпосередньо в класі сервісу.

Усі реєстрації виконуються з часом життя в межах запиту (Scoped), що гарантує ізоляцію стану між різними сеансами користувачів у Blazor Server.

Перелік охоплює маппер, понад двадцять довідникових сервісів, сервіси персон та викладачів, навчальних планів та освітніх програм, часових слотів, навантаження та розкладу. Публічна підсистема підключає той самий метод AddBllServices без жодних змін – сервісний шар є спільним для обох систем.

Лістинг 3.2 Патерни реєстрації у методі розширення BLL

```
// Прості довідники — порожній валідатор
services.AddScoped<IValidator<AcademicDegreeDto>>(
    _ => new BaseValidator<AcademicDegreeDto>([]));
services.AddScoped<IAcademicDegreeService,
  AcademicDegreeService>();
// Сутності зі складними правилами — фабричний метод
services.AddScoped<IValidator<ScheduleDto>>(sp =>
    ScheduleService.CreateValidator(
        sp.GetRequiredService<IUnitOfWork>()));
services.AddScoped<IScheduleService, ScheduleService>();
```

Автентифікація налаштовується на основі cookie-схеми: при зверненні до захищеного маршруту без активного сеансу система перенаправляє на сторінку входу, визначену у LoginPath. Для підтримки каскадного стану автентифікації у Blazor-компонентах реєструються стани.

HTTP-клієнт налаштовується з базовою адресою, що визначається динамічно на основі поточного контексту запиту: якщо контекст наявний – береться схема та хост реального запиту, інакше застосовується адреса за замовчуванням. Це забезпечує коректну роботу як у середовищі розробки, так і на виробничому сервері без зміни конфігурації.

У конвеєрі обробки запитів порядок реєстрації є принциповим: статичні файли підключаються першими, після чого – автентифікація, авторизація та захист від підроблення запитів. Завершують конвеєр маршрути контролерів і маршрути Razor-компонентів із серверним режимом інтерактивності.

3.2 Реалізація DTO

Об'єкти передачі даних підключаються до всіх проєктів – будь-яка невідповідність між серверною і клієнтською моделями виявляється на етапі

компіляції. Усі об'єкти реалізовані як записи C#, що забезпечує незмінність після ініціалізації [39]. Ієрархія охоплює п'ять категорій: базова модель для списків, об'єкти довідників, предметних сутностей, персональних даних та розкладу.

Базова параметризована модель приймає структурний тип ідентифікатора та назву (ліст. 3.3). Кожен довідниковий об'єкт містить обов'язкові поля назви з ключовим словом required та метод ToLookup() для перетворення до базової моделі без зовнішнього коду (ліст. 3.4). Для сутностей зі зв'язками об'єкт містить як ідентифікатори пов'язаних сутностей, так і їх денормалізовані назви – це дозволяє відображати дані у таблицях без додаткових запитів.

Лістинг 3.3 Параметризований запис

```
1: public record LookupModel<TId>(TId Id, string Name)
2:   where TId : struct, IEquatable<TId>;
```

Кожен довідниковий об'єкт містить ідентифікатор, обов'язкові поля назви та метод перетворення до базової моделі. Наявність методу безпосередньо в об'єкті передачі даних дозволяє сервісам і компонентам виконувати перетворення в один виклик без зовнішнього коду відображення. Лістинг 3.4 демонструє цей підхід на прикладі академічного ступеня.

Лістинг 3.4 DTO довідникового об’єкта

```
1: public record AcademicDegreeDto{
2: public ulong Id { get; init; }
3: public required string AcademicDegreeName { get; init; }
4: public required string AcademicDegreeShortName { get;
   init; }
5: public LookupModel<ulong> ToLookup() => new(Id,
   AcademicDegreeName);}
```

Ключове слово required гарантує, що поля назви обов'язково будуть ініціалізовані при створенні об'єкта – компілятор відхилить код, де ці поля відсутні [40]. Це усуває цілий клас помилок, де об'єкт передачі даних міг бути створений із порожніми обов'язковими полями.

Перетворення між сутностями та DTO виконується через AutoMapper з власним адаптером, що ізолює сервісний шар від конкретної бібліотеки. Для

простих сутностей застосовується симетричне відображення одним рядком (ліст. 3.5), для складніших – явне відображення навігаційних властивостей (ліст. 3.6).

Лістинг 3.5 Маппінг довідникової сутності

```
CreateMap<AcademicDegree,
AcademicDegreeDto>().ReverseMap();
```

Лістинг 3.6 Маппінг сутності з навігаційними властивостями

```
CreateMap<Classroom, ClassroomDto>()
   .ForMember(dest => dest.ClassroomTypeName,
opt => opt.MapFrom(src =>
src.ClassroomType.ClassroomTypeName))
   .ForMember(dest => dest.DepartmentName,
opt => opt.MapFrom(src => src.Department.ShortName));
```

3.3 Реалізація сервісного шару

Центральним елементом сервісного шару є параметризований базовий клас із двома параметрами типу – типом об'єкта передачі даних та типом доменної сутності (рис. 3.1). Він реалізує стандартний інтерфейс сервісу і слугує базою для всіх конкретних сервісів системи. Такий підхід усуває дублювання типових операцій: замість того щоб писати однаковий код отримання, збереження та видалення для кожного з більш ніж двадцяти довідників, ця логіка реалізована один раз у базовому класі [5].

Базовий клас отримує через конструктор чотири залежності: одиницю роботи для доступу до репозиторіїв, реєстратор подій, маппер для перетворення між типами та валідатор. Усі чотири зберігаються у захищених полях, доступних підкласам.

Алгоритм збереження нового запису є фіксованим: виклик валідатора – перетворення DTO у сутність – передача до репозиторію – фіксація транзакції – повернення DTO. Операція оновлення розширює цей алгоритм отриманням існуючого запису, видалення делегує репозиторію та фіксує транзакцію, операції читання повертають результати через маппер.

Точками розширення для підкласів є захищена властивість репозиторію та три захищені віртуальні методи перетворення між сутністю і DTO. За замовчуванням методи делегують мапперу – підклас перевизначає лише ті, де є специфічна логіка, наприклад обробка зв'язків «багато до багатьох».

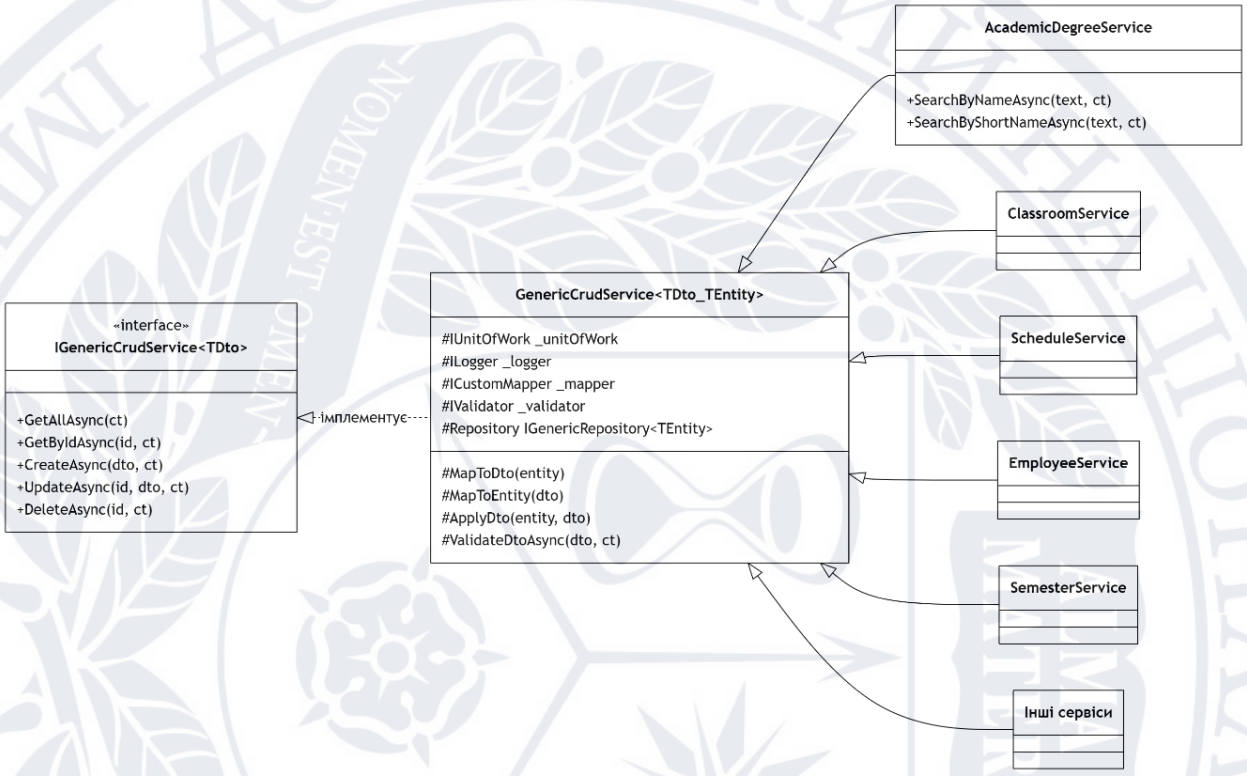


Рисунок 3.1 – Ієрархія GenericCrudService

Для простих довідникових сутностей підклас визначає лише властивість репозиторію – решта успадковується. Лістинг 3.7 демонструє повну реалізацію сервісу академічних ступенів.

Лістинг 3.7 Конструктор академічних ступенів.

```
1: public sealed class AcademicDegreeService
2:     : GenericCrudService<AcademicDegreeDto,
  AcademicDegree>, IAcademicDegreeService {
3:     public AcademicDegreeService(IUnitOfWork unitOfWork,
4:         ILogger<AcademicDegreeService> logger,
5:         ICustomMapper mapper,
6:         IValidator<AcademicDegreeDto> validator)
7:         : base(unitOfWork, logger, mapper, validator) { }
8:     protected override IGenericRepository<AcademicDegree>
  Repository
9:         => _unitOfWork.AcademicDegreeRepository; }
```

Довідникові сервіси, що потребують пошуку, додають власні методи поверх успадкованих базових операцій. Такий метод отримує текстовий критерій і передає критерій до спеціалізованого методу репозиторію, що виконує пошук на рівні бази даних. Результат перетворюється на об'єкти передачі даних через успадкований маппер. Прикладом є сервіс академічних ступенів, що реалізує пошук як за повною назвою, так і за скороченням – кожен через окремий метод репозиторію (ліст. 3.8).

Сервіси, для яких пошук не є актуальним, обмежуються лише визначенням властивості репозиторію і більше нічого до класу не додають. Таким чином, обсяг конкретного довідникового сервісу прямо залежить від складності предметної сутності: від одного рядка визначення репозиторію до кількох методів пошуку та фільтрації [6].

Лістинг 3.8 Інтерфейс сервісу академічних ступенів.

```
1: public interface IAcademicDegreeService :
   IGenericCrudService<AcademicDegreeDto>{
2:       Task<IEnumerable<AcademicDegreeDto>>
   SearchByNameAsync(
3:       string text, CancellationToken ct = default);
4:       Task<IEnumerable<AcademicDegreeDto>>
   SearchByShortNameAsync(
5:       string text, CancellationToken ct = default);}
```

3.4 Реалізація модуля валідації

В додатку А зображено структуру модуля валідації. Система реалізує два паралельних шляхи перевірки: CRUD-валідацію одиничного запису через BaseValidator<T> та пакетну валідацію сітки редактора через ScheduleEditorValidator. Обидва шляхи ініціюються з сервісом розкладу, але мають принципово різний механізм роботи [41].

Модуль валідації складається з чотирьох незалежних елементів: запису результату, звітної моделі, контракту правила та базового валідатора.

Запис результату – незмінна структура з двома полями: назвою поля, до якого відноситься порушення, та текстовим описом причини (ліст. 3.9). Звітна

модель накопичує такі записи у списку та надає властивість, що вказує на відсутність помилок, і метод серіалізації всього переліку в один рядок для передачі клієнту.

Лістинг 3.9 Запис результату.

```
1: public class ValidationReport {
2: public void Add(string property, string message)
    => Errors.Add(new ValidationResult(property, message));
```

Контракт правила визначає єдиний асинхронний метод, що отримує об'єкт передачі даних та спільний звіт [39]. Правило записує знайдені порушення у звіт і повертає управління без переривання. Базовий валідатор отримує колекцію правил через конструктор і послідовно виконує кожне, збираючи результати у спільний звіт. Завдяки цьому після одного виклику клієнт отримує повний перелік усіх порушень одразу.

3.4.1 Узагальнені правила

Система містить набір параметризованих правил без прив'язки до конкретної предметної галузі. Правило заборони циклу в ієрархії обходить ланцюжок батьківських посилань та перевіряє, чи не зустрічається поточний ідентифікатор серед предків. Правило відповідності діапазону дат перевіряє, що дата початку строго передує даті завершення. Правило невід'ємного числового поля перевіряє, що значення не є від'ємним. Усі три є параметризованими – конкретні поля передаються як лямбда-вирази при створенні правила, що дозволяє застосовувати їх для різних сутностей без дублювання коду.

3.4.2 Доменно-специфічні правила розкладу

Правила розкладу реалізують IValidationRule<ScheduleDto> і виконуються базовим валідатором при кожній операції збереження або оновлення одиничного запису. Лістинг 3.10 показує правило конфлікту аудиторії: один запит до репозиторію по часовому слоту, фільтрація по даті та ідентифікатору запису,

перевірка збігу аудиторії. Аналогічно влаштовані і інші правила. Кожне доменно-специфічне правило отримує IUnitOfWork через конструктор і звертається до репозиторіїв самостійно, без координації з іншими правилами.

Лістинг 3.10 Правило конфлікту аудиторій.

```
1: public class ClassroomConflictRule :
   IValidationRule<ScheduleDto>{
2: private readonly IUnitOfWork _uow;
3: public ClassroomConflictRule(IUnitOfWork uow) => _uow =
   uow;
4: public async Task ExecuteAsync(ScheduleDto data,
5: ValidationReport report, CancellationToken ct =
   default){
6: var sameSlot = (await _uow.ScheduleRepository
7:       .GetByTimeSlotAsync(data.TimeSlotId))
8:       .Where(s => s.ClassDate == data.ClassDate
9:             && s.ScheduleId != data.Id).ToList();
10:    if (sameSlot.Any(s => s.ClassroomId ==
   data.ClassroomId))
     a. report.Add("ClassroomId",
11:    "Аудиторія вже зайнята іншим заняттям у цей час");}}
```

3.4.3 Пакетна валідація

Адміністративний редактор розкладу оперує одночасно розкладами всіх груп кафедри за семестр – словником, де ключем є ідентифікатор групи, а значенням – список записів її розкладу. Перевіряти кожен запис окремо через BaseValidator у цьому режимі означало б по N запитів до бази даних на кожне правило. Тому реалізовано окремий ScheduleEditorValidator, що приймає всю сітку одразу і виконує перевірки агрегацією в пам'яті.

Механізм створення ScheduleEditorValidator принципово відрізняється від BaseValidator<T>. Він отримує через конструктор сервіс навантаження і сам перевіряє виконання правил (ліст. 3.11).

Два синхронних правила – конфлікту викладача та конфлікту аудиторії – реалізують звичайний метод Check, а не ExecuteAsync. Вони отримують список усіх комірок сітки та словник розкладів, групують комірки по ключу і повертають рядки помилок для всіх груп, де одночасно виявлено більше одного

унікального ідентифікатора групи. Такий підхід виявляє всі конфлікти за один прохід без жодного звернення до бази даних.

Лістинг 3.11 Пакет правил валідації

```
1: public sealed class ScheduleEditorValidator {
2: private readonly EditorTeacherConflictRule
   _teacherConflict   = new();
3: private readonly EditorClassroomConflictRule
   _classroomConflict = new();
4: private readonly EditorWorkloadExceededRule
   _workloadRule;
5: public ScheduleEditorValidator(ITeacherWorkloadService
   workloadSvc)
        => _workloadRule = new(workloadSvc); }
```

Асинхронне правило перевищення навантаження отримує список усіх залучених викладачів, для кожного завантажує дані навантаження через сервіс і порівнює заплановані години з кількістю комірок у сітці – кожна комірка рахується як два академічні часи.

Метод ValidateAsync послідовно збирає результати всіх трьох правил у спільний список рядків і повертає його як IReadOnlyList<string>. Це принципова відмінність від BaseValidator<T>, де результатом є ValidationReport з прив'язкою до конкретного поля: пакетний валідатор повертає плоский перелік текстових повідомлень для відображення у зведеному блоці редактора, оскільки помилки тут стосуються взаємодії між записами різних груп, а не окремого поля конкретного об'єкта.

3.5 Реалізація контролерів

Контролери є шаром між клієнтом і сервісним шаром: вони приймають HTTP-запити, делегують виконання відповідному сервісу та формують стандартизовані відповіді. Повний ланцюжок обробки запиту зображено на рис. 3.5: Blazor UI надсилає запит контролеру, той викликає сервіс, сервіс отримує репозиторій з одиниці роботи, репозиторій виконує запит до MySQL через EF Core і результат повертається у зворотному напрямку у форматі JSON [30, 43].

Кожен контролер довідника позначається двома атрибутами: атрибутом API-контролера, що вмикає автоматичну валідацію моделі та прив'язку параметрів, та атрибутом маршруту у форматі api/[controller] – назва контролера автоматично стає частиною адреси [25]. Контролер отримує через конструктор реєстратор подій та відповідний сервіс.

Лістинг 3.12 показує структуру контролера довідника з прикладом базової операціями та маршрутом пошуку.

Лістинг 3.12 Структура контролера довідника

```
1: [ApiController]
2: [Route("api/[controller]")]
3: public class AcademicDegreeController : ControllerBase{
4:     [HttpGet]
5:     public async Task<IEnumerable<AcademicDegreeDto>>
   GetAll(CancellationToken ct)
```

На рисунку 3.2 зображено потік обробки одного HTTP-запиту. Операція отримання всіх записів є однорядковою і делегує сервісу без додаткової обробки. Операція отримання за ідентифікатором перевіряє результат на порожнє значення та повертає 404 з описом, якщо запис не знайдено.

Операції зміни даних обгорнуті у блоки перехоплення виключень з трьома рівнями обробки:. Операція створення при успіху повертає 201 з посиланням на новостворений ресурс. Операції оновлення та видалення при успіху повертають 204 без тіла. Операція оновлення додатково перевіряє збіг ідентифікатора у URL та тілі запиту перед зверненням до сервісу.

Контролер розкладу є найрозгалуженішим у системі – він надає як базові CRUD-операції, так і окремі групи спеціалізованих маршрутів для адміністративної та публічної підсистем.

Базові операції мають ідентичну до довідника структуру з тією відмінністю, що операція створення виконує CRUD-валідацію через BaseValidator перед збереженням – виключення валідації перехоплюється і повертається клієнту з кодом 400 та текстом звіту порушень.

Спеціалізовані маршрути фільтрованого отримання даних є однорядковими делегатами до відповідних методів сервісу, де параметр фільтрації передається як частина адреси. Реалізовано фільтрацію за датою заняття, часовим слотом, рядком навчального плану, викладачем та аудиторією. Лістинг 3.13 показує типову структуру такого маршруту.

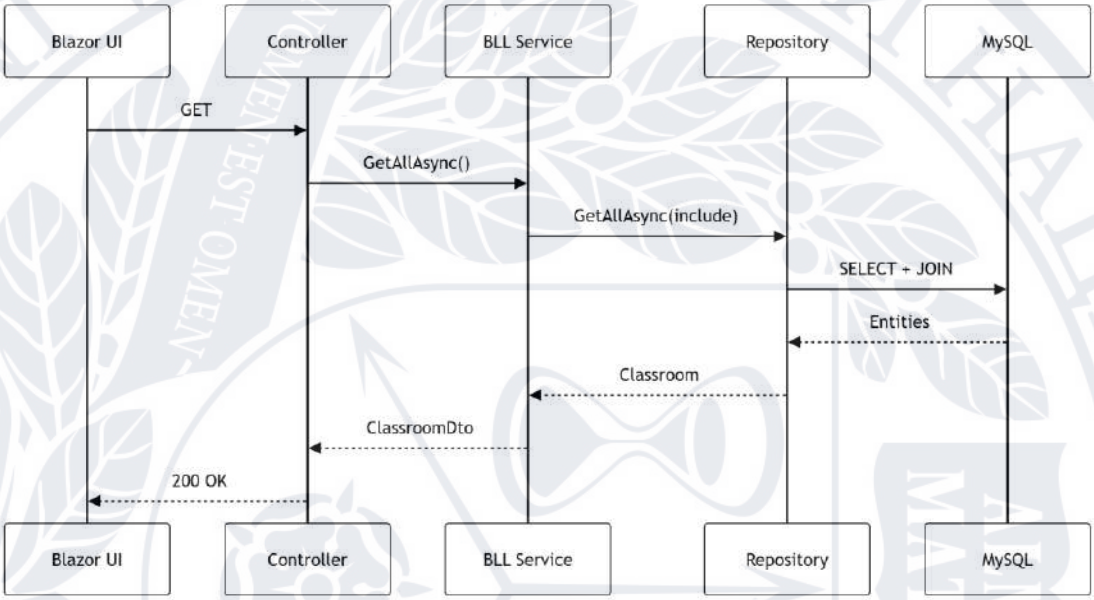


Рисунок 3.2 – Потік обробки HTTP-запиту

Лістинг 3.13 Фільтрація

```
1: [HttpGet("by-employee/{employeeId}")]
2: public async Task<IEnumerable<ScheduleDto>>
   GetByEmployee(
3: ulong employeeId, CancellationToken ct = default)
       => await _service.GetByEmployeeAsync(employeeId,
       ct);
```

Маршрут calc-dates приймає у тілі запиту параметри генерації – день тижня, тип тижня, дату початку та кількість тижнів – і повертає перелік конкретних календарних дат занять. Це дозволяє клієнту одним запитом отримати всі дати для цілого семестру без власних обчислень.

Група маршрутів редактора обслуговує виключно адміністративну підсистему. Маршрут validate/{semesterId} приймає у тілі словник groupId – список комірок і повертає перелік рядків із описом виявлених конфліктів без збереження – це дозволяє адміністратору перевірити сітку перед фіксацією змін.

Збереження окремої комірки виконується через editor/save-cell з повним об'єктом комірки у тілі запиту. Видалення комірки отримує чотири параметри як частини адреси – ідентифікатори групи, семестру, числове значення дня тижня та ідентифікатор часового слоту.

Публічна підсистема обслуговується окремим контролером ScheduleViewController, що також позначений атрибутами API-контролера та автоматичної маршрутизації. Він використовує той самий IScheduleService, що й адміністративна підсистема, але надає виключно маршрути читання – без жодних операцій зміни даних.

Контролер реалізує ієрархічну навігацію для режиму студента: спочатку клієнт отримує список кафедр, потім за ідентифікатором кафедри – список спеціальностей, далі за кафедрою та спеціальністю – список батьківських груп, і нарешті за батьківською групою – список підгруп. Кожен маршрут повертає базову модель з ідентифікатором та назвою, що мінімізує обсяг переданих даних. Лістинг 3.14 ілюструє цю каскадну структуру. Для режиму викладача аналогічно реалізовано маршрут отримання викладачів кафедри.

Механізм автентифікації зображено в додатку Б. При запиті до захищеної сторінки система перевіряє наявність активного сеансу. Якщо сеансу немає – користувач перенаправляється на сторінку входу, де вводить логін і пароль. Форма надсилає дані методом POST на маршрут контролера автентифікації.

Контролер автентифікації успадковує базовий клас MVC-контролера і доступний без авторизації. При вході він отримує обліковий запис з репозиторію за логіном: якщо запис не знайдено або пароль не збігається – перенаправляє на сторінку входу з параметром помилки; якщо дані коректні – формує масив заяв з логіном та ідентифікатором, створює ідентифікаційний об'єкт за схемою cookie-автентифікації та підписує користувача.

Лістинг 3.14 Демонстрація маршруту

```
1: [HttpGet("departments")]
2: public async Task<IEnumerable<LookupModel<ulong>>>
   GetDepartments(CancellationToken ct)
        => await _service.GetDepartmentLookupsAsync(ct);
```

Вихід з системи реалізовано окремим маршрутом – метод викликає SignOutAsync для видалення cookie-сеансу і перенаправляє на сторінку входу. Після успішного входу користувач отримує доступ до всіх сторінок та API-маршрутів адміністративної підсистеми, захищених атрибутом авторизації. Маршрути, захищені атрибутом авторизації, автоматично перенаправляють неавтентифікованого користувача на сторінку входу без додаткової логіки в компонентах. Це забезпечує єдину точку контролю доступу на рівні middleware, а не на рівні окремих сторінок.

3.6 Реалізація клієнтської частини UI-Creator (адміністратор)

Адміністративна підсистема побудована на основі ієрархії базових компонентів, що централізують повторюваний функціонал і надають конкретним сторінкам лише точки розширення. логіки для кожної сторінки вона реалізована один раз у базовому класі.

Базовий клас сторінки довідника є параметризованим і приймає два параметри типу – тип DTO та тип ідентифікатора. Він успадковує стандартний базовий клас Blazor-компонента і надає захищені властивості та методи, що охоплюють повний цикл роботи з даними.

Ключовим елементом базового класу є захищений метод виконання асинхронної операції. Він приймає делегат і рядок повідомлення про помилку, встановлює прапорець завантаження, виконує делегат у блоці перехоплення та у будь-якому випадку знімає прапорець завантаження. Завдяки цьому кожна конкретна сторінка загортає будь-який HTTP-запит в один виклик без дублювання логіки стану (ліст 3.15).

Окрім базового класу сторінки реалізовано набір спільних компонентів інтерфейсу: кнопки дій у рядку таблиці з іконками редагування та видалення, фільтр колонки таблиці з випадаючим списком унікальних значень, кнопка очищення фільтрів, компонент посторінкового виведення та кнопка додавання запису. Кожен з них є самодостатнім компонентом з вхідними параметрами і

подіями, що дозволяє вбудовувати їх у будь-яку сторінку без дублювання розмітки.

Кожна сторінка довідника успадковує базовий клас та реалізує конкретну логіку роботи з даними. Файл розмітки визначає маршрут сторінки, успадкування та розмітку таблиці й модальних вікон. Файл коду визначає адресу API, поля форми та методи завантаження, збереження і видалення.

Лістинг 3.15 Метод виконання асинхронної операції

```
protected async Task ExecuteAsync(
   Func<Task> action, string errorMessage){
 try {
       IsLoading = true;
       await action(); }
  catch (Exception ex) {
       ErrorMessage = $"{errorMessage}. {ex.Message}";
       Logger.LogError(ex, errorMessage); }
  finally { IsLoading = false; }}
```

Алгоритм збереження запису починається з очищення попередніх помилок та клієнтської перевірки обов'язкових полів. Якщо перевірка не пройдена – встановлюється ValidationError і виконання зупиняється без звернення до сервера. При успішній перевірці формується DTO і надсилається або POST-запит для нового запису, або PUT-запит для існуючого (ліст. 3.16).

Модальне вікно редагування містить форму з полями, прив'язаними через @bind до полів компонента. Кнопка збереження викликає метод SaveItem. Модальне вікно підтвердження видалення відображає назву запису та дві кнопки дії, повідомлення про помилку видалення базовий клас встановлює у ErrorMessage – воно відображається над таблицею.

Сторінка редактора розкладу є найскладнішим компонентом адміністративної підсистеми. Вона складається з двох рівнів: сторінки-менеджера контекстів та сітки редактора для конкретного контексту.

Сторінка-менеджер відображає таблицю наявних контекстів та підтримує фільтрацію по кожній колонці через компонент фільтра, що формує випадаючий список унікальних значень з поточних даних.

Створення нового контексту відбувається через модальне вікно з каскадними списками: спочатку обирається факультет або кафедра, після чого підвантажуються батьківські групи. При виборі групи автоматично підвантажуються доступні семестри. Після вибору семестру і типу тижня та натискання «Далі» система обчислює дати занять через маршрут calc-dates і відкриває сітку редактора для щойно створеного контексту.

Лістинг 3.16 Логіка збереження запису довідника

```
HttpResponseMessage response = IsEdit
  ? await Http.PutAsJsonAsync($"{ApiUrl}/{dto.Id}", dto)
  : await Http.PostAsJsonAsync(ApiUrl, dto);
  if (!response.IsSuccessStatusCode){
      ValidationError =
          $"Помилка від сервера: {await
response.Content.ReadAsStringAsync()}";
      return;}
CloseEditDialog();
await LoadDataAsync();
```

Сітка редактора відображає тижневий розклад у форматі матриці: рядки – часові слоти, стовпці – підгрупи батьківської групи (рис. 3.3). Кожна комірка відображає поточне заняття або порожній стан. Заповнення комірки відбувається через модальне вікно, де адміністратор обирає дисципліну, викладача та аудиторію з відфільтрованих списків. При збереженні комірки надсилається запит до editor/save-cell – якщо сервер повертає помилку валідації, вона відображається у тому самому вікні.

Сторінка входу використовує порожній макет без бокової панелі та навігації. Вона позначена атрибутом дозволу анонімного доступу, тому захищений middleware пропускає її без перевірки сеансу. При ініціалізації компонент перевіряє поточний стан автентифікації – якщо користувач вже автентифікований, одразу перенаправляє на сторінку розкладів.

Форма входу є звичайною HTML-формою з методом POST, що надсилає дані безпосередньо на маршрут /account/login контролера автентифікації – без JavaScript і без Blazor-інтерактивності (ліст. 3.17). Це свідоме архітектурне рішення: стандартна HTML-форма гарантує коректну роботу навіть при

відсутності активного SignalR-з'єднання, яке Blazor Server потребує для обробки подій.

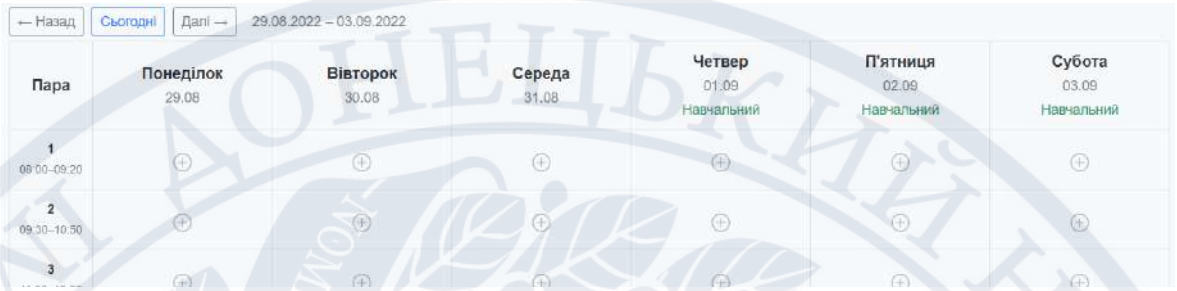


Рисунок 3.3 – Сітка розкладу

Після успішного входу усі сторінки та API-маршрути адміністративної підсистеми, захищені атрибутом авторизації, стають доступними після успішної автентифікації і недоступними після виходу.

Лістинг 3.17 Розмітка форми автентифікації користувача

```
1: <form method="post" action="/account/login">
2:    <input id="login"  name="login"  class="form-control"
   required />
3:    <input id="password" name="password" type="password"
      a. class="form-control" required />
4:    <button type="submit" class="btn btn-
   primary">Увійти</button>
5: </form>
```

3.7 Реалізація компонента UI-Watcher (публічний доступ)

Публічна підсистема є самостійною Blazor Server системою із власною точкою входу, що підключає той самий сервісний шар через метод розширення BLL без змін.

Підсистема повністю відкрита без автентифікації – розклад є публічною інформацією, тому жоден компонент або маршрут не позначений атрибутом авторизації. Операції зміни даних відсутні повністю на обох рівнях.

При ініціалізації компонент завантажує список кафедр, активних семестрів та часових слотів. Після завантаження система автоматично визначає поточний

семестр за датою і встановлює його вибраним за замовчуванням, або обирає перший зі списку, якщо поточна дата не потрапляє в жоден семестр.

Компонент перегляду розкладу підтримує два режими роботи. Стан режиму зберігається у перелічуваному типі. Перемикання режиму скидає всі обрані фільтри та завантажені і ховає поточну сітку розкладу. Це гарантує, що при переключенні користувач починає з чистого стану без артефактів від попереднього режиму.

Завантаження розкладу виконується однією кнопкою, що стає активною лише після заповнення обов'язкових фільтрів. Лістинг 3.18 показує логіку визначення ідентифікатора для запиту.

Лістинг 3.18 Логіка визначення ідентифікатора

```
1: if (_mode == ViewMode.Group){
2:    var groupId = _subgroupId > 0 ? _subgroupId :
  _parentGroupId;
3:    _currentSchedule = await
  Http.GetFromJsonAsync<List<ScheduleItemDto>>(
4:       $"{ApiBase}/group/{groupId}/{_semesterId}") ??
  new(); }
5: else {
6:    _currentSchedule = await
  Http.GetFromJsonAsync<List<ScheduleItemDto>>(
7:     $"{ApiBase}/teacher/{_teacherId}/{_semesterId}") ??
  new(); }
```

Після завантаження сітка відображає тижневий вигляд: рядки – часові слоти з номером та часом початку і завершення, стовпці – шість робочих днів тижня. Кожна комірка відображає дисципліну, тип заняття та залежно від режиму – або викладача з групами (режим студента), або аудиторію з групами (режим викладача).

Після завантаження розкладу компонент визначає початковий тиждень для відображення. Алгоритм перевіряє, чи потрапляє поточна дата в межі обраного семестру. Якщо так – встановлює початок поточного тижня. Якщо ні – встановлює початок першого тижня семестру. Початком тижня завжди є понеділок, що обчислюється через залишок від ділення (ліст. 3.19).

Лістинг 3.19 Алгоритм обчислення початкової дати поточного тижня

```
var today = DateOnly.FromDateTime(DateTime.Today);
_weekStart = today >= sem.StartDate && today <=
sem.EndDate
   ? GetMonday(today)
   : GetMonday(sem.StartDate);
private static DateOnly GetMonday(DateOnly date) {
   int diff = ((int)date.DayOfWeek -
(int)DayOfWeek.Monday + 7) % 7;
   return date.AddDays(-diff); }
```

Для мобільних пристроїв реалізовано окремий механізм навігації по днях: на вузьких екранах таблиця відображає лише два сусідніх стовпці одночасно. При ініціалізації мобільний навігатор встановлює активним стовпець поточного дня – або перший стовпець, якщо поточний день виходить за межі тижня.

У десктопній версії Фільтри розміщені у лівій бічній панелі та організовані у вигляді пронумерованих кроків. Такий підхід одразу показує користувачу порядок заповнення без додаткових пояснень – кожен крок активується лише після заповнення попереднього (ліст 3.20).

Лістинг 3.20 Логіка каскадного оновлення залежних фільтрів

```
private async Task OnDepartmentChangedGroup() {
   _specialtyId = 0; _parentGroupId = 0; _subgroupId =
0;
   _specialties = new(); _parentGroups = new();
_subgroups = new();
   if (_departmentId == 0) return;
   _specialties = await
Http.GetFromJsonAsync<List<LookupModel<ulong>>>(
      $"{ApiBase}/specialties/{_departmentId}") ??
new(); }
```

У режимі студента фільтри утворюють чотирирівневу ієрархію. На першому рівні обирається факультет або кафедра – одразу після вибору надсилається запит і підвантажується список спеціальностей. До цього моменту другий рівень неактивний. Після вибору спеціальності підвантажуються уже інші рівні.

Скидання нижчих рівнів при зміні вищого є принциповою деталлю: якщо користувач змінює факультет, всі нижчі рівні очищуються і їхні списки також.

Це унеможливлює ситуацію, коли у списку груп залишаються дані від попереднього факультету.

У режимі викладача ієрархія є дворівневою: факультет або кафедра – викладач. При зміні кафедри список викладачів очищається і підвантажується заново. Семестр є спільним для обох режимів і відображається окремим блоком під фільтрами.

3.8 Огляд готової системи

Форма містить два поля – логін та пароль – і кнопку «Увійти» (рис. 3.4). Сторінка використовує порожній макет без навігаційної панелі, що унеможливлює доступ до захищених розділів до завершення автентифікації.

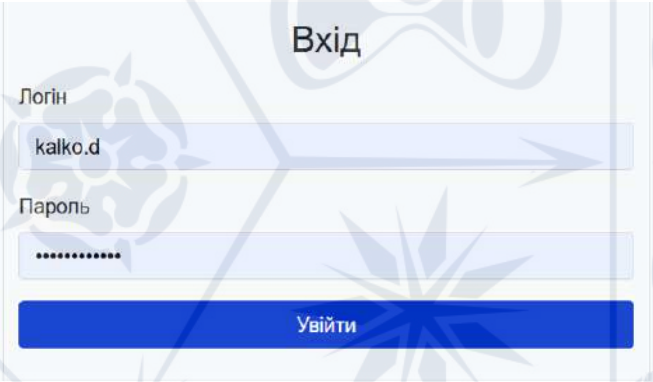


Рисунок 3.4 – Сторінка входу до адміністративної підсистеми

Повідомлення «Невірний логін або пароль» з'являється між полями форми та кнопкою (рис. 3.5). Контролер автентифікації при невідповідності облікових даних перенаправляє на ту саму сторінку з параметром ?error=1 в URL, компонент зчитує цей параметр при ініціалізації та умовно відображає повідомлення.

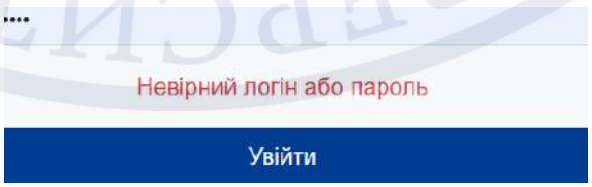


Рисунок 3.5 – Відображення помилки при введенні невірних облікових даних

Розділи згруповано за трьома предметними областями: «Розклад занять» є прямим посиланням на редактор розкладу; «Навчальний процес», «Люди та групи». Поточний активний розділ підсвічено жовтим кольором – на рисунку 3.6.а активно «Спеціальності». Групи розділів позначені заголовками у верхньому регістрі з жовтим кольором, що створює чітку візуальну ієрархію між категорією та її елементами. (рис. 3.6)

а) б)

Рисунок 3.6 – Бічна навігаційна панель адміністративної підсистеми

Таблиця будь-якої довідникової сторінки містить фільтри у заголовку кожного стовпця – поля «Фільтр» дозволяють фільтрувати записи в реальному часі без перезавантаження сторінки: при введенні тексту Blazor миттєво перераховує відповідні рядки (рис. 3.7). Відображаються всі ключові атрибути сутності: код галузі, галузь знань, код спеціальності, повна назва, скорочення та дати початку і завершення дії запису. Кнопки дій у останній колонці – синя для редагування та червона для видалення – присутні в кожному рядку і мають

контрастні кольори для зменшення ризику помилкового натискання. Кнопка «+ Додати» розміщена у заголовку сторінки поряд з назвою, що відповідає стандартному розташуванню первинної дії.

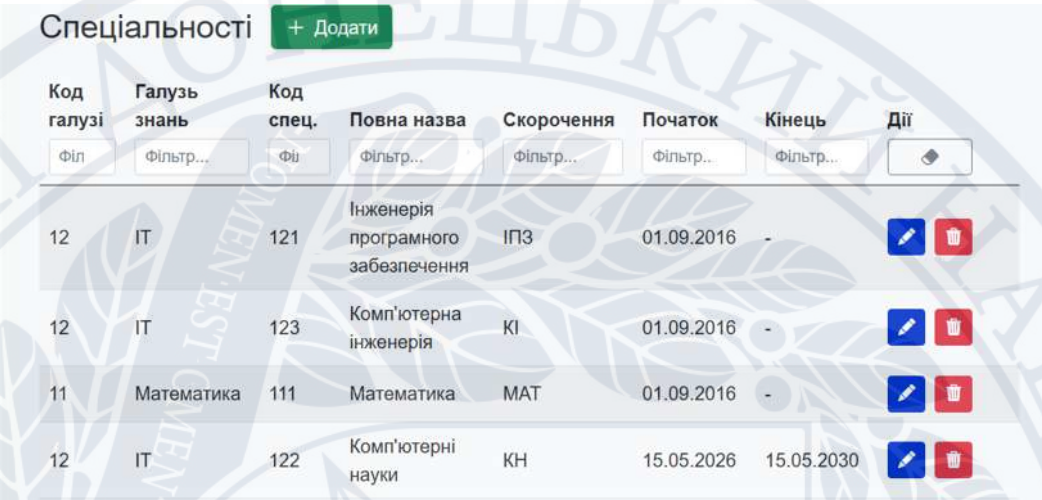


Рисунок 3.7 – Приклад типової CRUD-сторінки

На рисунку 3.8 зображено приклад створення та редагування записів. Текстові поля з ручним введенням використовуються для атрибутів, що є унікальними ідентифікаторами: номер аудиторії – довільний рядок, що вводиться вручну, та кількість місць – числове значення. Обидва поля не мають фіксованого набору варіантів, тому вільне введення є єдиним підходом. Поля розміщено у два стовпці в одному рядку, що економить вертикальний простір форми.

Випадаючі списки використовуються для полів з фіксованим переліком варіантів: тип аудиторії та кафедра. Тип аудиторії має три варіанти і відображає їх у розгорнутому вигляді при натисканні. Кафедра також обирається зі списку, що підвантажується з відповідного довідника через сервісний шар. Такий підхід виключає помилки вільного введення для полів з обмеженим набором значень і одночасно гарантує реляційну цілісність.

Прапорці з числовими полями використовуються для переліку обладнання. Кожна позиція обладнання має прапорець вибору та числове поле кількості одиниць поруч, яке стає актуальним лише при встановленому прапорці.

Текстовий пошук над переліком дозволяє швидко знайти потрібну позицію у довгому списку без прокручування. Кнопки «Вибрати всі» та «Зняти всі» забезпечують масовий вибір при первинному заповненні або скиданні даних. Прокручуваний контейнер обладнання обмежує висоту блоку, не збільшуючи загальну висоту модального вікна незалежно від кількості позицій у переліку.

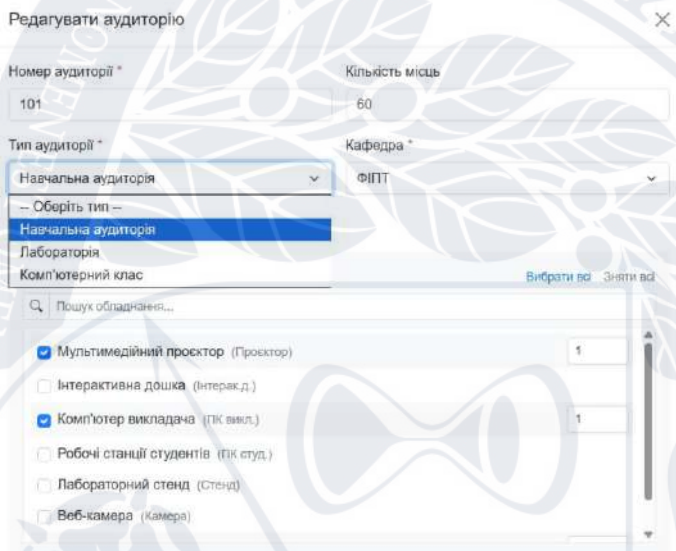


Рисунок 3.8 – Модальне вікно редагування

Повідомлення «Помилка від сервера: An error occurred while saving the entity changes...» відображається у блоці помилки всередині модального вікна – вікно залишається відкритим, поля форми зберігають введені значення, що дозволяє виправити дані без повторного введення. Помилка є результатом перехоплення DbUpdateException на рівні контролера при порушенні унікального індексу бази даних – база даних відхиляє транзакцію і повертає виключення, яке контролер серіалізує та передає клієнту з кодом 400. Такий підхід виключає необхідність додаткової перевірки унікальності на рівні сервісного шару: база даних є остаточним арбітром унікальності, а система лише коректно обробляє її відповідь.

На рисунку 3.9 зображено сітку редактору розкладу через яку здійснюється створення занять для розкладу. Навігація між тижнями реалізована трьома кнопками над сіткою. Кнопки «Назад» та «Далі» зсувають сітку на один тиждень

у відповідному напрямку і блокуються при виході за межі семестру. Кнопка «Сьогодні» активна лише тоді, коли поточний відображуваний тиждень відрізняється від тижня поточної дати. Поруч з навігатором відображається діапазон дат поточного тижня та тип тижня «Навчальний» з відповідним кольором.

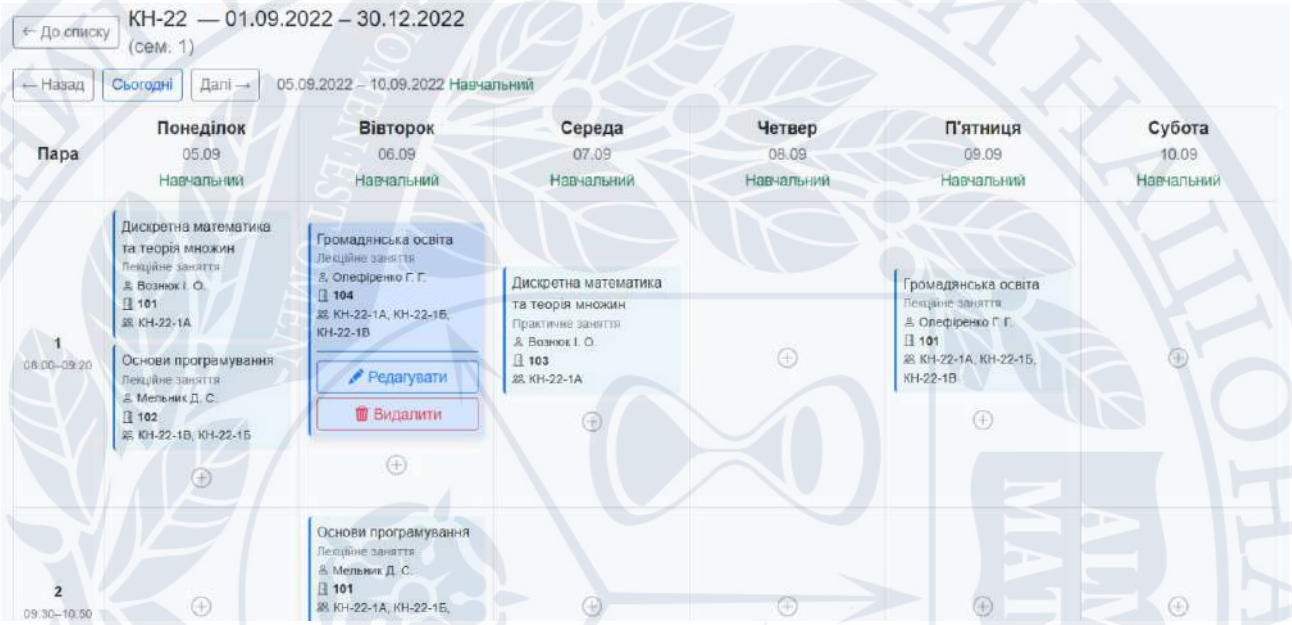


Рисунок 3.9 – Сітка редактора розкладу

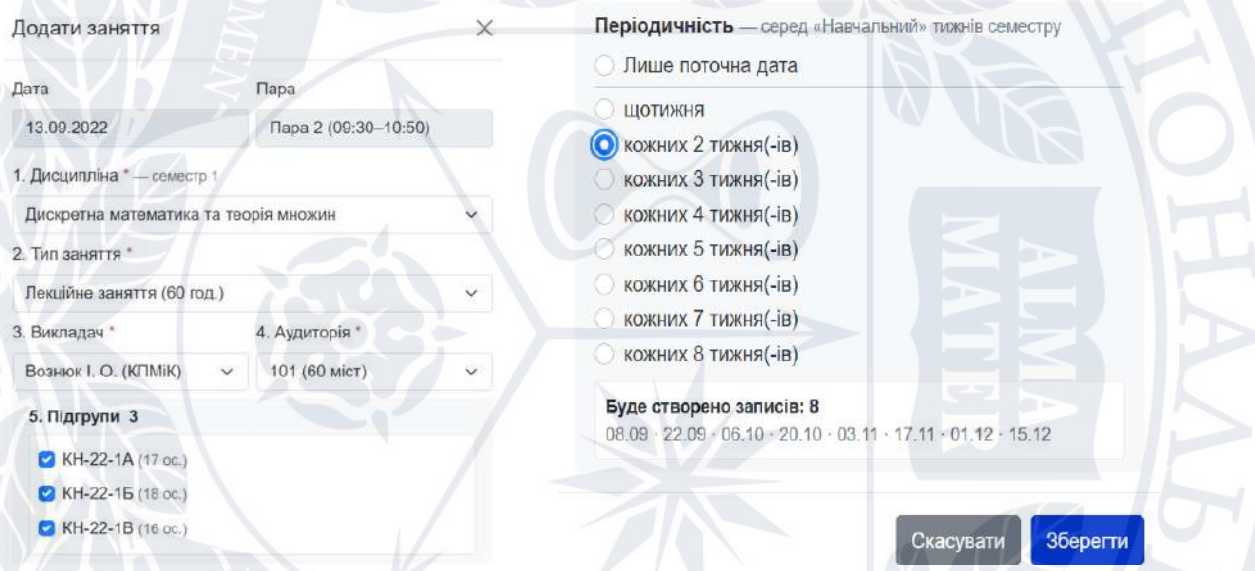
Заголовки стовпців є інформаційними елементами лише для читання: кожен стовпець містить назву дня тижня, дату у форматі «ДД.ММ» та тип тижня для цього конкретного дня. Заголовки рядків містять номер пари та діапазон часу – теж лише для читання.

Порожня комірка є клікабельним елементом: натискання по комірці відкриває модальне вікно додавання заняття з автоматично заповненими полями дати та часового слоту. Дата обчислюється з позиції стовпця, часовий слот – з позиції рядка.

Заповнена картка заняття має два стани. У звичайному стані вона відображає назву дисципліни, тип заняття, викладача, аудиторію та підгрупи – без елементів керування. При натисканні на картку вона переходить у виділений стан: з'являється синя рамка та дві кнопки – «Редагувати» та «Видалити».

Натискання на іншу картку або на порожній простір знімає виділення. Такий підхід запобігає випадковому видаленню при простому перегляді сітки.

Поля дати та пари є лише для читання – вони заповнюються автоматично з комірки, по якій клікнув адміністратор (рис. 3.10.а). Кроки вибору організовано ієрархічно: крок 1 – дисципліна з підказкою номера семестру, крок 2 – тип заняття з кількістю годин, крок 3 – викладач з кафедрою, крок 4 – аудиторія з кількістю місць. Кожен наступний крок активується лише після заповнення попереднього. Крок 5 – підгрупи з кількістю студентів у кожній та прапорцями вибору.



а) б)

Рисунок 3.10 – Вікно додавання заняття

Адміністратор може обрати між одноразовим збереженням та періодичним: щотижня, кожних 2–8 тижнів (рис. 3.10.б). Система автоматично обчислює перелік дат і показує попередній перегляд – «Буде створено записів: 15» з переліком найближчих дат і кількістю прихованих. Це дозволяє заповнити весь семестр для одного заняття за один раз замість поодиноких записів для кожного тижня.

Блок помилки «Деякі записи не збережено» містить перелік конкретних дат та опис порушеного правила для кожної з них – «Студентські групи вже

зайняті у цей час: 6» (рис. 3.11). Записи без конфліктів при цьому також не збережені. Повідомлення «... ще 8» вказує на те, що перелік скорочено і повний список довший. Система відкидає весь пакет при перших помилках і не зберігає коректні записи, що дозволяє передивитися побудову розкладу без подальшого редагування.

Ліва панель містить перемикач режиму та чотири пронумеровані кроки: факультет або кафедра, спеціальність, група, підгрупа як необов'язковий крок (рис. 3.12.а). Нижче розміщено вибір семестру та кнопка «Показати розклад». Кнопка активна лише після заповнення обов'язкових полів.

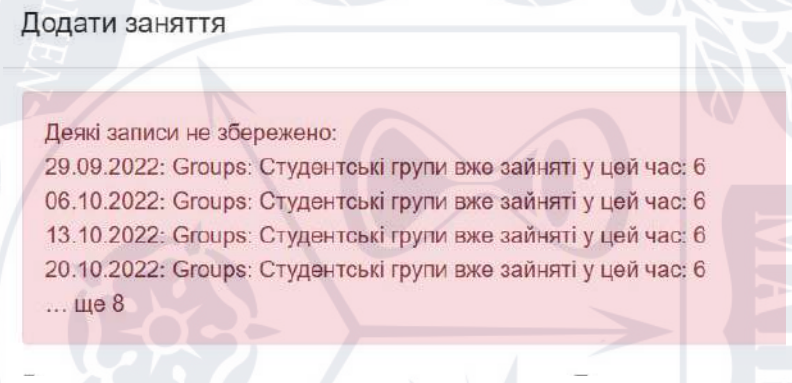
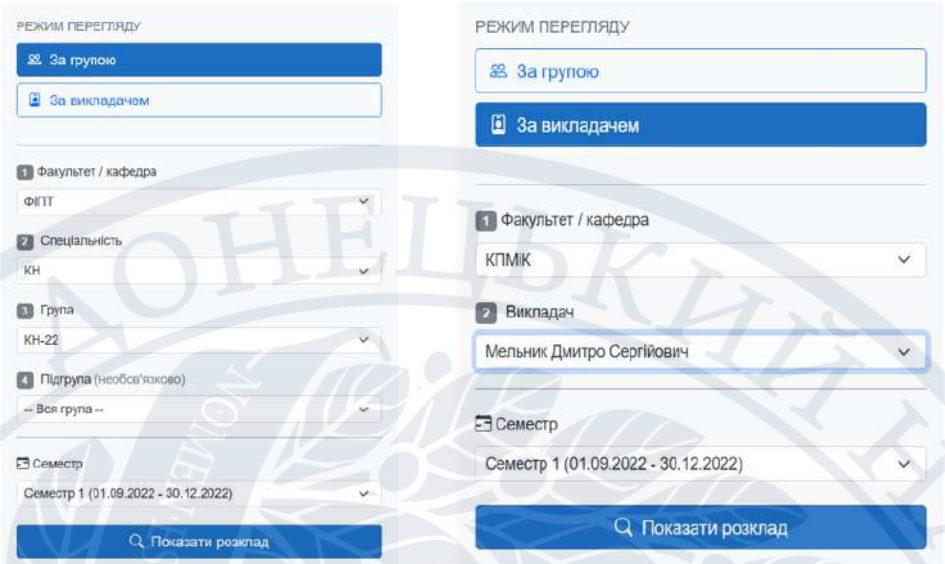


Рисунок 3.11 – Помилка збереження заняття з періодичністю

Ієрархія скорочена до двох кроків: кафедра, викладач (рис. 3.12.б) Режим перемикається кнопками у верхній частині панелі, активний режим виділено синім.

Навігатор тижнів у мобільному вигляді є компактнішим: відображається лише діапазон дат поточного тижня з кнопками «<» та «>» для переходу між тижнями (рис. 3.13).

Мобільний навігатор днів розміщений між навігатором тижнів та сіткою і є унікальним елементом вузького вигляду. Він показує назви та дати двох поточних видимих днів через роздільник «/» та стрілки для посування на один день. При досягненні кінця тижня навігатор автоматично переходить до наступного тижня, при досягненні початку – до попереднього.



а) б)

Рисунок 3.12 – Панель фільтрів публічної підсистеми

Картки занять у мобільному вигляді зберігають повний обсяг інформації – назву дисципліни, тип заняття, викладача з іконкою, підгрупи та аудиторію з іконкою. Зменшення кількості видимих стовпців компенсує обмежену ширину екрану без втрати інформації про конкретне заняття. Іконки викладача та аудиторії замінюють текстові мітки, що дозволяє зберегти щільність інформації у вузькому просторі картки.

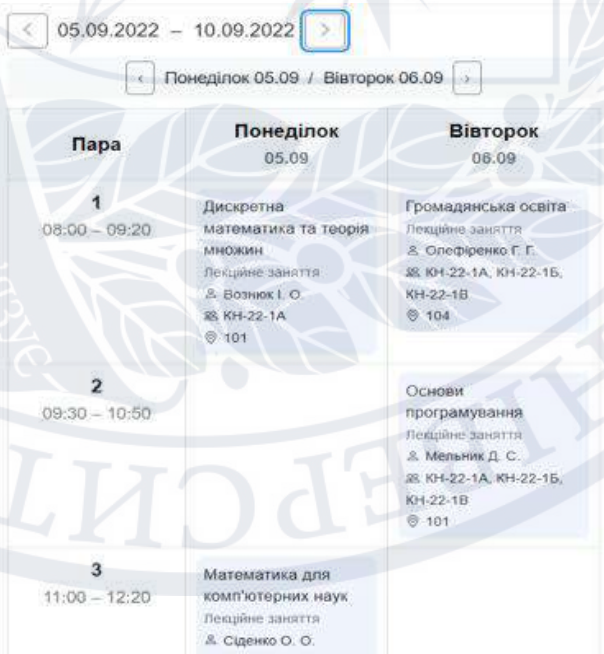


Рисунок 3.13 – Мобільний вигляд фрагменту сітки розкладу

Висновок до розділу 3

У третьому розділі реалізовано всі компоненти системи відповідно до проєктних рішень другого розділу. Організація у шість ізольованих проєктів із спільним контрактом об'єктів передачі даних забезпечила єдність між серверною і клієнтською сторонами та усунула дублювання конфігурації. Узагальнений базовий сервіс підтвердив ефективність: більшість довідникових сервісів зводяться до визначення репозиторію без дублювання операційної логіки.

Модуль валідації зі звітною моделлю забезпечив повний перелік порушень за один прохід та незалежність правил одне від одного. Тонкий шар контролерів із розмежуванням відкритих і захищених маршрутів на рівні атрибутів унеможливлює обхід обмежень доступу.

Адміністративна підсистема покриває повний цикл керування даними з редактором розкладу та автентифікацією, публічна – забезпечує зручний доступ до розкладу без реєстрації з адаптивним інтерфейсом. Огляд готового продукту підтвердив відповідність реалізованого функціоналу вимогам підрозділу 2.2.

ВИСНОВКИ

У кваліфікаційній роботі вирішено задачу розробки веб-системи управління розкладом занять закладу вищої освіти з реалізацією сервісного шару бізнес-логіки, модуля валідації, контролерів прикладного програмного інтерфейсу та двох клієнтських підсистем з розмежуванням ролей доступу.

Аналіз існуючих рішень показав, що жодна з розглянутих систем не поєднує адміністративний інтерфейс із вбудованою перевіркою предметних обмежень та відкритий публічний інтерфейс перегляду, адаптований до специфіки вітчизняних закладів вищої освіти. Виявлені прогалини сформували конкретні вимоги до розробленої системи.

Обгрунтовано вибір технологічного стеку на платформі .NET з фреймворком Blazor. Єдине мовне середовище C# на всіх рівнях системи усуває клас помилок невідповідності між серверною та клієнтською моделями даних і виявляє їх на етапі компіляції.

Спроєктовано та реалізовано архітектуру з шести ізольованих проєктів. Сервісний шар на основі узагальненого базового класу усунув дублювання для двадцяти довідникових сервісів. Модуль валідації з сімома доменно-специфічними правилами та окремим пакетним валідатором забезпечує повний перелік порушень за один прохід і є відкритим для розширення без модифікації існуючого коду. Контракт прикладного програмного інтерфейсу з розмежуванням відкритих і захищених маршрутів унеможливлює обхід обмежень доступу.

Реалізовано дві клієнтські підсистеми. Адміністративна містить редактор розкладу з періодичним збереженням, пакетною валідацією та автентифікацією. Публічна реалізує каскадні фільтри в режимах студента та викладача, адаптивну сітку з мобільною навігацією та автоматичне визначення поточного тижня семестру.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ:

1. Войтко Б.С., Мічківський С.М. Система формування розкладу навчальних занять з використанням платформи 1С: Підприємство. Збірник тез доповідей. Донецький національний університет імені Василя Стуса, 2020. URL: https://jait.donnu.edu.ua/article/view/9021 (дата звернення: 08.03.2026).
2. Гафіяк А.М., Бородіна О.О., Гусак В. Проблеми розробки клієнт-розкладу закладів вищої освіти. Мукачівський державний університет. URL: https://economyandsociety.in.ua/journals/18_ukr/148.pdf (дата звернення: 27.02.2026).
3. Голуб Б.Л., Ветрова Д.В., Пронішина К.О. Програмна система формування розкладу занять у закладі вищої освіти. Математичні машини і системи. 2019. №4. С. 100–109.
4. Донецький національний університет імені Василя Стуса. URL: https://www.donnu.edu.ua/uk/ (дата звернення: 04.02.2026).
5. Жалівців Р. Математичне моделювання процесу складання розкладу в університеті. Науковий блог Національного університету «Острозька академія». URL: https://naub.oa.edu.ua/matematychne-modelyuvannya-protsesu-skladannya-rozkladu-v-universyteti/ (дата звернення: 02.03.2026).
6. Калько Д.Р., Антонов Ю.С. Побудова шару бізнес-логіки багатошарових застосунків на основі Generic класу. Збірник матеріалів науково-практичної конференції. Донецький національний університет імені Василя Стуса, 2026.
7. Комп'ютерна програма: «Shedule Module For Antonov Students Test System». Авт. свід. №103272 від 18.03.2021, видане Українським інститутом інтелектуальної власності; Заявл. №с202100796 від 15.02.2021.
8. Нефункціональні вимоги: приклади, типи, підходи. URL: https://training.qatestlab.com/blog/technical-articles/non-functional-requirements-examples-types-approaches/ (дата звернення: 10.04.2026).
9. Розклад занять студентської групи. URL: https://fosslook.com.ua/articles/class-schedule (дата звернення: 17.03.2026).

10. Римар П.В., Войтко Б.С. Розробка автоматизованої системи формування розкладу навчальних занять з використанням 1С: «Підприємство». Вісник Хмельницького національного університету. 2021. №5. (дата звернення: 08.03.2026).
11. Функціональні та нефункціональні вимоги (з прикладами). Visure Solutions. URL: https://visuresolutions.com/uk/посібник-з-відстеження-управління-вимогами/функціональні-та-нефункціональні-вимоги/ (дата звернення: 28.02.2026).
12. A.S.T.S. Shedule module. URL: http://antonov.donnu.org/ (дата звернення: 15.02.2026).
13. AK Academy. Send Register Request to the API | PlannerApp Full Client-Side Project with Blazor WebAssembly. URL: https://www.youtube.com/watch?v=waG2U9_Wgww (дата звернення: 15.04.2026).
14. Antonov Yu.S., Kalko D.R., Maruniak R.V. 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. 2026.
15. APIs with ASP.NET Core. Microsoft. URL: https://dotnet.microsoft.com/en-us/apps/aspnet/apis (дата звернення: 30.03.2026).
16. ASP.NET Core Blazor. Microsoft Learn. URL: https://learn.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-9.0 (дата звернення: 30.03.2026).
17. blazor-starter-kit. GitHub. URL: https://github.com/fullstackhero/blazor-starter-kit (дата звернення: 30.04.2026).
18. Bogard J. Vertical Slice Architecture. 2018. URL: https://www.jimmybogard.com/vertical-slice-architecture/ (дата звернення: 27.04.2026).
19. Bollens E., Rocchio R.A., Peterson J.E. Understanding Responsive Web Design in Higher Education. ECAR Working Group Paper. 2014.
20. Bootstrap 5 Tutorial. W3Schools. URL: https://www.w3schools.com/bootstrap5/ (дата звернення: 19.04.2026).

21. Casal J. Blazor Full Course For Beginners. URL: https://www.youtube.com/watch?v=RBVIclt4sOo (дата звернення: 12.04.2026).
22. CodeOpinion. DTOs & Mapping: The Good, The Bad, And The Excessive. URL: https://www.youtube.com/watch?v=FKFxWrwdAWc (дата звернення: 11.04.2026).
23. Comprehensive University Timetabling System. URL: https://www.unitime.org (дата звернення: 25.03.2026).
24. Configuration in .NET. Microsoft Learn. URL: https://learn.microsoft.com/en-us/dotnet/core/extensions/configuration (дата звернення: 30.04.2026).
25. Controller in ASP.NET Core MVC. URL: https://csharp-video-tutorials.blogspot.com/2019/02/controller-in-aspnet-core-mvc.html (дата звернення: 30.04.2026).
26. Davison M., Kheiri A., Zografos K.G. Modelling and solving the university course timetabling problem with hybrid teaching considerations. Journal of Scheduling. 2024. Vol. 28. P. 195–215. URL: https://doi.org/10.1007/s10951-024-00817-w
27. Enumerable.Where Method. Microsoft Learn. URL: https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.where?view=netframework-4.8.1 (дата звернення: 16.04.2026).
28. IEnumerable\<T\> Interface. Microsoft Learn. URL: https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.ienumerable-1 (дата звернення: 16.04.2026).
29. Implementing the Repository and Unit of Work Patterns in an ASP.NET MVC Application. Microsoft Learn. URL: 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: 16.04.2026).

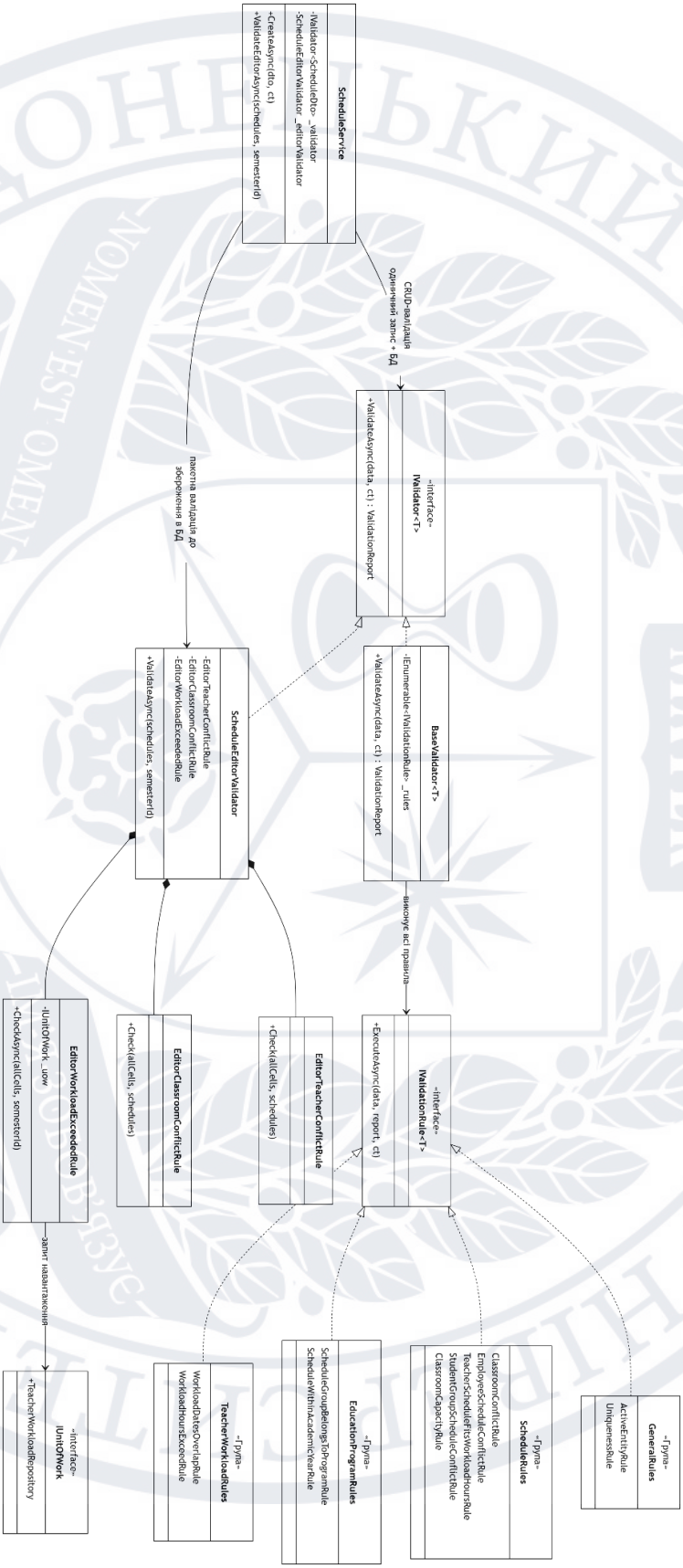
30. Introduction to ASP.NET Web Services. URL: https://csharp-video-tutorials.blogspot.com/2013/11/part-1-introduction-to-aspnet-web.html (дата звернення: 10.04.2026).
31. Kudvenkat. ASP NET Core Identity. URL: https://www.youtube.com/playlist?list=PLcu9C6NEcy-ctJMYJE9wFCKaG4IVn6pMJ (дата звернення: 15.34.2026).
32. Kudvenkat. Blazor tutorial for beginners. URL: https://www.youtube.com/playlist?list=PL6n9fhu94yhVowClAs8-6nYnfsOTma14P (дата звернення: 15.03.2026).
33. Lopateeva O.N. Development of an automated information system to support computational procedures for scheduling. IOP Conference Series: Materials Science and Engineering. 2021.
34. Model-View-ViewModel (MVVM). Microsoft Learn. URL: https://learn.microsoft.com/en-us/dotnet/architecture/maui/mvvm (дата звернення: 29.03.2026).
35. MPA vs. SPA: Technical Comparison of Web Architectures. URL: https://medium.com/@iinanyusuf/mpa-vs-spa-technical-comparison-of-web-architectures-fc5b52a648b9 (дата звернення: 12.02.2026).
36. Müller T., Murray K., Rudová H. Modelling and Solution of a Complex University Course Timetabling Problem. Practice and Theory of Automated Timetabling VI. PATAT 2006. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 2007. P. 189–209. URL: https://doi.org/10.1007/978-3-540-77345-0_13
37. Müller T., Murray K., Schlüttenhofer S. University Course Timetabling and Student Sectioning System. Proceedings of the 12th International Conference on the Practice and Theory of Automated Timetabling (PATAT 2018). 2018.
38. Mykulych O. DRY. URL: https://medium.com/@alexander.mykulych/dry-36c5b74ed6c4 (дата звернення: 17.02.2026).

39. .NET dependency injection. Microsoft Learn. URL: https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection (дата звернення: 17.04.2026).
40. Records (C# reference). Microsoft Learn. URL: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/record (дата звернення: 17.04.2026).
41. Richards M. Software Architecture Patterns. O'Reilly Media, 2015. URL: https://theswissbay.ch/pdf/Books/Computer%20science/O'Reilly/software-architecture-patterns.pdf (дата звернення: 17.03.2026).
42. TimeTable – Smart Timetable. URL: https://apps.apple.com/kz/app/smart-timetable/id1278473923 (дата звернення: 10.01.2026).
43. Tutorial: Create a controller-based web API with ASP.NET Core. Microsoft Learn. URL: https://learn.microsoft.com/en-us/aspnet/core/tutorials/first-web-api?view=aspnetcore-9.0&tabs=visual-studio (дата звернення: 28.03.2026).
44. WebAssembly Web API. URL: https://webassembly.github.io/spec/web-api/index.html (дата звернення: 28.03.2026).

ДОДАТКИ

ДОДАТОК А

Система валідації



ДОДАТОК Б

Блок-схема автентифікації